ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Robot Competition Report

TEAM 4

Bricq Arthur, Jacqueroud Lorenzo, Mussa Leonardo

16th January 2021

Contents

PROJECT LINKS

We have prepared a **video** presenting our project [1] and a **GitHub repository** containing all the code used in the robot [2]

FIGURE 1
Youtube Link

FIGURE 2
Github Repository

[1] Here is the real Youtube link.
[2] Here is the real Github link.

CHAPTER 1

INTRODUCTION

This report will present our robot: **Robottle**, designed to attend EPFL's Robot Competition. Robottle is an autonomous robot able to collect bottles in a random - yet controlled - environment: a 64 squared meters arena. It is mechanically robust and reliable, using a very simple bottle picking mechanism. We have invested a lot of efforts in building the software of the robot, as we wanted to fully explore the potential provided by EPFL for participating students. Robottle is equipped with a **Jetson Nano Board**, a **Lidar** to perform localization and mapping of the environment, and a **camera** that runs a neural network on the computer's **GPU** for accurately detecting bottles. We encourage readers to give a look at section 5.2 which presents the main components of the software's logic.

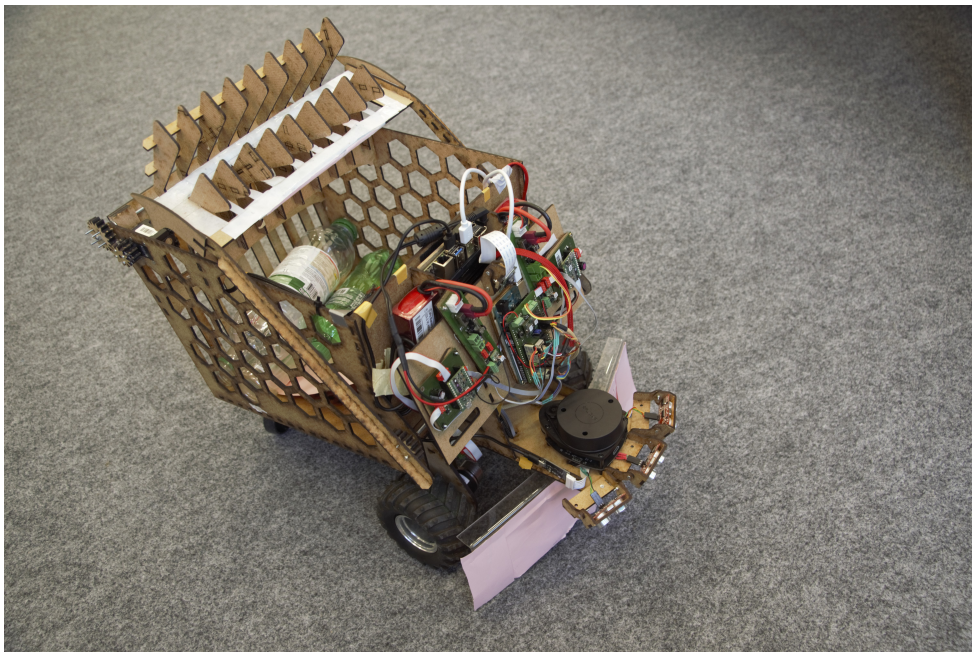


FIGURE 1.1
Robottle

1.1 THE COMPETITION

The Robotic Competition is a semester master project proposed at EPFL. The objective is for 6 teams of 3 students each to build a fully autonomous robot, who's goal is to collect PET-bottles inside an arena and

bring them back to a recycling area. In order to navigate this environment, the robot must be able to move on its own, avoid randomly placed obstacles and bring the bottles to the designated area with its own means. The arena is an 8 by 8 meter square, divided in 4 different zones, where bottles are worth more points depending on the difficulty of access. Zone 1 is the main area, zone 2 contains grass, zone 3 lays behind a rock barrier and zone 4 is accessible only through a ramp or through stairs, and the bottles are worth 10, 20, 40 and 40 points respectively in each of those zones. At the end, the robot which collects the most amount of points wins the competition.

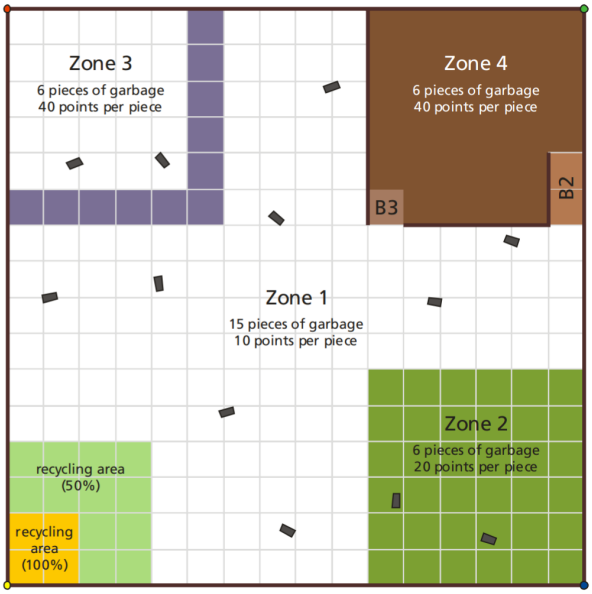


FIGURE 1.2
The arena

1.2 THE TEAM

Our team name is Robottle and we are three students, in their third semester of the master program at EPFL. Arthur Bricq is currently studying Robotics, but has a Bachelor's degree in Mechanical Engineering, Lorenzo Jacqueroud is also in the Robotics's section, with a background in Microengineering and Leonardo Mussa is in Electrical Engineering.

(A) Arthur Bricq

(B) Lorenzo Jacqueroud

(C) Leonardo Mussa

FIGURE 1.3
The Team

1.3 GOALS

The main objective of this project was to win the final competition, by collecting the most amount of points possible. We wanted to create a robot able to **localize** itself precisely in the arena, and have a **smart way of moving around** to grab the bottles, while also capable of going either in zone 3 or zone 4. With this in mind, it was still very important for us to learn a lot and **gain experience** in the field of robotics.

CHAPTER 2

STRATEGY

2.1 GENERAL STRATEGY

During the early stages of the project, we discussed a lot within our group and we tried to get some insight from people who had participated to the competition in the previous years. One thing that seemed to keep coming out from the discussions with our more experienced friends was that robots that failed during the competition, did so because of some kind of **mechanical failure** or because the design was so complex that it became quickly overwhelming, because of the limited time available. Given that none of us has a strong background in building robots we decided to keep our mechanical design **as simple as possible**, and to focus instead on the sensing and control aspects of the robot. As we will see, this turned out to be a wise choice as even building the simplest of robots turned out to be quite challenging, but due to the simplicity of our design we were able to build a functional prototype rather quickly. Also we were able to spend more energy on the **localization** and **bottle-detection** algorithms and to build a **robust** and **stable** code.

Since our strengths lie in the bottle localization we decided **not to go** in the **elevated zone** (zone 4), because our tests revealed it would require a very precise control of the robot to be able to climb the ramp. We chose to focus on the **grass area** (zone 2) and the **stone barrier** instead (zone 3), which are also very attractive since the bottles are worth 20 and 40 points respectively.

Our general strategy is to head for zone 1 and 2, collect as much bottles as possible and return to the recycling area. If there is still some time left, the same strategy should be repeated again in the same zones and eventually for zone 3, as shows in Fig. 2.1. There is a considerable risk of the robot getting stuck when crossing the stone barrier, and that is why we decided it would be wiser to first secure points from the other zones and if there is still time, go for the high-risk high-reward option.

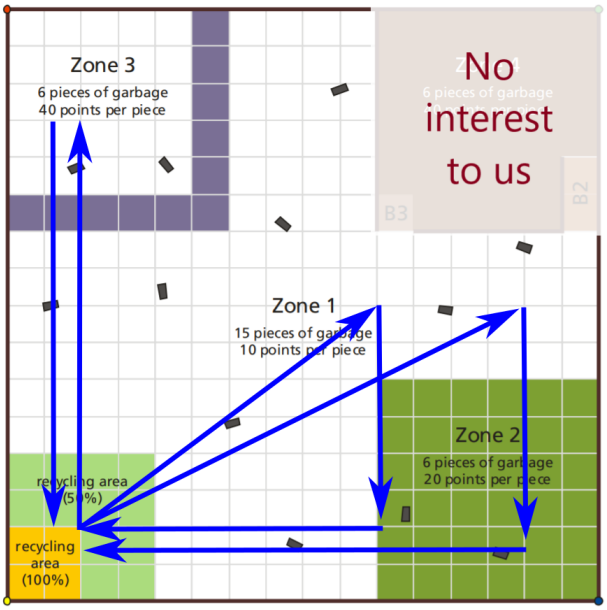


FIGURE 2.1
The path followed by the robot

We designed our robot to have a capacity of about 7-8 bottles and we planned to use an ultrasonic sensor to count the number of bottles picked (more on this later). We chose to implement a very simple heuristic rule to pick up the bottle: once the robot has reached its destination (the center of the target zone) it rotates until it detects a bottle, which has to lie within a certain distance from the robot otherwise it is ignored. As soon as the robot detects a bottle, it aligns itself and then moves in a straight line to reach it and grab it. This procedure is then repeated until the robot is full or the time is running out.

2.2 LOCOMOTION

In the beginning we considered a 6 wheels solutions but eventually we opted for a 4-wheeled robot, because it seemed to be the most effective and simple solution. We also looked into the possibility of using caterpillar tracks but we concluded they are too slow and significantly more complex. Because of the stone barrier, we hesitated for a long time between a 2-driving wheel solution and a 4-driving wheels one, however we eventually opted for the first option, keeping faith to our "the simpler, the better" motto. Another important advantage of this solution is that, as the two back wheels are caster wheels, the robot can turn on itself very easily, as opposed to the 4-driving wheels prototype, which had some trouble in doing so.

In conclusion, our robot consists of a rigid wooden chassis with two driving wheels in the front and two caster wheels in the back. To make sure we were able to pass the rocks we chose wheels with good grip and a large diameter and we put more weight in the front of the robot to keep good traction. We chose to put 2 passive wheels in the back instead of only one to have more stability and because our tests showed that they wouldn't be too much of an hindrance when crossing the rock barrier.

	Pros	Cons
2 wheels drive	**Easy to control** **Simple to implement**	**Hard to go in zone 3**
4 wheels drive	Good grip and strength	Turning can cause drifting
6 wheels drive	Very good grip and strength	Contact points hard to manage Turning can cause drifting
Caterpillar Tracks	Good grip and strength	Complicated to implement Slower

TABLE 2.1
Locomotion analysis

2.3 LOCALIZATION

For the localization we were looking for a solution that could be as robust and flexible as possible. This is why we chose to implement a SLAM algorithm based on the output of a lidar, which is able to localize the robot and build a map of the arena **without using the beacons**. In the beginning we weren't sure of the final precision that we could obtain with the SLAM, so we kept the localization with the beacons as a backup solution. As we will show, this turned out to be unnecessary, because the algorithm performance proved itself to be very good.

	Pros	Cons
Beacons	Easier to implement	Might not be very precise Requires wide range vision
SLAM	**Very powerful and more precise**	**Software is a lot harder** **Requires hardware (Lidar)**

TABLE 2.2
Localization analysis

2.4 OBSTACLE AVOIDANCE

Since we are already building a map of the environment with SLAM, a possibility for avoiding obstacles was to simply use that to check if there was something in the path of the robot. This though could be prone to errors due to imprecisions in the SLAM, therefore we decided to instead use directly the Lidar data. Another possibility was to use the ultrasonic sensor, but since the Lidar is already present because it's needed for the SLAM, we opted for the more precise method.

	Pros	Cons
SLAM map	Already implemented for localization	Might not be very precise
Lidar	**Precise** **Already present because of SLAM**	**More computation required**
Ultrasonic Sensors	Easy to implement	Hard to distinguish bottles and obstacles

TABLE 2.3
Obstacle Avoidance analysis

2.5 BOTTLE LOCALIZATION

Since the bottle localization is a critical part of the project, we tried to implement a very robust method which could detect the bottles with a high certainty. Using ultrasonic sensors, although being a very simple method, requires a lot of hand-tuning and can turn out to be very unpredictable. For this reason we decided to equip our robot with vision: we chose to use a Raspberry-PI camera and a neuronal network to recognize the bottles and locate them. Unfortunately, the neuronal-network proved to be too computationally expensive to be run on a RaspBerry-Pi and we took the (hard) decision to invest in a Jetson Nano, which contains a GPU and was able to reach a faster FPS.

	Pros	Cons
Neural Network	**Precise** **Long range**	**Computationally intensive**
Ultrasonic Sensors	Easy to implement	Not very precise, would need a lot of sensors Hard to tune to not see the ground and have a good range

TABLE 2.4
Bottle Localization analysis

2.6 BOTTLE PICKING

For the bottle picking mechanism we considered many different options, but we discarded most of them because they involved some kind of complex mechanism which, as we explained, goes against our design philosophy. in the end we opted for a very unsophisticated solution: a sticky arm with a pad on one extremity that can be lowered to grab a bottle. The arm has only one degree of freedom (rotation) and the bottle is collected in the following way: the arm is lowered until it touches the bottle, which will (hopefully) stick to it. The arm is then raised and the bottle is detached from the sticky arm by a passive mechanism.

	Pros	Cons
Sticky Arm	**Simple**	**Reliability of the sticky tape**
"Eat" the bottles	Easy to implement and control	Harder to go to zone 3
Mechanical claw	More reliable	Hard to implement
Conveyor belt in front	Easy to control	Hard to implement
Push the bottle	Easy to implement and control	Harder to go to zone 3

TABLE 2.5
Bottle Picking analysis

2.7 BOTTLE RELEASE

For the bottle release mechanism we chose the simplest and easiest to implement option: the bottle are released by opening a door in the back of the robot and since its floor is slanted the bottles should slide outside of the basket without the need for additional actuators. In case the bottles got stuck, the robot should use its driving wheels to shake and expel them.

	Pros	Cons
Back Door with slanted floor	**Easy to implement**	**-**

TABLE 2.6
Bottle Release analysis

2.8 TEAM MANAGEMENT AND TIMELINE

Team management is a very delicate issue, which could determine the success or failure of a project. In our case we thought the most efficient way to deal with this was to define the responsibilities of each team member at the beginning of the project, so that each of us could work independently from the others. Over the course of the semester we met regularly to check on our progress and discuss possible problems/solutions, and after each meeting we tried to assign ourselves a specific task with an indicative deadline. Of course things never go as expected, so we kept things informal and we avoided writing things down explicitly, which granted us more flexibility and created a positive team atmosphere. The team responsibilities were chosen as follows:

- Arthur: software
- Lorenzo: electronics and motors
- Leonardo: mechanics and electronics

In the end, the time management was handled well, and although the last week was very intense, there were no delays and we concluded all the work we set to do in our plans. The time division over the semester was as follows:

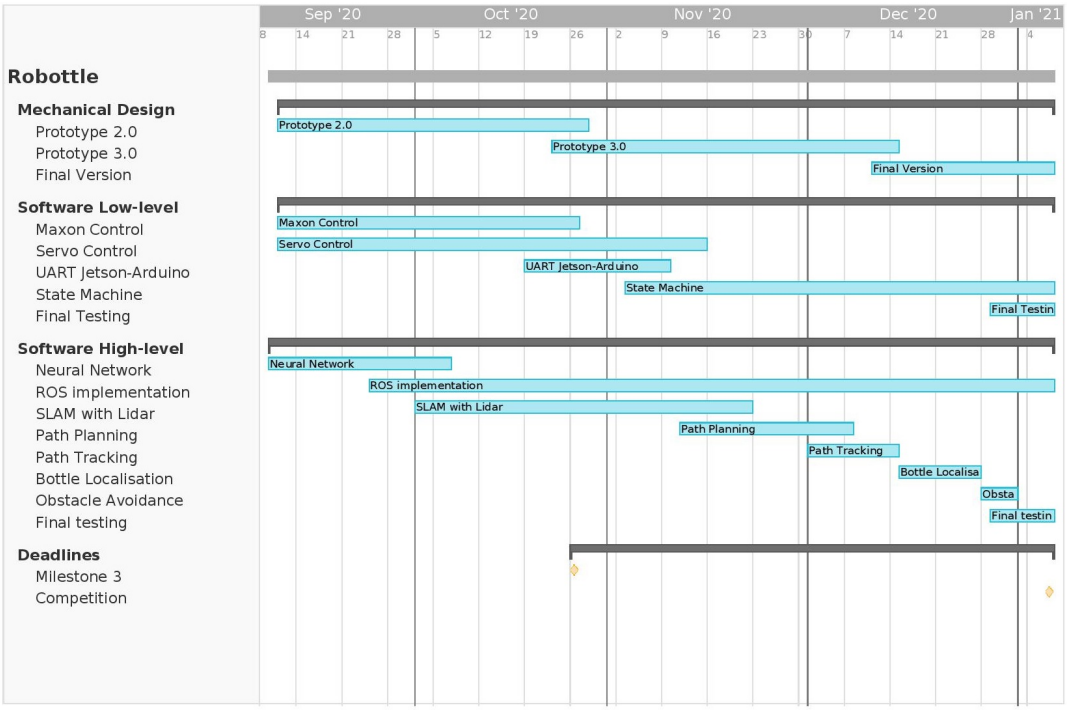


FIGURE 2.2
Gantt chart

More details on the timeline in Section 7.2.

CHAPTER 3

MECHANICAL DESIGN

Over the course of the semester we built a total of 4 prototypes, starting from a very rugged chassis, and ending with a perfected design, fruit of the experience accumulated from its predecessors. We are well aware that this is a rather large number, and that this proliferation of robots may raise some questions in the reader, who may wonder why we didn't choose to build a definitive version earlier. The reason of this is very simple: given our limited experience with real world mechanical design, we decided to approach the problem in an **iterative manner**. Looking back, this was the winning strategy because, as we came to realize multiple times, it is extremely difficult to **predict everything** that could go wrong, and problems tend to appear in the most unpredictable way. Following these guidelines, we tried to increase the complexity of our prototypes incrementally, and to test basic functionalities before moving on to more sophisticated aspects of our robot.

3.1 ROBOTTLE 1.0

The very first prototype we made was assembled extremely quickly and we didn't even make a 3D drawing of it. Its main purpose was to test the locomotion of the robot, namely the strength of the motors and the quality of the wheels.

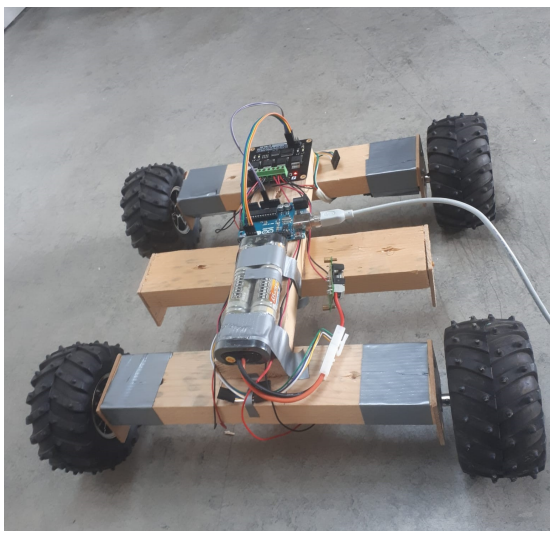


FIGURE 3.1
Robottle 1.0

RoBottle 1.0 was very helpful because it guided our choice about the number of wheels (initially we considered 6 driving wheels, which looking back sounds crazy) and it allowed us to get a feeling for obstacles like the ramp or the stone barrier.

3.2 ROBOTTLE 2.0

Once the wheels were tested we put more effort in building a solid chassis and designing a platform to mount the Lidar and start testing the other functionalities of the robot. Even though RoBottle 2.0 was still a very rough prototype, it allowed us to start working on the SLAM and the motors control and it showed to be a good starting point for the successive prototypes. From a mechanical point of view, it gave us plenty of insight over the process of designing and assembling a robot, and it helped us familiarize with wood, which is a living and sometimes unpredictable material. Another important lesson that we learned while building this prototype was the need for **proper routing** of the connections, as we will see, this turned out to be one of the main challenges we had to face during the design of our successive prototypes.

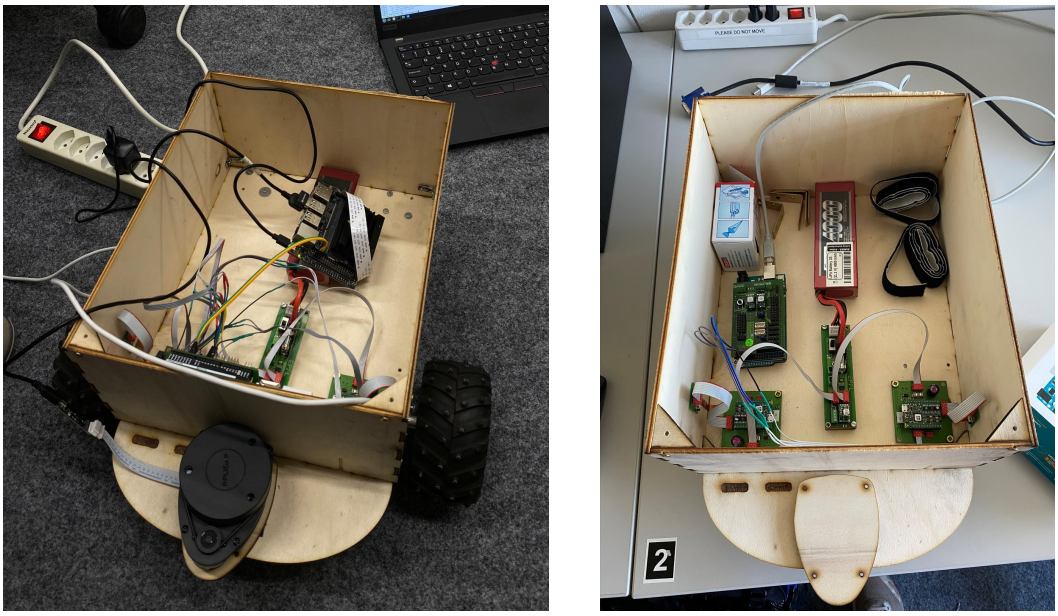


FIGURE 3.2
Robottle 2.0

3.3 ROBOTTLE 3.0

This was the first 'stable' version of RoBottle, in the sense that all of the robots functionalities were implemented: chassis for the locomotion, a platform for the Lidar, the arm and the door. The prototype was designed to test high level tasks of the robot, such as picking up a bottle or moving autonomously within an environment. Even though we built the prototype rather early (around mid-november) we encountered some problems with the motor control, and the complexity of the software was such that we weren't able to merge the various functionalities of the robot into a single working system. In addition, we decided to leave all of the electronics very accessible in the front, since as we noticed with the previous prototype, having the electrical connections hidden, would have made the testing and debugging phase a lot less practical.

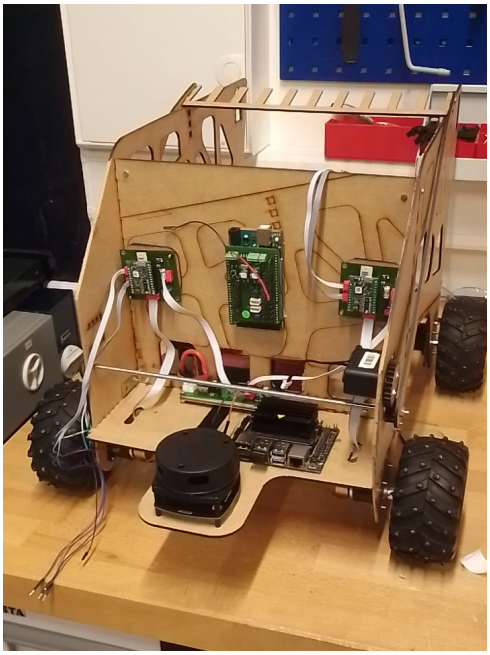


FIGURE 3.3
Robottle 3.0

3.4 FINAL VERSION

You may have noticed that our robot is made **entirely out of wood** and it doesn't include any 3D-printed part. This is a design choice that was made in the beginning, after we realized that there was no need for complex shapes and that laser-cutting is definitely faster than 3D-printing, which turned out to be very helpful during the prototyping stage (another, less pragmatic, reason is that we liked the idea of a 100% recyclable robot).

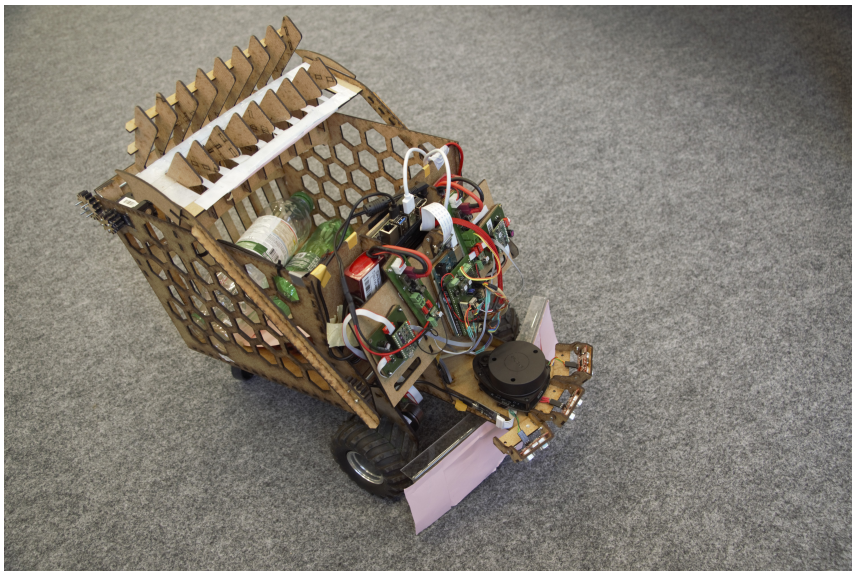


FIGURE 3.4
Final Version of Robottle

The last, final version of RoBottle is more compact, robust and most importantly, mechanically rigid than its predecessor. The arm was reinforced and the release mechanism underwent various modifications to optimize the bottle picking rate of the robot. The Lidar platform was stiffened to prevent oscillations. The gears were improved, as well as the connections between the axes and the wooden parts. Lastly, we built an adjustable support for the camera and for the ultrasonic sensors, which proved themselves to be very useful during the final tuning phase before the competition.

3.4.1 CHASSIS

The chassis is basically a box with a slanted floor, assembled with glue and interlocking joints. On the bottom there is a structure to stiffen the frame and to support the wheels.

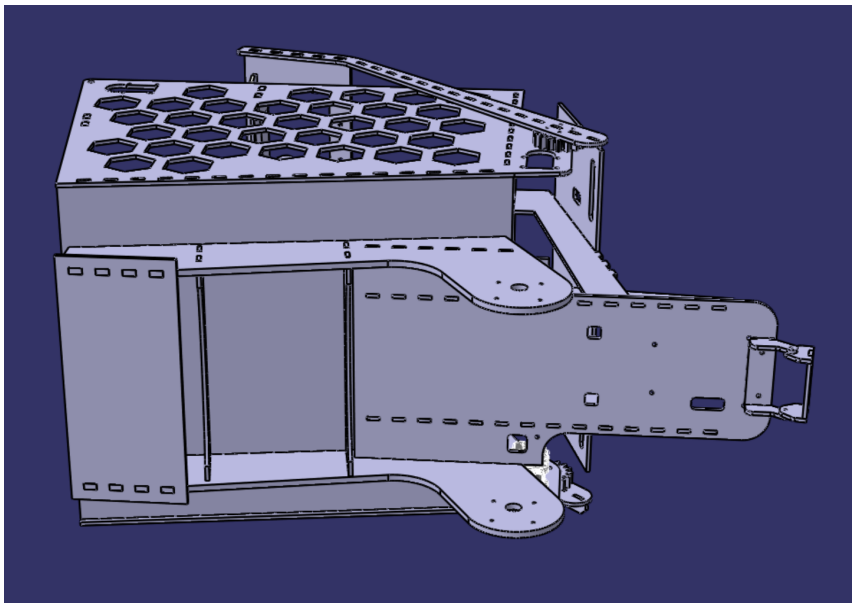


FIGURE 3.5
A view on the bottom of the robot. Notice the perpendicular reinforces used to stiffen the frame where the wheels are attached.

3.4.2 ARM AND RELEASE MECHANISM

The arm is directly screwed onto the gears, which are joined by a metal threaded rod. The sides of the arm are T-shaped to reinforce it and keep it straight, this was important because the release mechanism fits tightly around the arm and a slight misalignment could result in malfunctioning.

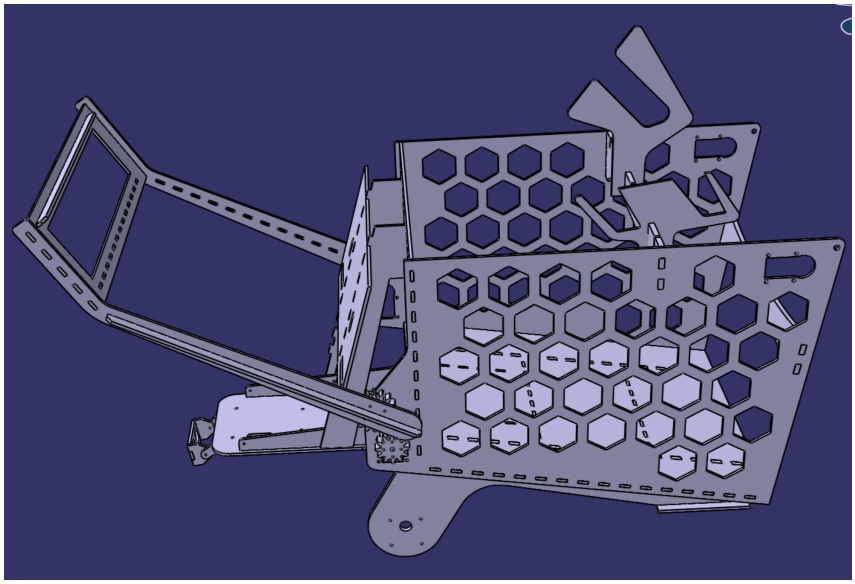


FIGURE 3.6
A view on the arm. Notice that the pad at the end of the arm is designed to be covered with a layer of foam covered by sticky tape. This is to make the interface between the bottle and the arm softer and increase the bottle picking rate. Also, the rectangular space in between should be filled with stripes of sticky tape.

3.4.3 BOTTLE RELEASE

When the arm is raised, it ends up inside of a 'Y' shaped structure, which causes the release of the bottle. Note that the shape of the release mechanism had to undergo some modifications before the correct shape was found.

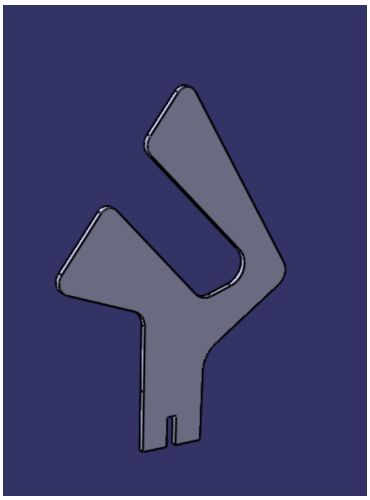


FIGURE 3.7
The shape of the detaching mechanism which was reached after a few, less usccessful iterations.

CHAPTER 4

ELECTRONICS

In the following section we will describe all the electronic equipment that was used to build the robot, such as microcontrollers, actuators, sensors etc.. We will start with a brief overview of the system as a whole and we will continue with a more detailed discussion over the advantages/disadvantages of each component and the reasons that led to its choice.

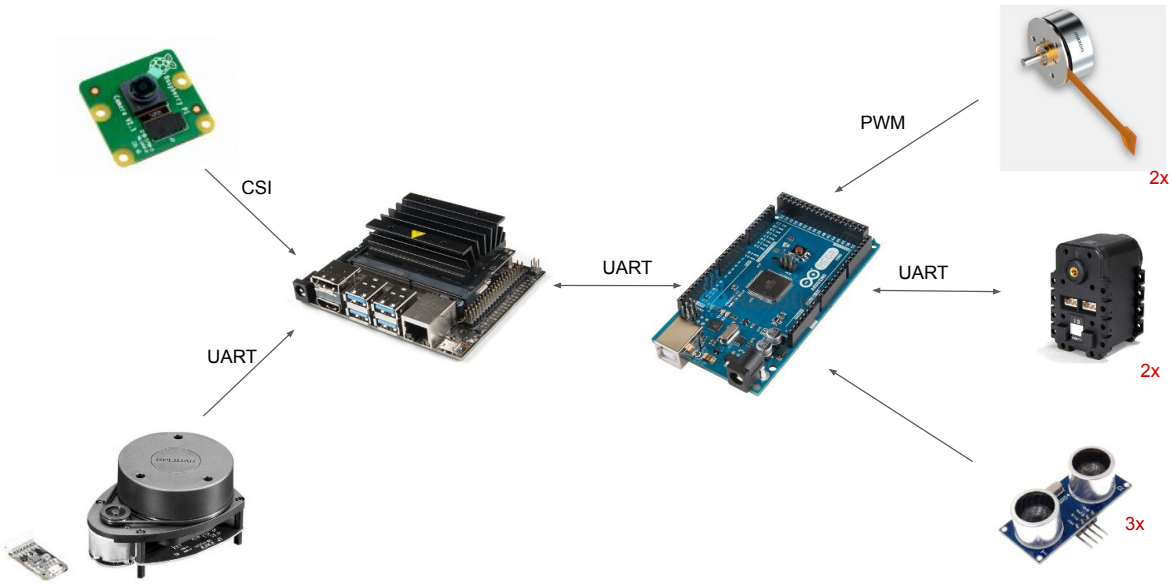


FIGURE 4.1
Diagram representing all the electronics component of the robot.

4.1 GENERAL DESCRIPTION

The robot contains two processing units: a Jetson Nano and an Arduino Mega, the former responsible for control and computations, the latter responsible for actuation and low level control. There are 2 Maxon Motors for locomotion and 2 Dynamixel Servo motors to control the arm and the back door. Additionally there are 3 ultrasonic TOF sensors, a Lidar and a camera.

	Name	Function
CONTROL	Jetson Nano	High level control, bottle detection, SLAM, strategic planning
	Arduino Mega	Low level control of motors, ultrasonic sensors
ACTUATORS	Maxon Motors (2X)	Locomotion
	Dynamixel Servos (2X)	Arm, back door
SENSORS	RP Lidar-A1	SLAM, obstacle detection
	Raspberry Camera	Bottle Detection
	Ultrasonic Sensors (3X)	Bottle and obstacle detection

4.2 CONTROL

Drawing an analogy with nature, the control system of a robot can be seen as the nervous system of an animal. As their biological counterpart, the circuits of a robot need to sense the environment, interpret the data, come up with a strategy and to send signals to the actuators to put it in place. In our particular case, the role of the "brain" is taken by a Jetson Nano, which interprets the sensor data (lidar and camera), runs the high level controller and sends the commands to the other microcontroller, the Arduino Mega. This last can be seen as the "spinal chord" of the robot, whose role is to interface the Jetson Nano with the hardware and to perform simple tasks, like moving the arm to pick a bottle or use the ultrasonic sensors to locate a target. Note that the Arduino has a very limited knowledge of the robot's state and surroundings, and all the high level decision making is done by the Jetson Nano. It is only capable of running sequences of pre-programmed actions which obey to simple logic, and that must be initiated by the high level controller hosted on the Jetson. Continuing the biological analogy, these predefined actions can be seen as "innate reflexes" that can be performed without intervention of the brain, like for example withdrawing the hand from the fire. They are fast, highly optimized and efficient, however they lack a bit of flexibility and are not very adaptable.

4.2.1 JETSON NANO

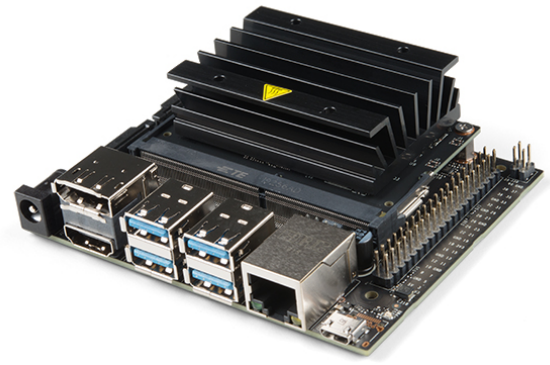


FIGURE 4.2
The Jetson Nano

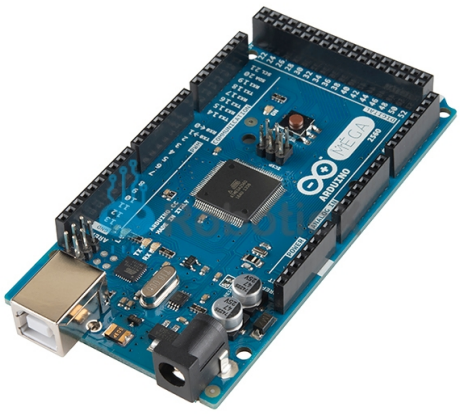
The decision of the main computer was a rather long process that we started at the very beginning of the semester. One of the orientation that our team took was to invest a lot in the software components of the robot. We believed that the opportunity provided by the competition is one of a lifetime, and that it was worth trying to build a robot equipped with **state of the art** technologies. Those technologies involve Deep Neural Networks, the use of a Lidar for simultaneous localization and mapping (SLAM) and of several other sensors, and the use of ROS for handling those many asynchronous tasks.

We chose to use the **Jetson Nano** as our main computer since we needed a lot of processing power to run all of this. When we performed early tests of the Neural Network with a Raspberry Pi - and by then, it wasn't sure yet if we wanted to do all of the mentioned tasks - we realized the detection was way too slow to be useful in practice (0.5 FPS, at maximum). However, the performances of the neural network were pretty much incredible (see Section 5.2.4). This is initially why we chose to invest part of our real budget to buy a Jetson Nano.

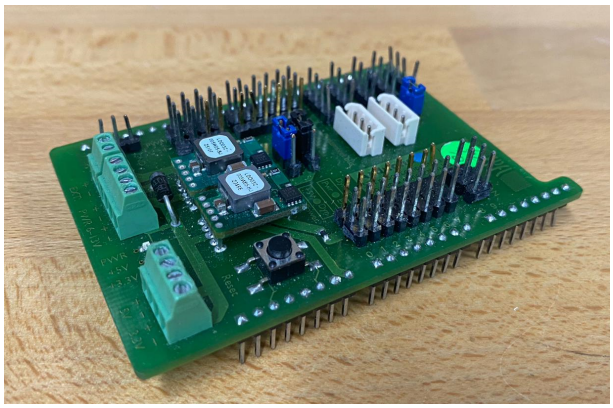
A Jetson Nano is an embedded system-on-module including a 128-core Maxwell GPU, a quad-core ARM A57 64-bits CPU and 4GB of RAM Memory. Similarly to the Raspberry Pi, it runs under a Debian distribution of Linux (however it is not Raspbian like the raspberry pi's, but it is a lightweight version of Ubuntu), it has a 5 to 10W power consumption, it has a connector for Raspberry Pi Cameras and it contains GPIOs directly accessible from the board. Overall, we can say that a Jetson Nano is a stronger version of the Raspberry Pi. Using the power of the GPU with the framework CUDA done with TensorRT, we could increase the detection speed by a factor of 40. As a side note, we can mention that this device produced by NVIDIA comes with an **extensive open-source documentation** with many examples.

We highly recommend future teams reading this report to use a Jetson Nano. It may require to gain some Linux experience, however the power provided by this device is very enjoyable.

4.2.2 ARDUINO MEGA



(A) The Arduino Mega



(B) The Arduone PCB

The Arduino Mega was chosen because of its flexibility and the big number of pins available for connections. Indeed the Arduino boards are some of the best known microcontrollers for prototyping and our case was no different. In addition, with the Arduone board, a PCB designed to go along with the Arduino Mega, we had all of the needed connections and pins for our needs, in particular both digital and analog pins and the Spox (3 pin) for connectors for the servo motors (Section 4.3.2).

4.2.3 COMMUNICATION BETWEEN MICROCONTROLLERS

FIGURE 4.4
The cable used to connect the two microcontrollers

We first tried connecting the Jetson and the Arduino with the UART pins and a voltage converter between the two, since they work at 3.3V and 5V respectively. Unfortunately, the vibrations and the motion of the robot made the communication unstable and too many packet losses were occurring. Therefore we decided to switch to a simple USB type A to B cable.

4.3 ACTUATORS

4.3.1 MAXON MOTORS

FIGURE 4.5
The Maxon motors which were used for locomotion

Among the available motors in the EPFL catalog, the Maxons were by far the best option. During the early testing phase, the Pololu were tested, but without good results. They lacked the torque needed to be able to go over obstacles with just two driving wheels, therefore we quickly switched to the more powerful motors.

4.3.2 DYNAMIXEL SERVOS

FIGURE 4.6
The Dynamixel AX-12A which were used to control the arm and the back door

As the only tasks to be done with a servo motor were rotating the arm to catch the bottles and open and close the back door, there was no need for a particular type of servo. The Dynamixel AX-12A were our first tried option in the available catalog, and it worked well until the end.

4.4 SENSORS

4.4.1 LIDAR

FIGURE 4.7
The RP Lidar A1 used for the SLAM

As mentioned above (Section 2.3), for the localization we chose to implement a laser based SLAM. In order to obtain the data needed to operate the algorithm, we purchased an RPLidar A1, which is one of the cheapest 2D Lidar on the market. Readers would be surprised by the quality of the sensing for the relatively cheap price of the device. The producer's website explains in more details the product.

This sensor scans the environment using a rotating laser and returns a list of angles and distances measured, with for each distance a quality of the measurement associated. It has a **12 meters range** and rotates at a frequency of **5.5 Hz** (this means a total of about 2000 recordings per seconds). There are several open source libraries to work with this device in different languages. We use this library which has an MIT License.

4.4.2 CAMERA

FIGURE 4.8
The RaspBerry PI camera used for the bottle detection

For the camera, we chose a RaspBerry Pi camera v1, compatible with the Jetson Nano.

4.4.3 ULTRASONIC SENSORS

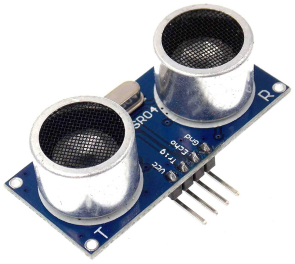


FIGURE 4.9
The ultrasonic sensors used for bottle and obstacle detection

Three ultrasonic sensors from Boxtec were chosen. They have a range of up to about 6 meters, but for our purposes we only used them to detect object closer than 30 cm.

4.5 BATTERIES

FIGURE 4.10
The LiPo battery

To power the entirety of the robot we use two LiPo batteries 11.1V. One is powering the Jetson and all of the connected components (Lidar and RPi camera), while the other powers the Arduino, the ultrasonic

sensors and all the actuators. Even though for the run at the final competition which lasted 10 minutes we could have used only one battery, having two greatly increased the life span of our robot for testing purposes.

4.6 ELECTRICAL CONNECTIONS

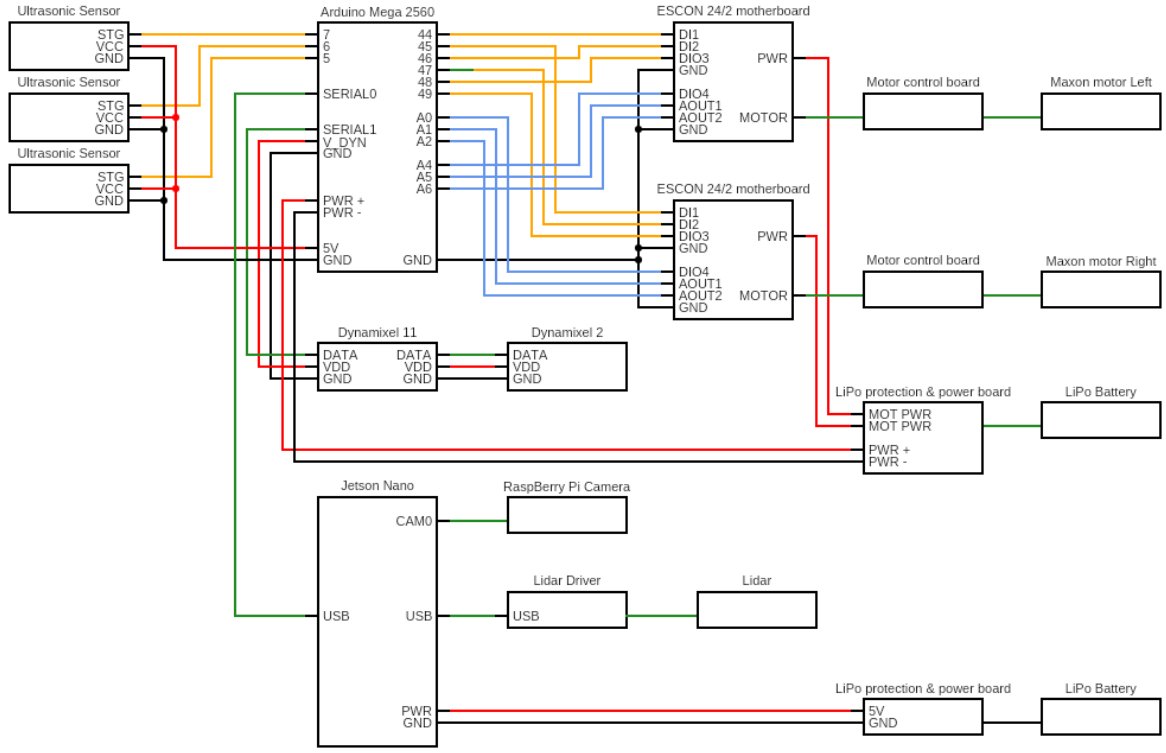


FIGURE 4.11
The diagram of the electrical connections. Black is the Ground, red is for High voltages and other colors have no particular meaning.

CHAPTER 5

SOFTWARE

5.1 SOFTWARE ARCHITECTURE USING ROS2 ON JETSON NANO BOARD

Although the final objective of this project is winning the competition, a very important goal to us was to learn a lot in trying to do so. That is why we decided to build a complex and robust software, incorporating many algorithms that are currently the **state of the art** in problems similar to the ones we are faced wirh. We did a lot of research and found 4 main components that we wanted to integrate in our robot's software based on the problem we were faced with:

- **SLAM** for localization and mapping (Section 5.2.1)
- **RRT-star** for path planning (Section 5.2.3)
- **Computer vision** for bottle detection (Section 5.2.4)
- **ROS** for the implementation

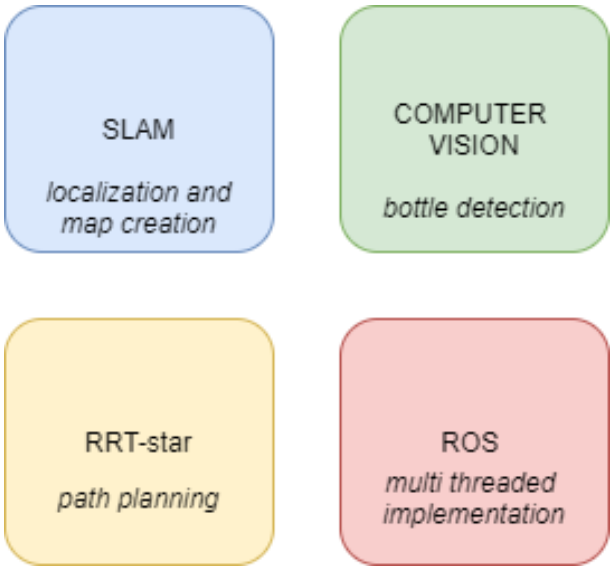


FIGURE 5.1

These elements are some of the most advanced techniques in their respective fields, and although it was a big challenge to make everything work, this taught us a lot.

5.1.1 WHY USING ROS ?

In previous courses at EPFL we learned the existence of ROS and had a very brief overview of its capabilities. So we decided to dig deeper and understand better why it is so widespread in the Robotics community and what are the advantages and disadvantages. We found three main advantages: multi-thread handling, modularity and the ability to save recordings and replay them live easily, and with retrospect, we think that this was very accurate.

1. **Multi-thread handling:**
   With the communication infrastructure of ROS, in particular the publishing and subscriptions of messages, it becomes practical and intuitive to handle asynchronous activity. This allowed us to have many threads working in conjunction one with the other, such as the main controller, the SLAM, the detection of bottles, the communication with the Arduino and more.

2. **Modularity:**
   ROS allows the user to implement only the parts of code that are useful to him. With the community packages that offer many different capabilities, this makes for a very modular system, easy to change and reuse quickly.

3. **Ability to save recordings and replay them live easily:**
   A big feature of ROS is the possibility of recording the messages sent from one node to another and in a later stage, resend them to the same nodes. Thanks to this, we were able to record a test run with the robot in the arena, and later replay the exact same run on our computer, in order to debug or even modify the code.

As a note, although we are happy to have used ROS for this project, there is a considerable disadvantage, which is the setup. It is often complicated and tedious to launch all of the nodes required. Even by using *launch files* to reduce the amount of manual commands to write, there is still the need of having multiple terminals open at once in order to manage all of the verbose of each node for debugging.

5.1.2 ROS DIAGRAM

The following diagrams represent the ROS Nodes and the Topics allowing asynchronous communication between nodes. Nodes can publish or subscribe to Topics and they have configurable parameters associated to them. In term of code, a ROS Node is a Python Class that inherits from the super-class **rclpy.node** and that is launched from a Python script.

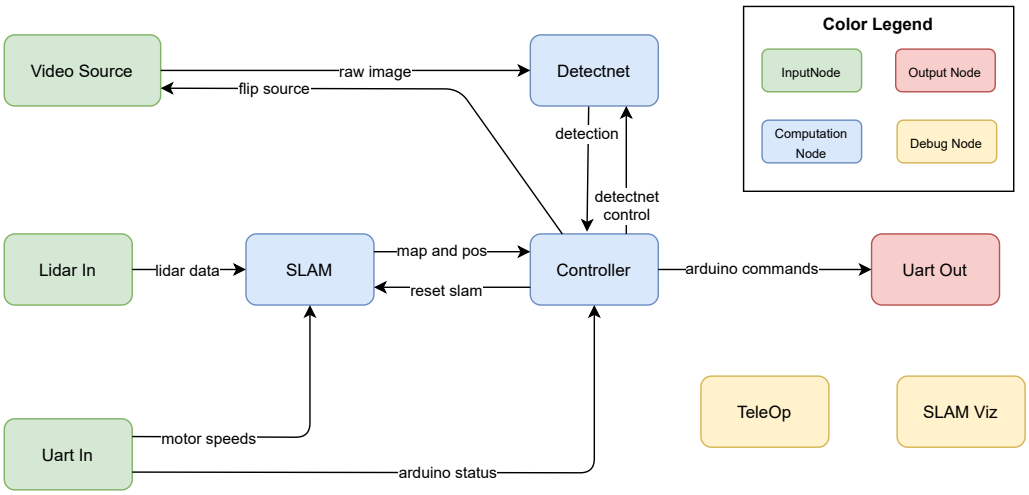


FIGURE 5.2
The simplified diagram of the ROS Nodes. Rectangles are ROS Nodes, as arrows are ROS Topics. The nodes in yellow were developed for debugging and therefore the ROS Topics were not drawn.

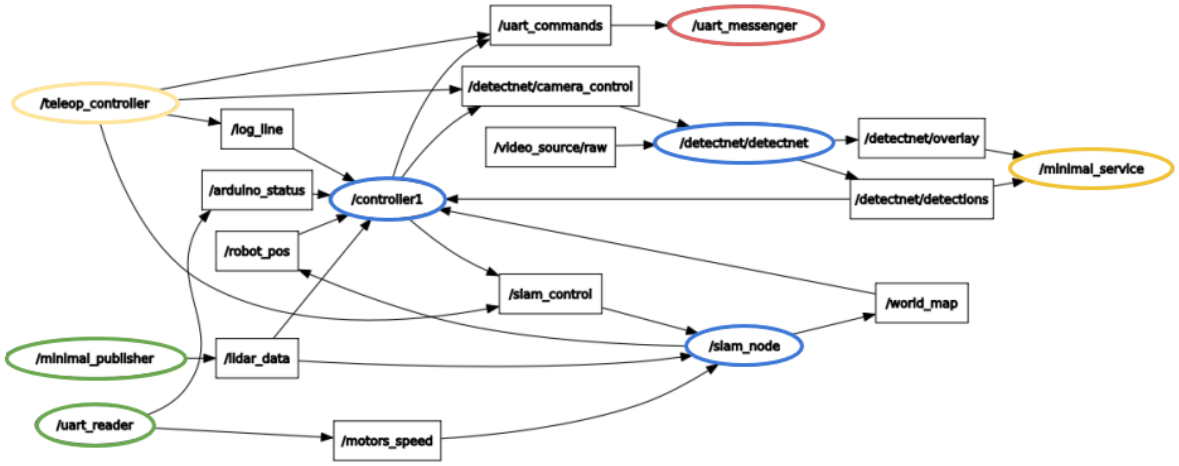


FIGURE 5.3
The actual diagram of the ROS Nodes. Rectangles are ROS Topics and ovals are Nodes. You can see the real topic names here.

The code of all those nodes can be found in the GitHub repository of the Team. Only 2 nodes were not entirely writen by us: the **Video Source** and the **Detectnet** nodes. Even though we have changed the content of those nodes, we found them in another ROS Project. This project is called Jetson Inference

[1] and was created by NVidia specifically to use the **computational power** of the Jetson Nano GPU for image based project. Indeed, image-based neural network are often very computationally expensive, however they are also well suited to be executed on a GPU. This task is often hard and this repository offers a framework to use **CUDA** (which is also made by NVidia) for 3 tasks: image recognition, object detection and segmentation. The project comes with an **extensive documentation**. One of the tool provided is called **detectnet** and performs the object localisation with many possible parameters. NVidia also provides the ROS2 Nodes to run those tools within any ROS project , the project is ROS Deep Learning and was very helpful.

We have slightly modified the code of the ROS Nodes to add 2 features: (i) we want to be able to flip the source image of 90 degrees (Section 5.2.4) and (ii) we want to be able to disable the detection of the Neural Network **without** killing the ROS process. Indeed, instantiating a new ROS Node takes time and we can't do it while the robot is running, but the Neural Network consumes a lot of power [2]. This is why we needed a programmatic way to turn it OFF. For those purposes, we created 2 topis (*detectnet_control* and *flip_source*)

The table below explains in more detail the content of the Topics that we used.

Topic	From	To	What
Raw Image	Video Source	Detectnet	The raw image from the CSI camera. The image can be flipped.
Lidar Data	Lidar In	SLAM	The measurements received from the LIDAR through UART. It contains distances and angles.
Motor Speeds	Uart In	SLAM	Motor Speeds read by the Arduino. Allowing SLAM to perform a 'faked' odometry.
Arduino Status	Uart In	Controller	Status sent by Arduino to inform the controller if a task Arduino was asked to do is finished, and how did it finish.
Reset SLAM	Controller	SLAM	A signal asking SLAM to either save its current state, or to reset itself to the previous saved state.
Detectnet Control	Controller	Detectnet	A signal to turn ON or OFF the detection. Running the detectnet is very power-intensive and Jetson performs better without it (when not required)
Flip Source	Controller	Video Source	A signal to perform a rotation of the video source.
Map & Pos	SLAM	Controller	Output of the SLAM algorithm.
Detections	Detectnet	Controller	Output of the Bottle Detection Neural Network.
Arduino Commands	Controller	Uart Out	Decisions made by the controller and sent to the Arduino as 'orders'.

TABLE 5.1
ROS Topics

Finally, the two Debug nodes in diagram 5.2 are not connected to others because it adds unnecessary information. However, we describe now their purposes.

1. **Teleop**: It is a Node to **remotely control the robot** via a controller. It can move the robot, ask for a specific action (initial rotation, pick bottles, move to bottles). It can also control the Neural Network,

[1] This is an Open Source Project with a MIT License
[2] A fun fact about the power consumption of the Neural Network: when it is activated, it consumes more power than the motors of the Robot...

the camera orientation, the SLAM reset state, and it can also take pictures. The development of this node was very helpful for the project.

2. **Slam Viz**: This node allows a visualization of the SLAM output (as in the YouTube Video)

5.2 MAIN COMPONENTS OF THE ALGORITHM

5.2.1 SLAM FOR ACCURATE LOCALIZATION AND MAPPING

SLAM, standing for **Simultaneous Localization And Mapping** is a very powerful **statistical method** used to gain knowledge of the environment and the position of the robot from Lidar measurements. There are several methods to solve this task, we used a **particle filter** (a Monte Carlo method) which proved to be rather simple and very efficient, using the code of the BreezySLAM algorithm that was implemented by Simon D. Levy, a professor of Computer Science at Washington and Lee University from a more famous algorithm named **TinySLAM**. TinySLAM, as explained on their website, is a Laser-SLAM which has been programmed in less than 200 lines of C Code [2]. However, the BreezySLAM algorithm contains much more than that, since it was really optimized by the author as much as possible. A great paper named **Particle filter in simultaneous localization and mapping (Slam) using differential drive mobile robot** [3] explains really well the theory behind the SLAM algorithm.

The **integration** of this code to ROS was fully realized by us and was a very long work. We had to spend of lot time to deploy the SLAM algorithm to the Jetson, because there are many parameters and understanding the effect of each parameter was very time-consuming. In order to explain which parameters were of importance for our implementation, let's first recall some of the most important point about the SLAM.

1. The **inputs** of the algorithms are (i) the Lidar measurements, that are for each batch of data an array of N tuples (angle of measurement, distance measured), and (ii) the motor speed measurements done by the arduino which can be integrated into a *faked* odometry measure.
2. The **outputs** of the algorithms are (i) an **occupancy grid**, i.e. a matrix that characterize where obstacles are located, and (ii) the estimated position of the robot inside this occupancy grid. TinySLAM is a **Single-Position** SLAM version, therefore the estimated position is just (x,y,theta).
3. One of the **major drawback** of the TinySLAM algorithm is that the output of the algorithm is highly dependent of the initial conditions.

Some of the important hyper parameters that we had to tune were the followings:

- The **resolution** of the **occupancy grid**, by tuning both the *number of points* per row, and the *real length* of a row. On the figure 5.4, the real length is 12 [m] and there are 500 points per rows of the matrix. Higher resolution means better results but comes at a computation price.
- The importance to give to new measurements for the **map update step**. SLAM attributes a **belief** to its current state. There is a trade-off between accumulation of knowledge and accumulation of errors.
- The **variances** of new positions, which translate into the belief to attribute to the current position.
- The ratio of importance to attribute between position updates from Lidar data or from the *fake* odometry.
- The hypothetical **obstacle length** to attribute when obstacles are detected.

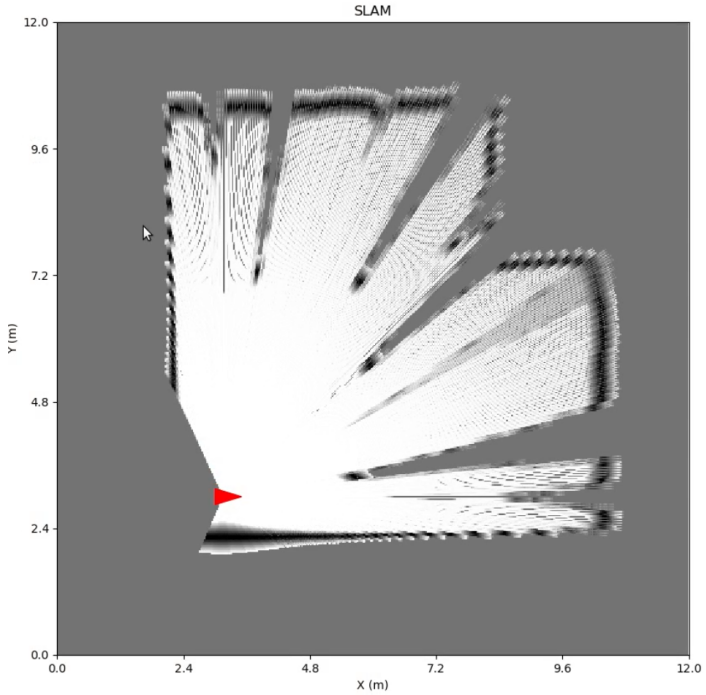


FIGURE 5.4
The SLAM output when Robottle was in the arena, at the recycling area before starting to move.

There are many other parameters, readers can find more information in our code accessible here. Another problem that we encountered was the **computation rate** of the SLAM, and of what is done **on top of** SLAM (path planning, path tracking, neural network). The only solution that we found was to **discard** half of the measurements done by the Lidar. The SLAM algorithm was still good enough to perform well while being provided only 50% of the data.

5.2.2 SLAM POST-PROCESSING: MAP ANALYSIS

The output of the SLAM is an occupancy grid, i.e. a squared matrix of 8 bits integers, where the value of each cell represents the **probability that there is an obstacle** at this point of the map. The raw map, as illustrated in Fig. 5.4 is hard to work with because of the non binary scale. In order to apply Path Planning and Path Tracking, one must first extract **meaningful features** from this matrix.

Before introducing the map analysis logic, 2 definitions must be given. **Zones** and **targets** are what we are looking to extract from the SLAM output.

- **zones**: the zones are the corners of the real arena.
- **targets**: the targets are the points where we decided to send the robot to, and can be computed as **weighted linear combinations** of the zones.

Treating the SLAM map as an image is very helpful as it allows to apply **Image Analysis** on the map. The Fig. 5.5 describes visually what is performed on the SLAM. Let us provide also a more detailed explanations of all the logical steps.

1. The first step is a **binary threshold** on the SLAM output (which is a gray-scale image). Indeed, working with a binary image is much easier.
2. Once the image is binary, we perform a **median blurring** with a circular kernel to remove small

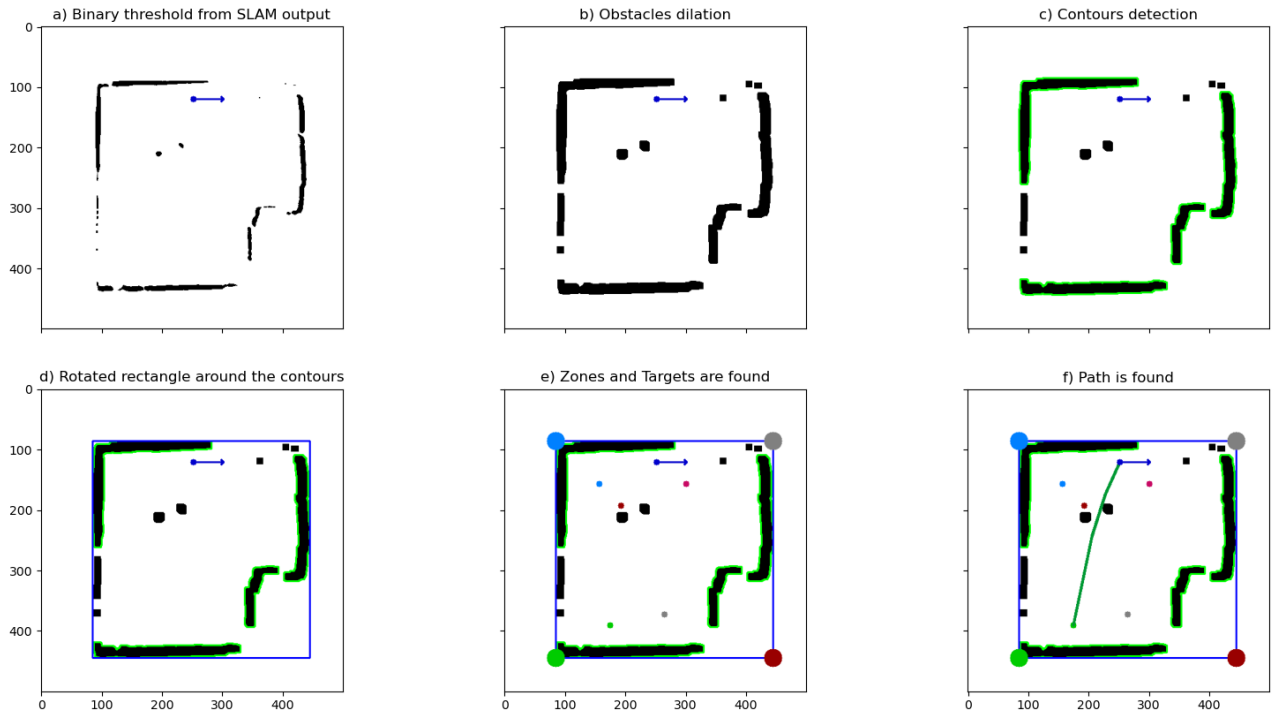


FIGURE 5.5
All the steps performed on the SLAM occupancy grid are precised in this plot.

points considered as outliers. Median blurring is very robust against noise following a 'salt and pepper' distribution.

3. A **dilation** of the binary obstacles with a squared kernel is performed to make the obstacles bigger. This step is performed to take into the actual size of the robot without changing the path planning. The algorithmic complexity of including the robot size in the path planning is actually N times bigger than doing this step.

4. The obstacles contours are found using **contour detection** from cv2. Contours are filtered to contain a minimum number of points (to remove contours around really small 'salt and pepper' noise).

5. A **rotated rectangle** that encapsulates all the contours previously found is computed by minimizing the area. This rectangle is called the **arena rectangle** and its corners represent the actual corners of the arena.

6. Here the logic depends on the question: is it the first time the algorithm runs (i.e., are the initial zones found ?)

    - If the initial zones were NOT found, the algorithm tries to validate the rotated rectangle that was found to verify if this rectangle is actually representing the arena by performing a threshold on the area of the rectangle. This unique check proved to be good enough. If it is a valid rectangle, the **zones** are attributed to the **rectangle corners**. If it is not a valid rectangle, the logic stops here and waits until a good rectangle is found. This is the case at the very

beginning when the robot is still constructing the SLAM map.

- If the initial zones were found and that the new rectangle is valid, the zones are updated. If the rectangle is not valid the logic stops here (controller enters recovery mode).

7. The **targets** are computed from the zones.
8. The path planner computes the best path (see more in the next section).

5.2.3 PATH PLANNING AND PATH TRACKING

Once the SLAM generates maps under the form of occupancy grid and that this map is processed to become analyzable, the next step is to **generate a path** and to **have the robot follow this path**. The logic to do so is the following.

- (*Path Planning*) Once in a while [3], update the **current path**. There is a **trade-off** between the **accuracy of the map** when the path is generated (in order to have a map which is very representative of the local neighborhood, one wants the path to be generated as often as possible) and the **stubbornness** of the robot: we want the robot to act smart, and therefore believing in the previous path is good. If the robot always updates its path, it will never have time to really follow it.
- (*Path Following*): At every instant, verify that the robot is following the proper path. If it is not, then correct the trajectory to adapt to the current path.

PATH PLANNING

The path planning algorithm that we choose is called **RRT***, standing for **Rapidly Exploring Random Tree ***. It is an algorithm designed to efficiently search a non-convex space (i.e. a space filled with obstacle, for instance) by randomly building a tree which goes filling this space. The tree expands as new nodes are randomly sampled from a given distribution.

The algorithm of the basic RRT algorithm is very famous in Robotics. Its variant RRT* was found in a paper published by MIT from the Laboratory for Information and Decisions Systems [1] [4]

Algorithm 3: RRT

$V \leftarrow \{x_{\text{init}}\}$; $E \leftarrow \emptyset$;
for $i = 1, \ldots, n$ **do**
  $x_{\text{rand}} \leftarrow \texttt{SampleFree}_i$;
  $x_{\text{nearest}} \leftarrow \texttt{Nearest}(G = (V, E), x_{\text{rand}})$;
  $x_{\text{new}} \leftarrow \texttt{Steer}(x_{\text{nearest}}, x_{\text{rand}})$;
  if $\texttt{ObtacleFree}(x_{\text{nearest}}, x_{\text{new}})$ **then**
    $V \leftarrow V \cup \{x_{\text{new}}\}$; $E \leftarrow E \cup \{(x_{\text{nearest}}, x_{\text{new}})\}$;
return $G = (V, E)$;

FIGURE 5.6
RRT algorithm as defined in the paper that we based our work upon

The RRT* variant is an optimized version which - for convergence reasons - complicates a lot the hand-writen algorithm (readers can find it under Algorithm 5 of the path planning paper [1]). In a few words, what it does is that each time a new point is inserted, it will try to create a path towards this point from many other points (and not only the closest point) and select the path with the shortest cost.

[3] this is defined by a hyper-parameter of the controller called *CONTROLLER_TIME_CONSTANT*
[4] This paper also gives a really strong and comprehensive mathematical description of several path planning methods.

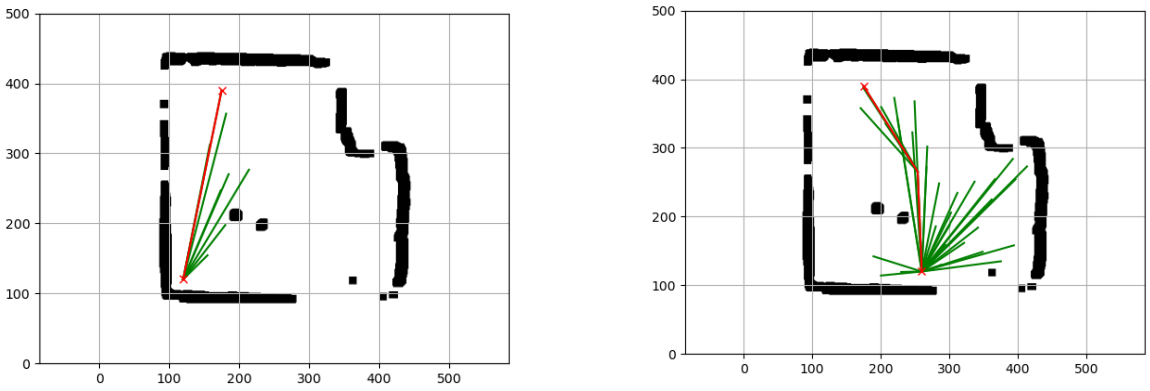


FIGURE 5.7
Two path generated for the same map at different initial position. We clearly see the growing tree that tries to randomly reach the target. On the right image, there is an obstacle between the robot and its target position.

PATH FOLLOWING

The path following algorithm is really easy. The reason why the path following is not difficult is because we know that new paths are generated rather often and therefore we don't have to attribute too much precision to following the current path. Indeed, if the robot deviates the path, when the new path is generated a few second later the robot will be exactly at the new path's origin. Also, the robot couldn't run into obstacles as there is always an **obstacle avoidance** logic which runs independently of the path following.

To follow a path, the robot continuously:

1. Look at the orientation difference between its orientation and the orientation of the path.
   (a) If bigger than a threshold (hyper-parameter), ask for a precise rotation to ajudst the orientation difference and return.
   (b) Else, go forward.
2. Look at the distance between robot current position and first point ahead of the robot in the current path
   (a) If smaller than a threshold (hyper-parameter), if this point is the target (target reached) then leave the travel mode. Else, remove the first point ahead of the robot in the current path from the path (path update) and return.
   (b) Else, go forward.

5.2.4 BOTTLE DETECTION WITH CUDA

Detecting the bottles is one of the fundamental requirements for this project. As previously explained, we decided to use a **neural network** that is ran on the GPU of the Jetson Nano to detect the bottles. Such a process is very power-intensive, however it is possible to reach **5 FPS** for the detections.

NEURAL NETWORK

Training a neural network able to do object recognition is a very difficult task. It would mean collecting thousands of labelized pictures of bottles and running the classifier. Instead we wanted to see if it was not possible to find networks already trained and performing well enough for this task. Even tough it was a long task (*before finding the final network, we tried many different networks*), we found provided by NVidea on an open-source repository a network trained to detect only bottles. It was created in 2017.

The network that we use is a Deep Neural Network that was trained on the COCO dataset and contains millions of weights. It is a really deep network of 172 layers and millions of parameters. The weights of the Network are compressible into 30 MB.

The network is ran using NVIDIA Tensor-RT, and for high-performance deep learning inference includes a deep learning inference optimizer. Optimized network are cached to disk and once a network is optimized it runs a lot faster.

About the **quality of the detection**, the network is very accurate and has the following properties of classification. Its false positive rate is very low: false positives (i.e. detected bottles that are not real bottles) are extremely rare. The false negatives rate (i.e. when no bottles are detected but there is actually one) is rather low. Almost all bottles present in the frame are detected, but not all.

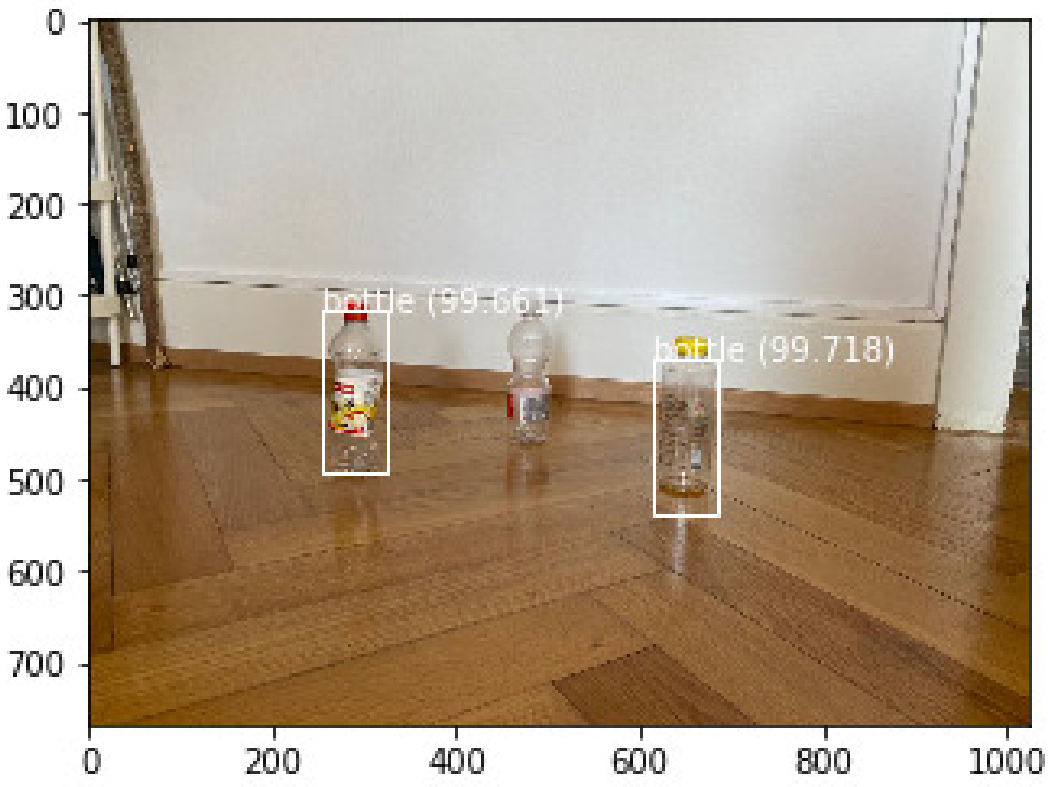


FIGURE 5.8
Detection results illustrating 2 true positives and 1 false negative

DETECTING LAYING BOTTLES

While working on the bottle detections we realized a very interesting feature. The bottle detection was very accurate on standing bottles, however it was not so good on laying bottles. Often the robot would

not detect bottles that were flipped on the ground. This observation is perfectly relevant in the context of neural network. If the network was trained using images were bottles are standing (which is a very natural hypothesis) then it is normal that the network performs better at detecting those bottles.

To solve this problem, our idea was to manually 'flip' the image of 90 degrees and give this image as the input to the neural network. The results proved to be much better. This is demonstrated in the Fig. 5.9. The detection is much better for the flipped image than for the normal image. Another property of the neural network that we use is that it is *180-degree rotation invariant* (frankly, this is really by chance for us). This means that we only need to do one flip of 90 degrees and that bottles laying on the left or on the right will be detected properly.

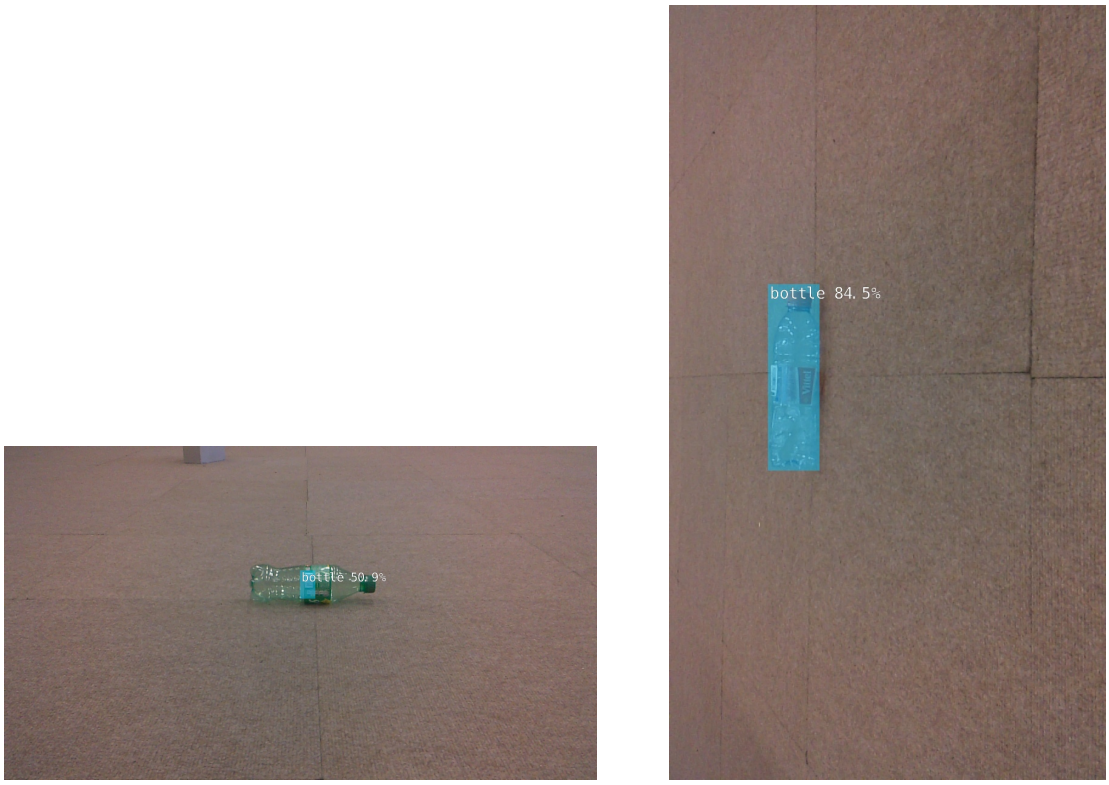


FIGURE 5.9
Results of the neural network detections for the same situation with the image flipped and not flipped. We note that for a bottle that is laying on the ground (not a standing bottle) the result over the flipped image is much better.

EVALUATING THE BOTTLES' POSITION

Simply detecting bottles in the image frame is not enough. Indeed, the robot must travel to those bottles. For this task, one must evaluate for each bottle two parameters: its **angle** with respect to the robot's orientation and the **distance** to the robot. Only then can the robot move to get the bottle.

While working with the camera, we noticed the two following empirical rules: (**i**) as the camera angle range is rather limited, the angle estimation error from the bounding box is rather small, and (**ii**) since the bounding box is not always well located (*as in the left image of Fig. 5.9*) the distance estimation using the bounding box is very poor.

We decided to go for the following method.

- The **angle** is detected using the bounding box and a regressive function that will be introduced just

now.

- As for the **distance**, the task is harder. It is not possible to use the bounding box because of high variance. It is not possible to use the LIDAR because it only detects standing bottles and not laying bottles. That is why we use 3 **ultrasonic** sensors placed at the front of the robot.

The regressive function that estimates the angles takes as input the bounding box (x_center, y_center, w, h) and returns the estimated angle. This function was trained using a very simple dataset, consisting of a list of tuples (bounding box, angle). To create this dataset, we followed an easy procedure. We placed one bottle somewhere in front of the robot, while measuring the exact angle at which this bottle was placed. We also ask the robot to save the detection's bounding box for this bottle, and we repeated this process for as many positions as we could [5]. The polynomial fit that we applied is a third order polynomial of the form

$$p_0x^3 + p_1x^2y + p_2xy^2 + p_3y^3 + p_4x^2 + p_5xy + p_6y^2 + p_7x + p_8y + p_9$$

where x and y are the position of the bottom center of the bounding box in pixels. For flipped images, we first simply transform the coordinates of the bounding box with an inverse rotation of 90°.

FIGURE 5.10
The angle function interpolates, using a cubic regressive function, the angle at which is located a bottle using the center of the bottom of the detected bottle's bounding box. Image on the left shows the horizontal axis where most of the variance happens. However we can observe that the vertical axis also influences the estimation. The model can adapt to the **camera distortion** at the edges of the images.

[5] The full training dataset includes not more than 30 recordings.

5.3 STATE MACHINE FOR CONTROLLING THE ROBOT

For the control of the robot we use a state machine:

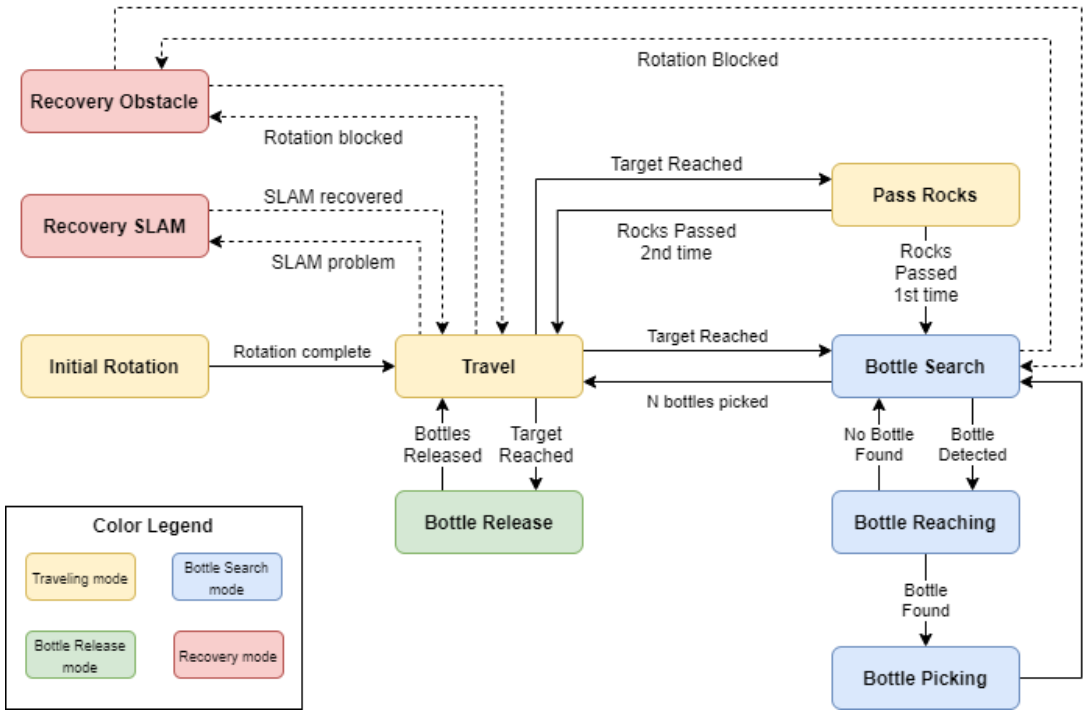


FIGURE 5.11
The state machine diagram of the high-level software

5.3.1 INITIAL ROTATION

This is the initial state of the program. The robot moves a little bit towards the center and does a rotation of 360°, and the purpose is to create a map of the arena. In this state, the Jetson simply sends a command to the Arduino and waits for its response. Once it receives the message, it changes to **Travel mode**.

5.3.2 TRAVEL

Travel mode is used to go to different predefined target locations inside the arena. It does so by having one of the parameters in the code, which corresponds to the list of all the targets the robot will have to go to, and every time this state is entered, the target is switched to the next in the list. This approach makes it very easy to change the behavior of our robot, since we can quickly modify to which area we want to send it. During this phase, the program regularly calculates a path to follow using RRT* and tracks it (Section 5.2.3), and once it reaches the destination, it switches state, depending on the current target:

- If it reaches the recycling area, it switches to **Bottle Release** (Section 5.3.6)
- If it reaches the rocks, it switches to **Pass Rocks** (Section 5.3.7)
- If it reaches another zone, it switches to **Bottle Search** (Section 5.3.3)

After testing, we found the best strategy to be to collect bottles around zone 1 and 2, by doing two round trips, and at the end go to the rocks area (Fig. 2.1). Given that crossing the stones is tricky and can cause problems, instead of going directly there in **Travel mode**, the target is set right before the barrier, and

then it switches to **Pass Rocks**, giving the control to the Arduino which outputs a sequence of commands that allows the robot to pass the obstacle. The same strategy is done when crossing the rocks in the other direction.
The path generated avoids the obstacles, but as a safety measure, while the robot is traveling, obstacle avoidance is constantly performed. Indeed, when an obstacle is detected, a rotation is performed until the path in front is free. This is done by analyzing directly the data received from the Lidar, and checking if there is an obstacle in a rectangle of a given length in front of the robot. This method also allows for the differentiation between bottles and obstacles/walls, since the size of the objects is different.

5.3.3 BOTTLE SEARCH

Once the robot reaches an area where bottles have to be collected, **Bottle Search** is activated. The objective here is to find the bottles, and it does so by continuously alternating between trying to detect a bottle with the camera and rotating 40° to the right. Once a detection happens, the robot aligns itself with the bottle, and then goes into **Bottle Reaching** mode.
This process continues until either a certain number of bottles has been collected or a given amount of 40° rotations have been completed. The second condition is there to avoid falling in a local minimum where there are no more bottles around and the robot keeps rotating indefinitely. After that it goes back to **Travel mode**.

5.3.4 BOTTLE REACHING

The purpose of this state is to get to a position where the bottle is in arm's reach. The Jetson simply sends a command to the Arduino, which is in charge of the control of the robot, and waits for the response. If it return a positive answer, it means that the bottle is now right in front, and therefore it switches to **Bottle Picking**, while if the response is negative, the Arduino couldn't find the bottle, and the state goes back to **Bottle Search**.
Similarly to **Travel mode**, while the robot is advancing, if the Lidar detects an obstacle, it performs obstacle avoidance. This is especially important when a bottle is behind an obstacle but still in the visible range of the camera, in which case the robot could crash against it before reaching the bottle.

5.3.5 BOTTLE PICKING

When entering this state, the Arduino detected something right in front of the robot, but the problem is, it could either be a bottle or a rock, because bottles on the other side of the barrier can also be detected by the camera. The first thing to do is therefore to check which of the two cases it really is, and it's done using the SLAM output. Knowing the position and orientation of the robot, it's possible to check if it is in front of a stone and if that is not the case, the Jetson tells the Arduino to pick the bottle up. Finally, again the response from the low-level can be either positive or negative, and in the first case, a counter to keep track of how many bottles have been collected is incremented. In both instances though, the state is then switched back to **Bottle Search** to look for other bottles if there is still space.

5.3.6 BOTTLE RELEASE

Once enough bottles have been collected and the robot is back to the recycling area, the first thing it does is align himself with the diagonal between the elevated and the recycling zone, facing the latter. Then it starts to advance slowly until it detects the walls, at which point it stops and sends the order to the Arduino, which handles the release of the bottles. After this operation, the state will always go back to **Travel mode**, which will send Robottle to the next destination.

5.3.7 PASS ROCKS

This state is used to cross the stones. Since going over the rocks offsets the Lidar plane, the data received will be completely false and therefore the SLAM will fail. To avoid this, the current state of the map is saved, and after telling the Arduino to cross the rocks and receiving the confirmation that the operation was complete, it sets the SLAM map back to the correct one. By doing this, the algorithm is able to localize itself again and not get lost.
Since this has to be done two times: one to reach inside the zone and one to get out, in the first case the state is switched to **Bottle Search** to collect bottles inside the area, and in the second, it goes back to **Travel mode**, to go back to the recycling area for the last time.

5.3.8 RECOVERY OBSTACLE

From our testing, we found two situation in which the robot was getting stuck and couldn't continue its job: when it was trying to rotate against an obstacle and when it ran over something and the SLAM failed as a consequence. This can only happen either in **Travel mode** or in **Bottle Search**, since they are the only states where a rotation can cause the robot to get stuck. There are other cases where the robot is rotating (i.e. **Initial Rotation** or **Bottle Reaching**), but they are either small rotation or there are no obstacles around. To address this problem, two recovery states were introduced: **Recovery obstacle** and **Recovery SLAM**. The first one occurs when a rotation should happen but the orientation of the robot doesn't change and we detect this scenario thanks to the SLAM. The solution to getting out of that situation is simply to move forward a bit, and from our testing, with this simple rule the robot never got stuck against an obstacle.

5.3.9 RECOVERY SLAM

The second recovery state occurs less often, but could still pose big problems. We noticed it happening when the robot ran over a bottle or the rocks by mistake, and it caused the Lidar to "see" the ground or the walls of the room. To counter this, every time a new map is received, we check if it is valid, i.e. if we can detect the four zones and they are reasonable, and if that is the case, we save the current map. If instead the zones are not valid, the last saved state of the SLAM is restored. In fact, this is the same method used when crossing the rocks (Section 5.3.7) and it works decently well. It's still possible to fail when going over a big obstacle such as going to zone 3, because for it to work, SLAM has to find back the position of the robot after stabilizing again, and it wasn't always the case. During the testing though, this solution worked 100% of the time with small obstacles and failed only about 20% when going over the rocks.

5.4 CODE ON THE ARDUINO MEGA

The Arduino Mega has the job of controlling directly the motors, the servos and the ultrasonic sensors. It receives commands from the Jetson and executes them, returning a message each time. To do this another simpler state machine is used:

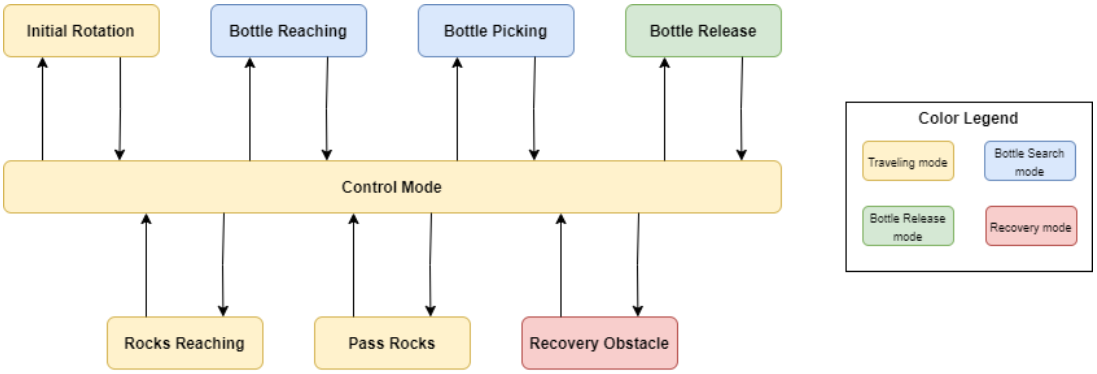


FIGURE 5.12
The state machine diagram of the low-level software

5.4.1 CONTROL MODE

This is the main mode. Here the Jetson can control the Arduino to either move forward/backwards and turn left/right, or to execute a command, and in that case, it will change state to the corresponding one.

5.4.2 INITIAL ROTATION

This is called at the beginning, and it simply sends the commands to turn 45°, advance forward a little bit and do a 360°rotation.

5.4.3 BOTTLE REACHING

When the Arduino enters this state, it means that a bottle should be in front of the robot. Therefore, it starts advancing, until something is detected with the ultrasonic sensor at the front. In addition to that, since it's hard to calculate precisely the angle of a bottle using the camera, the robot might not be aligned perfectly with it, and therefore it does a last adjustment of the orientation using the distances measured by the ultrasonic sensors.
As a safety measure, a maximum distance to travel is also introduced, in case the robot misses the bottle completely, and a negative response will be sent to the Jetson.

5.4.4 BOTTLE PICKING

Since the sticky arm has a hard time picking up standing bottles, the robot advances a few centimeters to tip the bottle off and then retreats the same amount or until it detects the bottle with the ultrasonic sensors. This method works surprisingly well, and the bottle rolled away only in about 1 in 10 times, and even then, the camera can often detect the bottle again and end up picking it up anyway. Finally the servo controlling the arm is activated and it tries to pick up the bottle. If after that the sensors still detect something, it tries a second time.

5.4.5 BOTTLE RELEASE

In this state, the robot is facing the corner of the recycling area and it simply performs a 180° rotation, followed by the opening of the back door, a little shake in the event a bottle gets stuck and the closing of the door.

5.4.6 ROCKS REACHING

When the robot wants to cross the rocks, it first has to align itself precisely with them. This process is very similar to the **Bottle Reaching**, in that it goes forward until the sensors at the front detect something close. From our testing, we noticed that passing the rocks was far more successful when approaching them backwards and at maximum speed than any other method, therefore the second step is to turn around to face away from them. Before continuing the manoeuver, the Arduino sends a message to the Jetson in order to let it save the current map of the SLAM.

5.4.7 PASS ROCKS

This is the direct continuation of **Rocks Reaching**. The robot is sent backwards at full speed for 5 seconds, then a message is sent to the Jetson telling it that it can activate the map update of the SLAM again, and a rotation of 90°is done to align itself with the line of rocks.

5.4.8 RECOVERY OBSTACLE

When the Jetson enters into **Recovery Obstacle** mode, it tells the Arduino to move forward about 30 centimeters in order to get away from the obstacle that's blocking the rotation.

5.5 COMMUNICATION BETWEEN JETSON AND ARDUINO

The communication is done via a protocol of messages that are sent between the Jetson and Arduino. The high-level will send a command to the low-level to be executed, and waits for its response. Here is the list of commands:

Commands	Mode	Response
'w','a','s','d','x'	Locomotion	None
'm1','m2'	Change speed	None
'r'	Initial Rotation	's1': rotation complete
'y'	Bottle Reaching	's0': maximum distance reached, no bottles detected 's1': bottle detected
'p'	Bottle Picking	's0': bottle not picked, still in front of the robot 's1': bottle picked
'q'	Bottle Release	's1': bottles released
'Y'	Rocks Reaching	's1': rocks reached
'c'	Pass Rocks	's1': rocks passed
'R'	Recovery Obstacle	's1': robot advanced, obstacle should be clear

TABLE 5.2
List of commands between Jetson and Arduino

5.6 ROBOTTLE PYTHON PACKAGE

Along with the ROS implementation and the above mentioned algorithms, we built a python library containing a lot of useful functions used in the main code. The next sections provide a list of all the Utils packages and their functions.

MAP UTILS

All of the map analysis functions. They are used to filter and perform computer vision on the occupancy grid map of the SLAM.

Function name	Description
pos_to_gridpos	Given a position in meters, it returns the position in pixels according to the map size and resolution.
get_map	Transforms the map array received, a 1xN array, into a numpy array of the correct size (N_xxN_y).
plot_map	Plots the occupancy grid map.
filter_map	Given directly the occupancy grid map from the SLAM, it filters it, creating a new map used later for zone and target analysis. It performs a binary threshold, a median filter and a dilation of the obstacles. More details in Section 5.2.2, steps 1-3.
get_bounding_rect	Given a filtered binary obstacle map, it finds the rectangle that surrounds the map. It first finds the obstacles to keep as contours and computes the smallest rectangle surrounding them. More details in Section 5.2.2, steps 4-5.
get_initial_zones	From the corners and the robot position, it returns the outermost position of the zones. The closest zone to the robot is set as a parameter, and all of the other zones are found according to it (Section 5.2.2, step 6).
get_zones_from_previous	Given the previous known zones and the new corners of the rectangle, it finds the new zones by checking the closest zone to each new corner (Section 5.2.2, step 6).
are_new_zones_valid	Returns True if the new zones are valid. This check is a simple threshold on the distances the zones have moved from the previous to the current position. It assumes that the SLAM map is stable from one iteration to the next and therefore, if the zones move a lot, it means that something went wrong (Section 5.2.2, step 6).
get_targets_from_zones	Computes the target points where the robot should go based on a weighted linear combination of the zones. Each target corresponds to two weights: one to determine the position on the vertical axis, and one on the horizontal axis (Section 5.2.2, step 7).
get_random_area	Given the four zones, returns the rectangle area where random sampling has to be performed for the path planning algorithm.
make_nice_plot	Plot the data given as parameters. It can plot the binary grid, the robot position, the obstacle contours, the rectangle surrounding the map, the four zones, the target points and/or the path generated.

VISION UTILS

The set of functions used to handle taking and saving pictures and to analyze the output of the Neural Network.

Function name	Description
take_picture	Takes a picture with the camera and returns it.
save_picture	Saves a picture in a given folder and with a given name.
get_angle_of_detection	Given the bounding box of a detection, computes the angle of the bottle. This function was created by fitting a dataset of collected samples to a polynomial. More details in Section 5.2.4.
get_best_detections	Given an array of bounding boxes of detections (either normal or rotated 90°), it finds the closest bottle among them.

CONTROLLER UTILS

Functions called by the controller and used mainly for path tracking, with the exception of *is_obstalce_a_rock*, a function that uses the SLAM map to check if the robot is in front of a bottle or a rock.

Function name	Description
get_distance	Gives the euclidean distance between two points.
get_rotation_time	Returns the rotation time in seconds given an angle of rotation and the rotation speed.
angle_diff	Computes the angle difference between two angles adapted to the map frame of reference.
get_path_orientation	Given two points along a path, it returns the angle orientation.
is_obstacle_a_rock	Returns True if the robot is in front of the rocks given its position and the zones' points. It is called after reaching a bottle, to check if it is indeed one or if instead the robot is in front of a rock. In order to do this, since the two line of rocks are known, it checks on which side of each line the robot is, and in conjunction with the distance from each line, it is able to predict with accuracy if the point in front of the robot is a rock or not.

LIDAR UTILS

A set of functions that use directly the Lidar data.

Function name	Description
check_obstacle_ahead	Given the data from the Lidar (angles, distances), it checks a rectangle in front of the robot of a certain width and length and checks how many measurements there are inside that rectangle. Depending on a low and a high threshold, it returns True if the number of measurements is inside those and False otherwise. By changing the thresholds, it is possible to detect precisely a wall, an obstacle or a standing bottle.
get_valid_lidar_range	This is a debugging function. Since the Lidar is placed in front of the robot, it sees a fixed wall behind him. In order for the SLAM to work correctly, this wall has to be removed from the data sent by the Lidar and this function allows to find the two limit angles at which the robot wall starts and ends.

UART UTILS

A function called to transform the speed message received by UART from the Arduino.

Function name	Description
get_speed	From the array of characters read by UART, returns the speed of the wheels as a number.

CHAPTER 6

RESULTS

6.1 EARLY TESTING

The very early stages of the semester were used to figure out what our strategy would be (Section 2). After a lot of brainstorming, We built very rudimentary mechanisms, such as the first prototype (Section 3.1) which helped us to define our goals and solutions early on.

6.2 LOCOMOTION

For the locomotion, we quickly established that we wanted either a 4-wheel drive or a 2-wheel drive with 2 caster wheels but up until Robottle 3.0 (Section 3.3) this was still an open question. Initially with the second prototype (Section 3.2) we used the second option for simplicity of construction, but with the thought of switching to the first option for the later stages of development. Once the next prototype was built though, we realized that having a 4-wheel drive wasn't such a good idea. By decreasing the friction of the back wheels, it was possible to turn, but not smoothly and with some drifting. We decided therefore to change back to the other idea.

6.3 LOCALIZATION

For the localization the plan has always been to implement SLAM and if that was not enough, to use the four colored beacons to help with the position update. In fact, a lot of time in the first months has been invested in making SLAM work, and when it finally did, we realized that even without odometry, the SLAM could accurately estimate the position and orientation of the robot.

6.4 OBSTACLE AVOIDANCE

Obstacle avoidance was actually implemented very late in the project. This is because the path planning algorithm (Section 5.2.3) was already taking into account the obstacles and therefore in theory, there was no need to add it. Practice though, as we know, is very different from theory, and the path tracking didn't work perfectly. This meant that rarely, the robot would crash into an obstacle because of some imprecision either in the path tracking or in the position estimation itself. Once implemented though, it proved to be very useful, since we could use it in many other situations, for example the same code was used to differentiate between obstacles and standing bottles.

6.5 BOTTLE PICKING

As explained in previous sections, the chosen method for picking up bottles is an arm with sticky tape at its tip (Section 2.6). As soon as the arm was built on the third prototype (Section 3.3) we started testing how it behaved just by moving it manually by hand. We soon noticed that it actually performed well, and the glue was even too strong at times. A mistake we made though, was to test with the servo very briefly, since after seeing the manual results we thought that the arm would work perfectly attached to a motor as well. This meant that until the last week before the competition, we hadn't tested the arm again and relied on the fact that it would work as expected. The reality though, was that when operated remotely, the bottles would often not stick to the arm because the tape was full of dust, stick too much because instead it was too new or sometimes they would fall to the side because the robot wasn't perfectly aligned with the bottle. In the end, with some last days adjustments, we managed to make it a lot better and brought the success rate of picking up bottles to about 75%. The modification done were mainly three: aligning the robot better in front of the bottle by going from one to three ultrasonic sensors at the front, making the passive release mechanism more rigid and higher so that bottles were more likely to fall inside and adjusting the strength and speed of the servo as best as we could. In the end we made the bottle picking work, but with some more testing we could have avoided some last minute stress.

6.6 BOTTLE RELEASE

The bottle release didn't need much testing. It was a very simple mechanism, implemented many times before us in previous competitions, therefore we were confident it would work. Indeed, when we actually tried to open the door, the bottles all fell out without problems. By repeating the process though, we noticed that sometimes a bottle in a particular position would get stuck inside and not fall. So we simply made the slope slightly bigger and made the robot shake a little bit when releasing the bottles and this easy fix made it work 100% of the time.

6.7 PASSING ROCKS

We had in mind since the beginning that we either wanted to go to zone 3 or 4 to collect a good amount of points. It wasn't until the arena was setup that we made the final decision though. We immediately saw that both approaches completely ruined the SLAM, since the Lidar plane is not horizontal while both going up the ramp and passing the rocks. We decided therefore to first solve all the other problems, and at the end, try to implement a robust enough code, such that it could go into one of those zones as reliably as possible. In retrospect, if we had thought more about these problems, we could have figured out a solution earlier, but in the end it wasn't an easy task and it's hard to predict all of the possible issues before being able to test in the actual arena.
During the last week, after we decided that it would be easier to go into zone 3, rather than 4, we tested a lot of different approaches until we found one that worked. The best method we found was to advance backwards at maximum speed to cross the rocks, and restore the SLAM after. The success rate of the complete operation (crossing in both directions and recovering the SLAM in both occasions) wasn't very high, with the robot returning safely to the recycling area only about 40% of the times. This is why we decided to only do it at the end of each run, after securing easier bottles.

6.8 TESTING IN THE ARENA

As soon as the arena was mounted, we started testing inside it. This was definitely a very smart decision, since a few of problems that we hadn't thought about appeared immediately. Luckily our robot wasn't

affected much by the room conditions such as lighting or ground roughness, therefore, other than the above mentioned bottle picking problem (Section 6.5), the only issues we had that couldn't be resolved quickly were in the code. This meant that during the very busy times right before the holidays, we didn't need to use the bigger machines such as laser cutters and 3D printers, which made us gain a lot of time overall.
As the competition approached, the robot became better and better and a few days before it, Robottle was able to complete a run without problems and collect bottles in areas 1 and 2. This allowed us to focus on making the code more robust during the last two to three days. We improved many elements, such as the angle detection of the bottles, the recovery modes, the reliability of the arm and once we felt satisfied with the current state of the code, we tried to make the robot go to zone 3 (Section 6.7).
In conclusion during all the testing, what happened was that up until the last month, we built the more general part of the code that we knew would be useful no matter what, as best as we could to face the given problem, and once the real environment was presented to us, we started fitting the implemented solutions to the problem. We invested a lot of time, especially in the arena, and we are very satisfied with the final state of the robot. At the end there were still some rare scenarios that could make the robot crash, but it would have been almost impossible to increase the robustness to avoid all of them, and in its final state Robottle is able to collect on average between 80 and 100 points from zone 1 and 2, and with the occasional one or two bottles from zone 3, there is potential for over 150 points.

6.9 COMPETITION

The final competition unfortunately didn't got as planned. After the first round of collection of bottles, while getting back to the recycling area, something went wrong, and Robottle crashed in the wall of the arena. We tried resetting it to start over, but because of the ROS setup, it took too long and in the end didn't have time to collect anything else. We are still not sure now what the cause of the problem is, but we have a strong suspicion it was a crash in the SLAM algorithm. That specific issue had appeared in only one of the many test runs we did, near the beginning of the last week, and ever since we couldn't replicate it and had no way of resolving it.
After the official 10 minute run of the competition we tried launching Robottle again, and this time one of the rare scenarios that could potentially cause problems happened. The robot found itself close the angle of the arena and went right by a bottle near the edge of the arena, only to find itself close to a wall, facing towards the angle. When it tried to turn against the wall, it activated the Recovery Obstacle mode as expected, and advanced forward, but again only to get very close to the other wall right in front of him. The same situation reproduced itself and as it couldn't turn properly, it activated the same recovery again and actually got out of that situation, but unfortunately the fact that it spent so much consecutive time with the Lidar facing an obstacle, caused some problems to the SLAM map update. After that, Robottle couldn't localize itself anymore and we had to stop the run there.
Looking back, there wasn't really much we could have done to improve our result. We tested a lot, we did in total more than 15 full runs in the arena starting and ending at the recycling area, without counting all the other smaller tests and unfortunately one of the problem was impossible for us to resolve and the other was so rare that we never observed it in practice before. With one extra week in the arena, we could have maybe done more to debug and resolve all of the rare scenarios, but we already spent a lot of time trying to do that and it's safe to say that we did what we could and although luck wasn't on our side, we are very happy with what Robottle was able to achieve at the end.

Chapter 7

Project Management

7.1 Budget

Amount	Description
2000.00	Initial budget
0.00	Breadboard [#5]
-299.50	Tetrix kit [#1]
-7.50	WildTumper 120x60 mm wheel, 4 mm shaft [#13]
-34.95	75:1 DC motor with encoder [#4]
-34.95	75:1 DC motor with encoder [#2]
-7.50	WildTumper 120x60 mm wheel, 4 mm shaft [#14]
-7.50	WildTumper 120x60 mm wheel, 4 mm shaft [#5]
-19.95	NiMH rechargeable battery pack (7.2 V, 3000 mAh) [#2B]
0.00	Arduone connector/power board [#5]
-15.00	Dynamixel AX-12A motor [ID 2]
-7.50	WildTumper 120x60 mm wheel, 4 mm shaft [#11]
-44.50	Arduino Mega2560 R3 [#15]
-7.50	WildTumper 120x60 mm wheel, 4 mm shaft [#12]
-7.50	WildTumper 120x60 mm wheel, 4 mm shaft [#6]
-45.00	DFRobot DRI0018 2x15A motor driver [#9]
-34.95	75:1 DC motor with encoder [#1]
-15.00	Dynamixel AX-12A motor [ID 11]
-34.95	75:1 DC motor with encoder [#6]
0.00	Battery connector/fuse board [#3]
7.50	Return: WildTumper 120x60 mm wheel, 4 mm shaft [#13]
7.50	Return: WildTumper 120x60 mm wheel, 4 mm shaft [#5]
7.50	Return: WildTumper 120x60 mm wheel, 4 mm shaft [#6]
-7.50	WildTumper 120x60 mm wheel, 4 mm shaft [#4]
-7.50	WildTumper 120x60 mm wheel, 4 mm shaft [#10]
-7.50	WildTumper 120x60 mm wheel, 4 mm shaft [#7]
-31.55	Raspberry Pi Camera v2.1 [#10]
-12.25	MicroSD card 32 GB UHS-I [#2]
-58.25	Raspberry Pi 4 (4 GB) [#5]
-10.00	Raspberry Pi USB-C power supply [#2]
-3.00	Micro-HDMI→HDMI adapter [#1]
0.00	HDMI-VGA converter [#4]
-40.00	LiPo Battery 3S (11.1 V) 4000 mAh [#3]
-25.00	LiPo protection & power board [#1]
-68.20	Maxon ESCON 24/2 controller [#30]
-12.00	ESCON 24/2 motherboard [#3]
-108.90	Maxon EC 32 flat (15 W), 1:60 gearbox [#5]

Amount	Description
-108.90	Maxon EC 32 flat (15 W), 1:60 gearbox [#4]
-68.20	Maxon ESCON 24/2 controller [#33]
-12.00	ESCON 24/2 motherboard [#4]
34.95	Return: 75:1 DC motor with encoder [#4]
34.95	Return: 75:1 DC motor with encoder [#2]
34.95	Return: 75:1 DC motor with encoder [#1]
34.95	Return: 75:1 DC motor with encoder [#6]
-5.00	Maxon EC-flat to MM8 adapter [#3]
-5.00	Maxon EC-flat to MM8 adapter [#4]
-7.50	WildTumper 120x60 mm wheel, 4 mm shaft [#13]
-7.50	WildTumper 120x60 mm wheel, 4 mm shaft [#8]
-7.50	WildTumper 120x60 mm wheel, 4 mm shaft [#5]
-7.50	WildTumper 120x60 mm wheel, 4 mm shaft [#6]
7.50	Return: WildTumper 120x60 mm wheel, 4 mm shaft [#4]
7.50	Return: WildTumper 120x60 mm wheel, 4 mm shaft [#14]
7.50	Return: WildTumper 120x60 mm wheel, 4 mm shaft [#10]
7.50	Return: WildTumper 120x60 mm wheel, 4 mm shaft [#11]
7.50	Return: WildTumper 120x60 mm wheel, 4 mm shaft [#12]
7.50	Return: WildTumper 120x60 mm wheel, 4 mm shaft [#7]
-108.90	Maxon EC 32 flat (15 W), 1:60 gearbox [#17]
-68.20	Maxon ESCON 24/2 controller [#14]
-12.00	ESCON 24/2 motherboard [#8]
-68.20	Maxon ESCON 24/2 controller [#17]
-12.00	ESCON 24/2 motherboard [#7]
-108.90	Maxon EC 32 flat (15 W), 1:60 gearbox [#18]
-5.00	Maxon EC-flat to MM8 adapter [#9]
-5.00	Maxon EC-flat to MM8 adapter [#32]
-40.00	LiPo Battery 3S (11.1 V) 4000 mAh [#25]
-25.00	LiPo protection & power board [#15]
68.20	Return: Maxon ESCON 24/2 controller [#33]
-68.20	Maxon ESCON 24/2 controller [#16]
-31.55	Raspberry Pi Camera [#10]
-55.20	Ultrasound sensor 3-pin (IC1MVO5 NO8MM4X 2L5SDYJ)
31.55	Return: Raspberry Pi Camera [#10]
31.55	Return: Raspberry Pi Camera v2.1 [#10]
-25.00	Raspberry Pi Camera v2.1 with IR filter [#1]

Current budget balance: 460.95 CHF.

Figure 7.1
Virtual budget

The final balance of the virtual budget is 460.95 CHF, but many of the elements bought were only used for testing and were only returned after the competition. What was actually on the robot in its final version is highlighted in yellow, and it amounts to 700.15 CHF which brings the balance up to 1'299.85 CHF.

Amount	Description
750.00	Initial budget
-98.62	Commande Mouser du 15.9.2020
-125.51	Commande Mouser du 27.9.2020

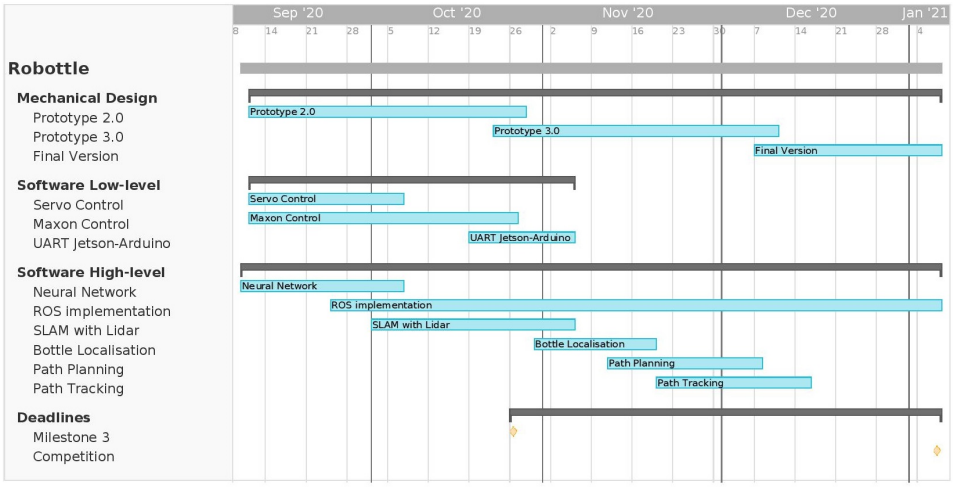
Current budget balance: 525.87 CHF.

FIGURE 7.2
Real budget

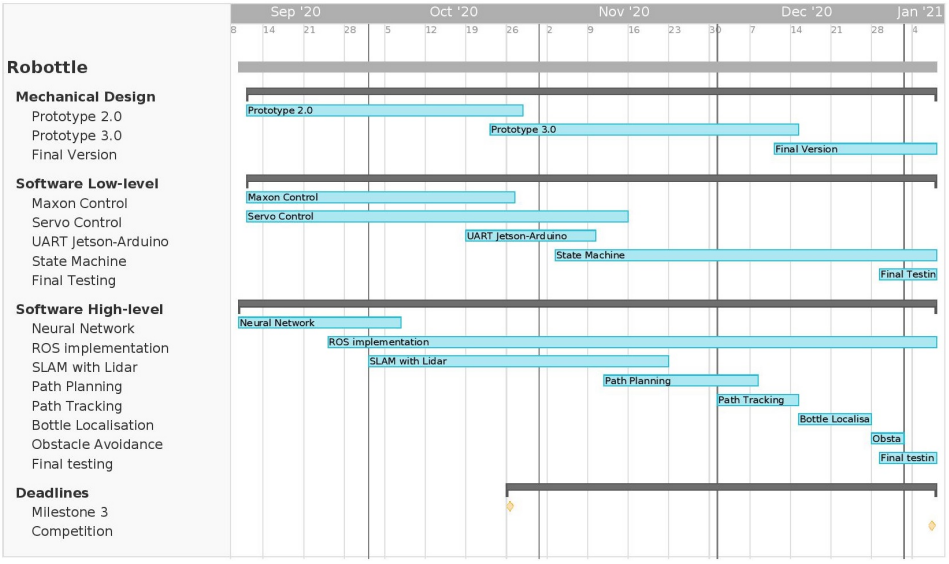
With the real budget, the two main components bought were the Lidar and the Jetson Nano. In the end, with all the materials, connectors and other random elements, the amount spent was 313.33 CHF, leaving a balance of 436.67 CHF. Combining both balances, in total we spent 1013.48 CHF out of a budget of 2'750 CHF.

7.2 TIMELINE OF THE PROJECT

Here are the timeline presented at the third Milestone and what the actual timeline ended up being:



(A) Predicted Timeline at Milestone 3



(B) Final Timeline

The predicted timeline is quite close to the final management of the tasks. On the mechanical design side, everything went as planned and the final version of the robot was started exactly at the expected time. For the low-level software, the servo motors ended up taking extra time in November, due to an unforeseen error in the code, and also a state machine was added, which was constantly modified up until the end. In the high-level software, the SLAM took longer than expected to implement fully, and this caused the bottle localization to be pushed to late in the semester. Finally the obstacle avoidance was not considered initially but ended up taking about a week to correctly implement. Overall there were some unexpected delays, due to an underestimation of the required time to finish some tasks, but because we left about two weeks of margin in our plans, we finished everything in time.

CHAPTER 8

CONCLUSION

8.1 ACHIEVEMENTS

We are very proud of what we were able to achieve in one semester. We tried to keep the mechanical and electronic design as simple as possible and build a robust software to go with it, and we are very happy with the result. We chose to not 3D-print anything and built the robot entirely with laser-cut wood, and although it would have been a good opportunity to learn how to use a 3D printer, for the scope of this project it was the right decision as it simplified a lot the prototyping phase, especially given our inexperience in the field. With the software, we looked for state-of-the-art solutions to problems such as localization, path finding and object detection, and even if it wasn't an easy task, we made everything work and gained a lot of experience in many complicated and useful algorithms. Despite the disappointing result at the competition, we think we made the right design choices and are happy with how Robottle turned out.

8.2 TIPS FOR FUTURE PARTICIPANTS

If the objective is to purely win the competition, then our path of focusing on state-of-the-art software solutions might not be the best idea, as there are a lot of less complicated and easier to implement algorithms which could achieve the same or better results in the controlled environment of the competition. But if instead the objective is to learn, we recommend our approach, as no other project at EPFL taught us this much.
One thing that we could have done differently is to test more the mechanical solutions, as especially the sticky arm gave us a few problems up until the end. In general though everything worked as expected and we were able to anticipate and solve most of the problems that arose over the course of the project. There are always a few sudden problems that appear unexpectedly no matter what, therefore it is a good idea to keep one or two extra weeks of leeway in case something happens.
We also highly recommend to start prototyping very early on, since it is what allowed us to decide quickly what the important key features and solutions of our robot were. Finally, go test in the arena as soon as possible, as many problems can only be observed in the final environment.

8.3 ACKNOWLEDGMENTS

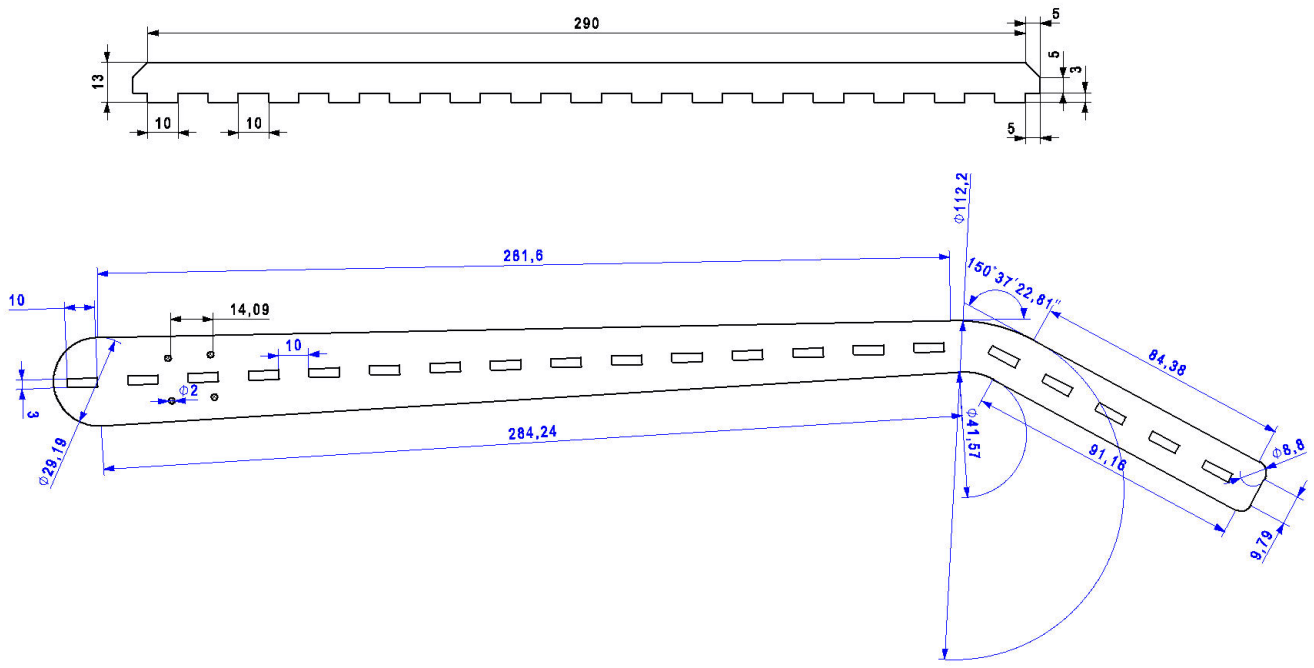
Firstly we would like to thank Alessandro Crespi for his help and advice. He spent a lot of time helping us solve many problems and without all of his hard work, this competition wouldn't be possible. Thank you also to Professor Auke Ijspeert supporting the robotic competition and giving us this great opportunity.

We want to thank our coach Hala Khodr which assisted us over this project. Finally we want to thank the people at SKIL for helping us and the other teams for the good times spent together especially during the intense last week.

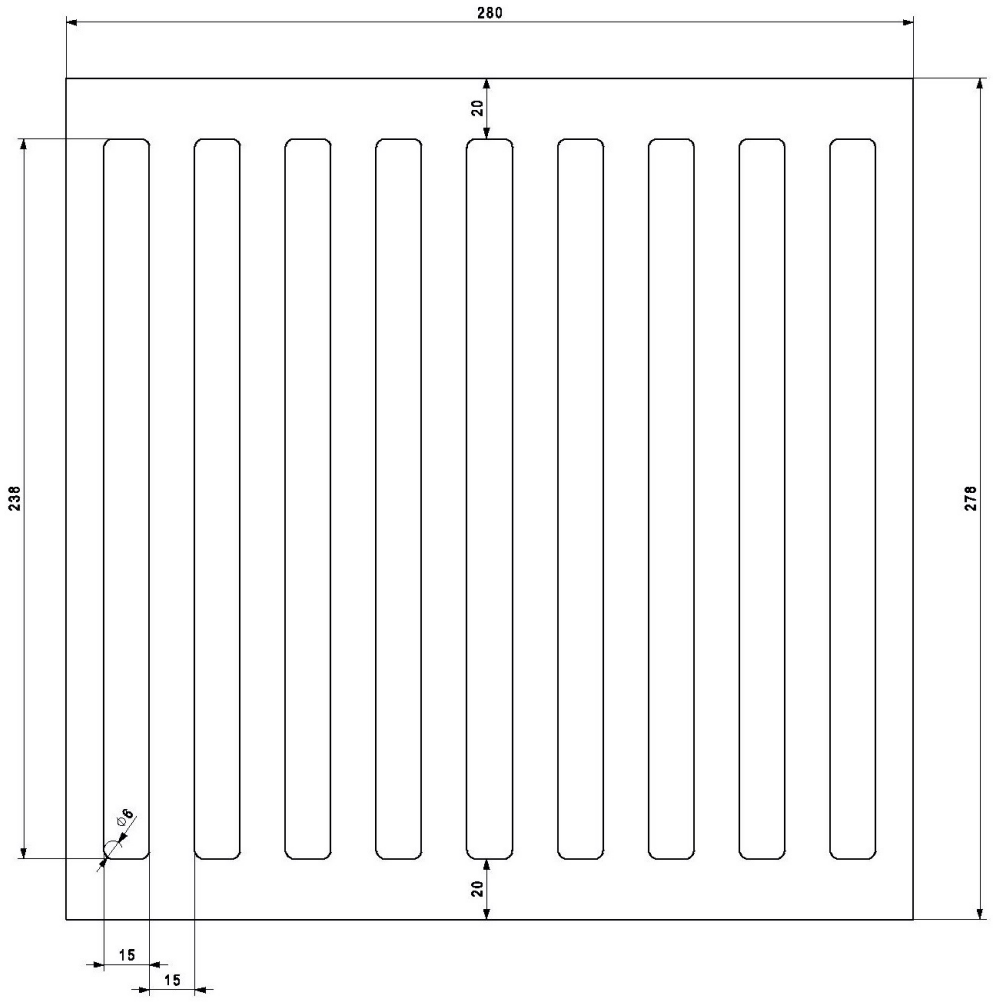
APPENDIX A

2D DRAWINGS

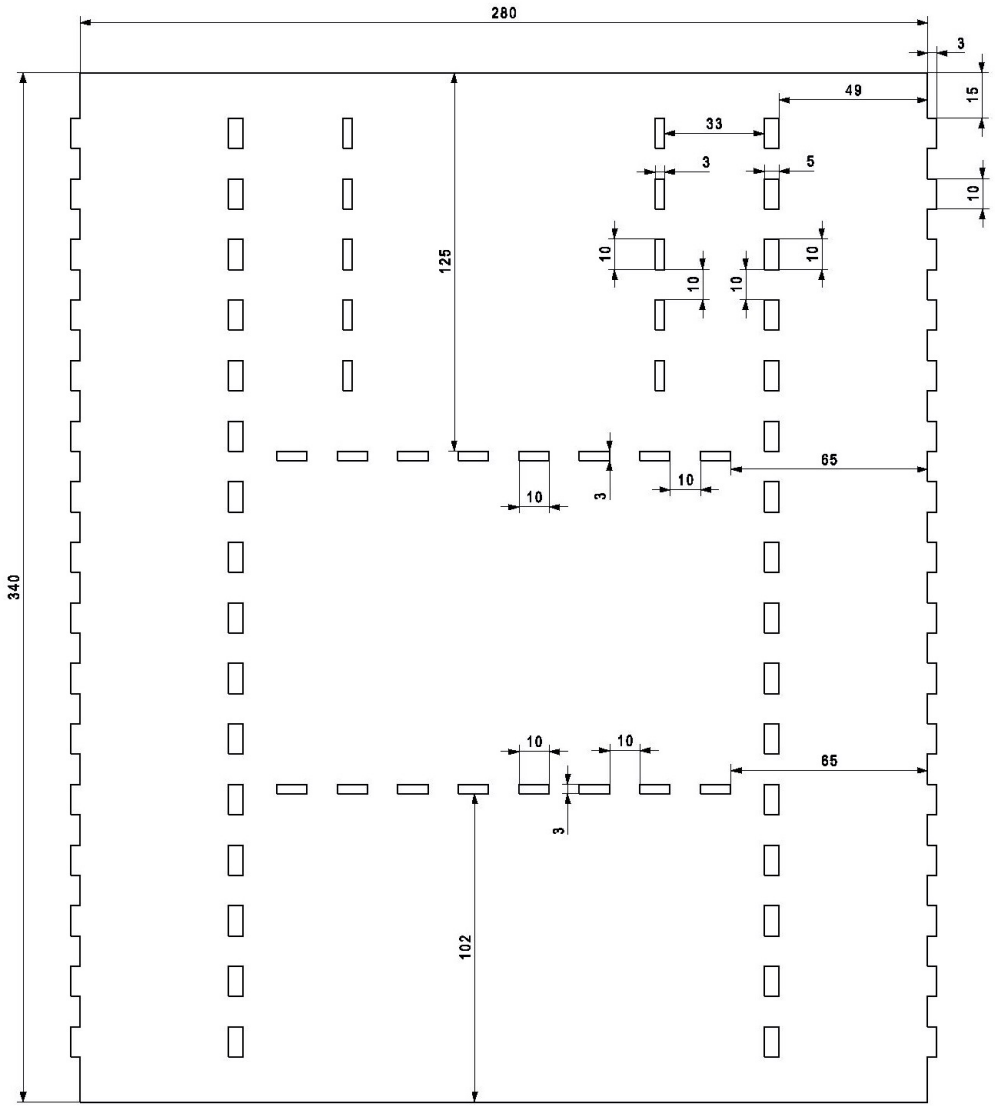
ARM



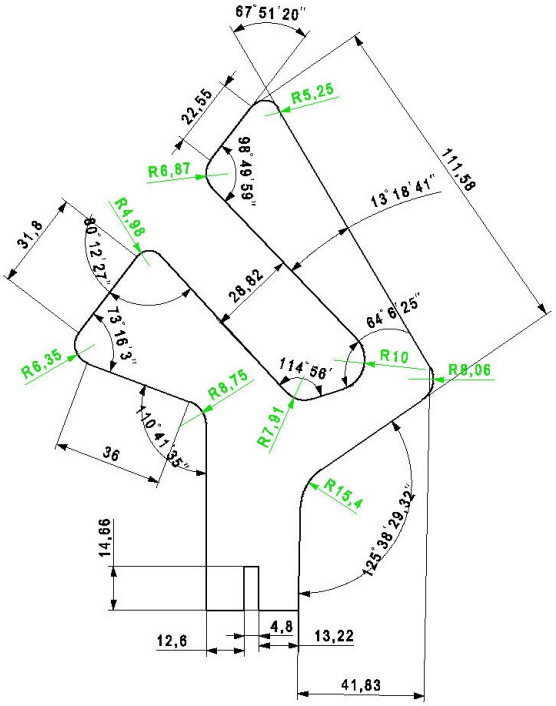
BACK DOOR



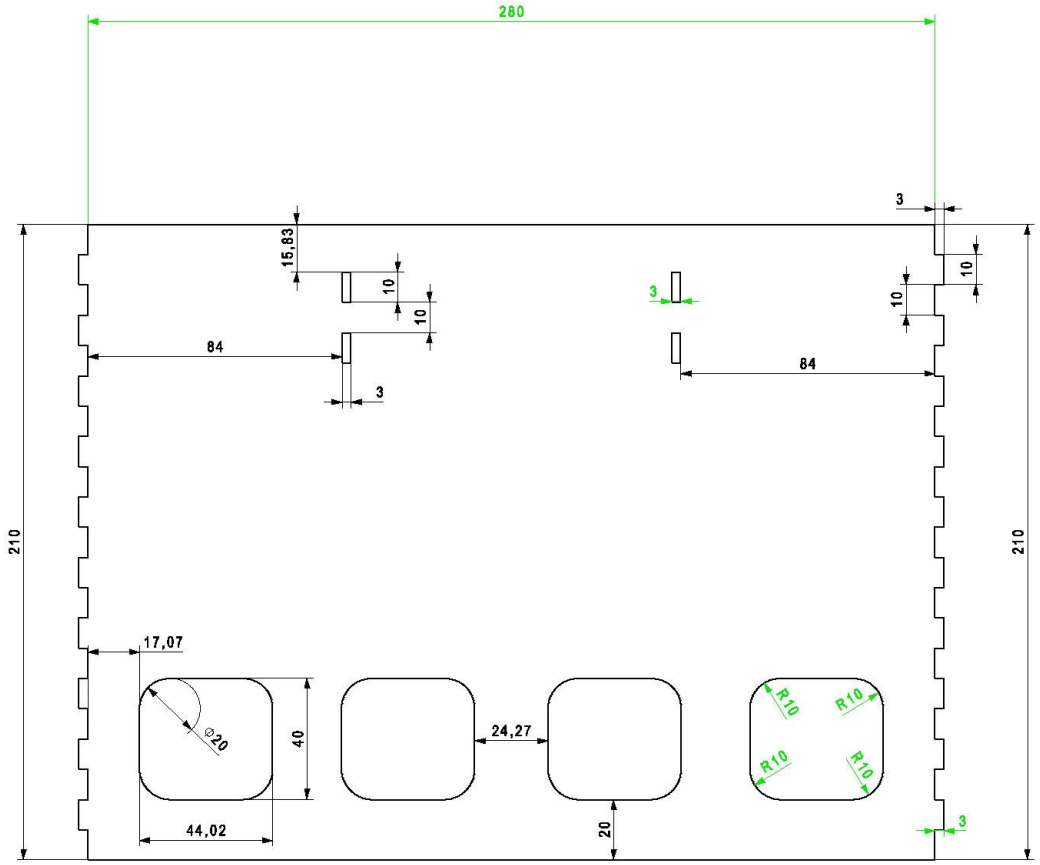
BOTTOM



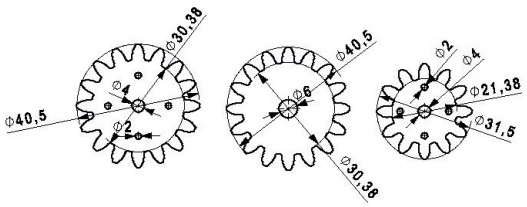
PASSIVE DETACHER



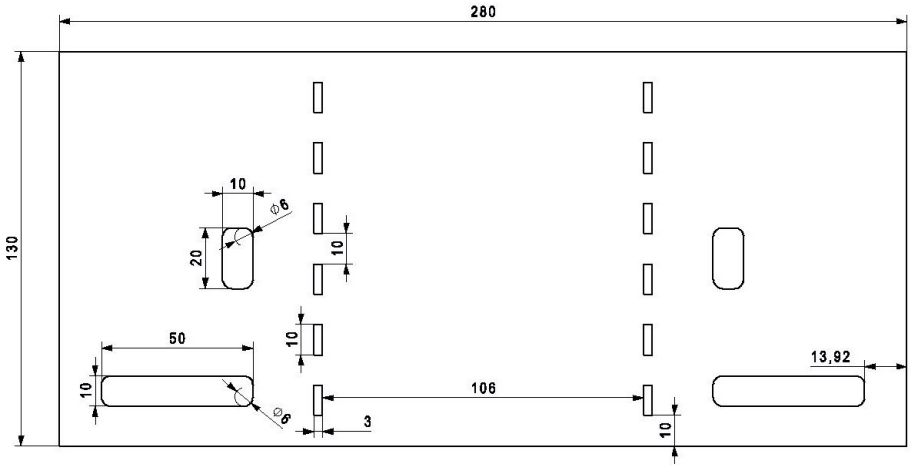
FRONT WALL

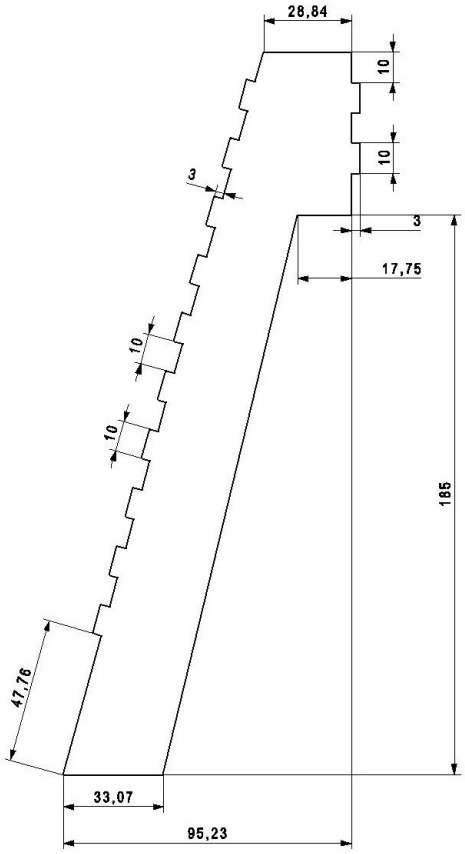


GEARS

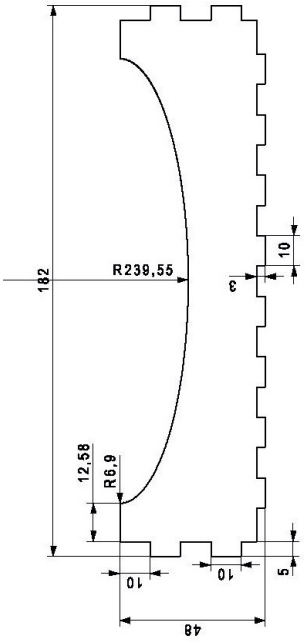


SUPPORT ELECTRONICS

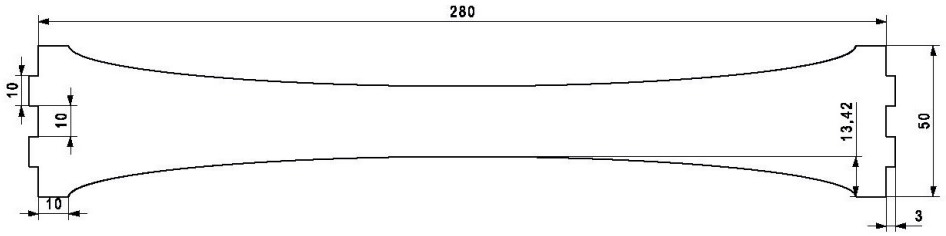




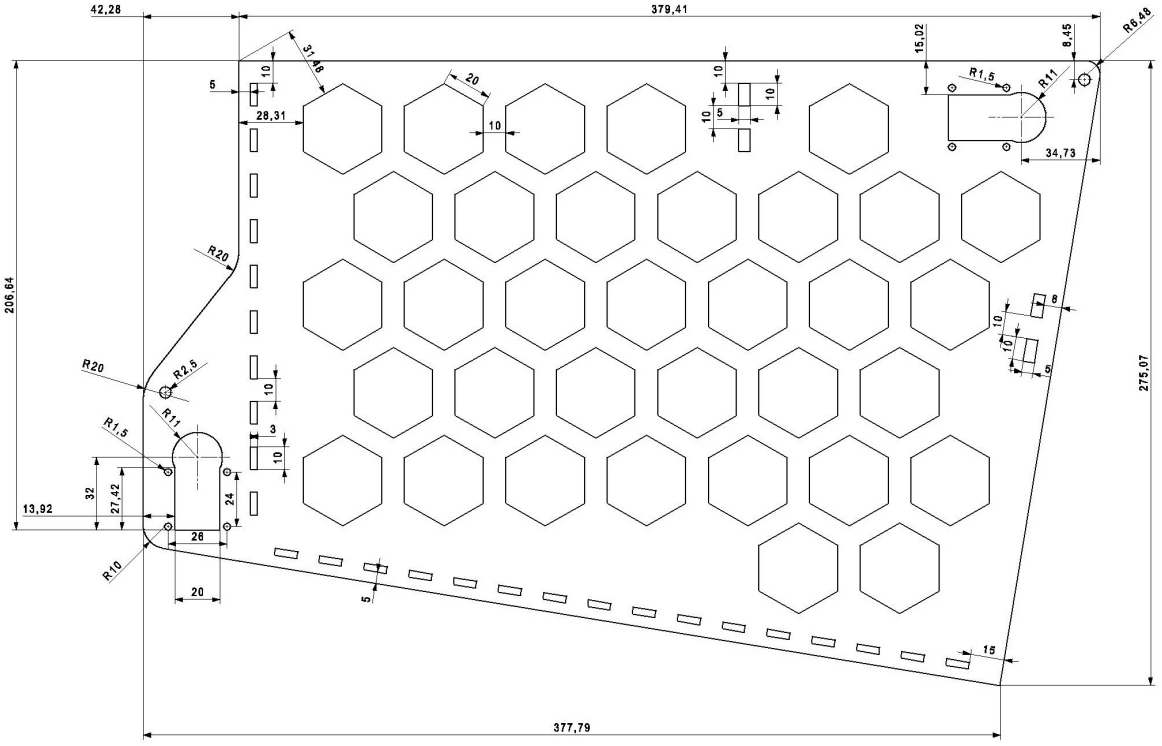
REINFORCE BOTTOM



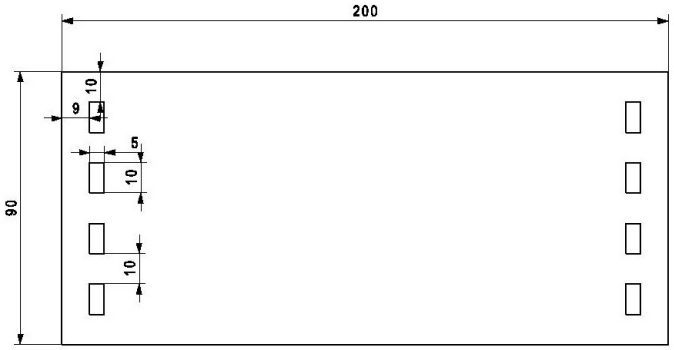
REINFORCE BOX



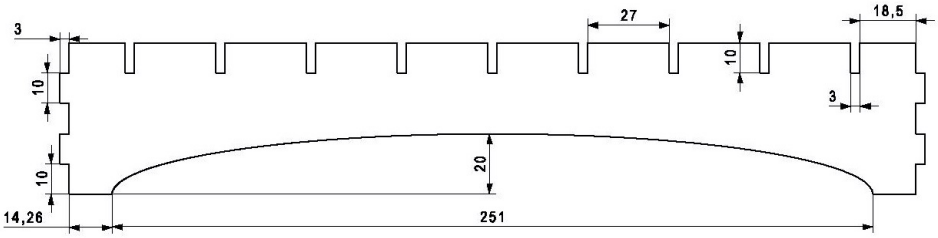
SIDE WALL



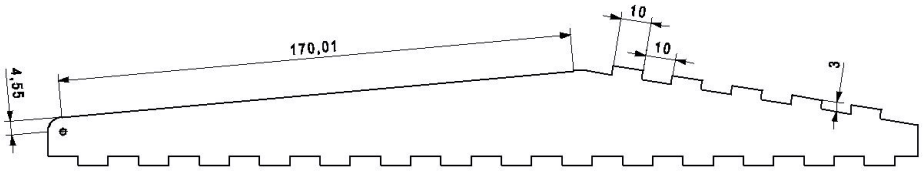
SUPPORT BACK WHEEL

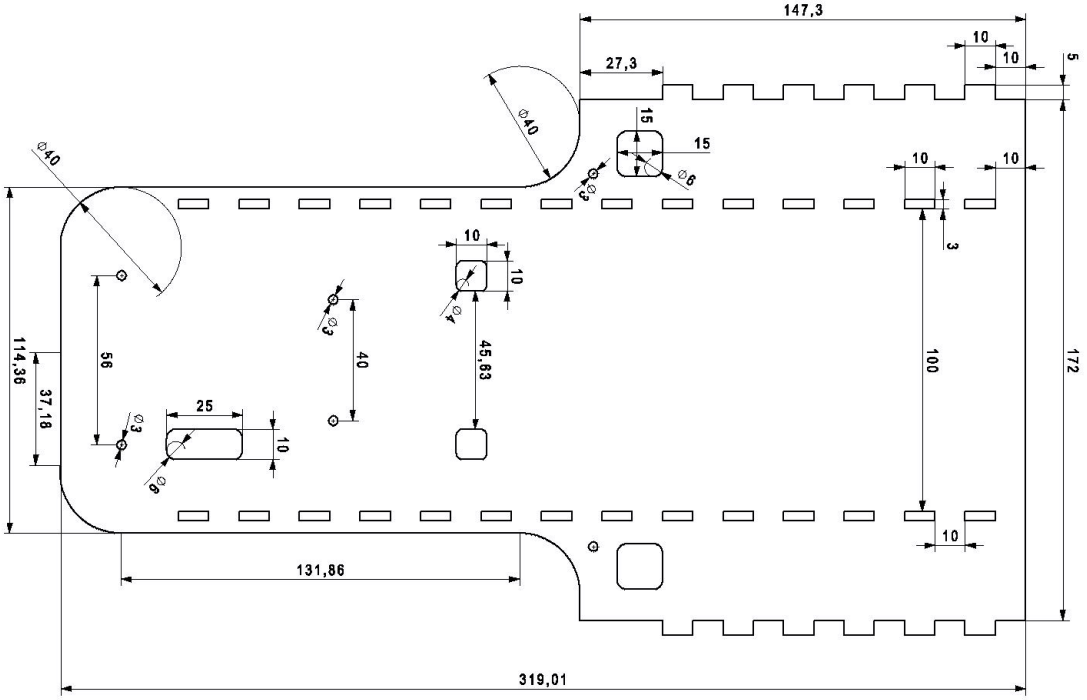


SUPPORT PASSIVE DETACHER



SUPPORT LIDAR





BIBLIOGRAPHY

[1] Sertac Karaman and Emilio Frazzoli. 'Sampling-based algorithms for optimal motion planning'. In: *The International Journal of Robotics Research* 30.7 (2011), pp. 846–894. DOI: 10.1177/0278364911406761. eprint: https://doi.org/10.1177/0278364911406761. URL: https://doi.org/10.1177/0278364911406761.

[2] B. Steux and O. E. Hamzaoui. 'tinySLAM: A SLAM algorithm in less than 200 lines C-language program'. In: *2010 11th International Conference on Control Automation Robotics Vision*. 2010, pp. 1975–1979. DOI: 10.1109/ICARCV.2010.5707402.

[3] Norhidayah Yatim and Norlida Buniyamin. 'Particle filter in simultaneous localization and mapping (Slam) using differential drive mobile robot'. In: *Jurnal Teknologi* 77 (Dec. 2015). DOI: 10.11113/jt.v77.6557.