University of Insubria

Department of Theoretical and Applied Science (DiSTA)

Master Degree in Computer Science

Tankerino
AI lawnmower

Thesis Advisor: Dott. Gallo Ignazio

Candidate: Francesco Belloni

Matricula: 725676

Academic Year: 2020-2021

Abstract

One of the most boring jobs in the garden is cutting the grass: the automation of this task can only be seen positively. The goal of this project is to produce an autonomous lawnmower prototype. The lawnmower should use sensory devices to detect the aspects of the mowing field and use this information to navigate the area with no user input, mowing as much of the field as possible while avoiding obstacles and staying on the mowing area. In this project, we wrote all the code to tailor the software for the specific needs of the autonomous lawnmower. This choice lengthened the development times and allowed us to understand every detail. Because of our financial limitations, the most difficult challenges will be perfecting the project's localization system and meeting the objectives mentioned above while minimizing the lawnmower's costs. This document details our approach to research, simulation, implement and testing the lawnmower.

Acknowledgement

I would like to thank my supervisor, Professor Gallo Ignazio, who gave me this opportunity to work on this project. I got to learn a lot from this project about robotics. I would like to express the most sincere gratitude.

I want to extend my heartfelt thanks to my brother, Stefano Belloni, parents, and wife Francesca. Without their help, this project would not have been successful.

Contents

Chapter 1

Introduction

The construction of an automatic lawnmower is a pretty complicated project. It will be composed of three subsystems: mechanical, electrical, and controls.

- the mechanical system includes the chassis, mower deck and blades, wheels and motors
- the electrical system is divided into two subsections, the control electronics and the sensors
- controls involve software algorithms loaded onto the control electronics.

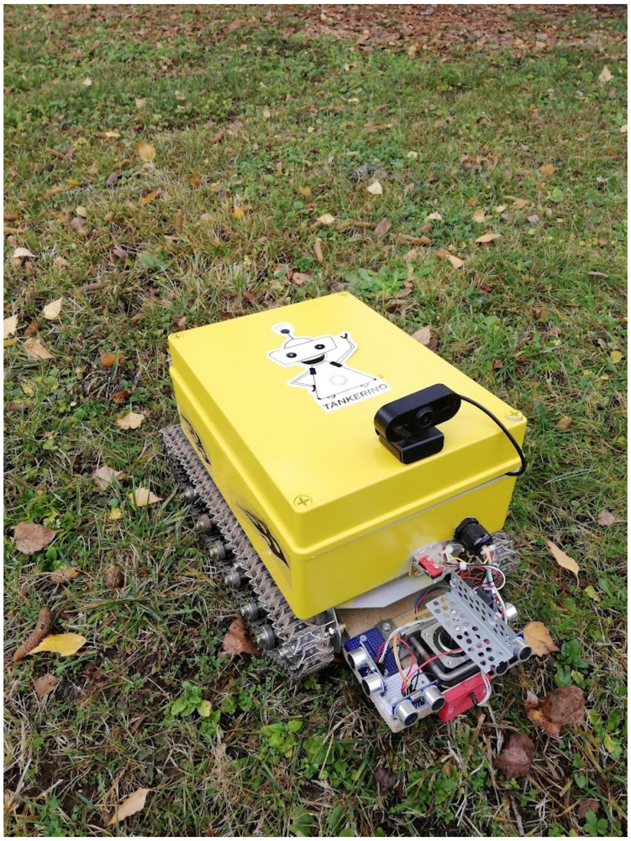


Figure 1.1: Tankerino

This project uses a variety of features to undertake its job. The primary duty necessary for automation is to track the mower's location and its environment. Tracking and mapping are performed using a combination of sensors, wheel shaft encoding and compass bearing detection. The combination of all this data provides the precise measurements required to navigate the lawnmower. The robot must also implement an obstacle avoidance system.

In this thesis we are going to focus primarily on the theoretical and practical problems involved in the development of an autonomous driving car that has to cover an area.

In **Chapter 2** we lay the theoretical foundations: we consider the problem to navigate a robot from a starting point to a destination point as well as find the best strategy to cover an unknown area. We briefly review some of the most common approach in the literature and concentrate on the algorithms that identifies the best strategy to complete the given task.
We describe and analyse the mathematical model used in this thesis that will allow up to implement the software to drive the lawnmower.

Chapter 3 is dedicated to the design and description of the Hardware used to build the prototype of an autonomous lawn mower. To calculate and monitor the position of a robot a lot of sensors and methods are used: in this chapter we will go through each of these sensors and discuss the strengths and weaknesses associated with them as well as the Hardware employed: `Raspberry PI` and `Arduino`.

Chapter 4 describes the developed software. `python` is used as a programming language for `Raspberry Pi`, while `C` and `C++` are used with `Arduino`. All the software was written without relying on specific robotics libraries. This choice has allowed us to study and understand all the steps necessary to develop an autonomous-driving robot. Several implementations have been tested to minimize the errors for many of the obstacles that occurred during the developing time.

Various experiments are described and analyse in **Chapter 5**. We performed different type of experiments: from a simulated lawn mower with different level of realism to experiments done on the field with the build prototype and analyse them to identify strengths and weaknesses of the implementation and suggest possible improvements and future work.

There were several difficulties encountered during the development of the prototype. One example was writing a robust code to manage the packets loss between `Raspberry Pi` and `Arduino`. Another example was managing the robot's movement on uneven ground: sometimes the commands sent to `Arduino` are not executed perfectly because commercial sensors are not always accurate.

It is possible to find photos and videos on our website at tankerino.com

Chapter 2

Coverage Path Planning for Robotics

2.1 Navigation Subsystem

The most elementary problems related with the development of an autonomous car are fundamentally two:

a) identifies the start point X and the end point Y to complete a task,

b) describe and control the movement of the given robot from X to Y;

Driving a robot between two points can be a challenge requiring a sophisticated algorithm and high computing power. However, some applications look for cheaper solutions that might only partially complete the task. For example, many robot vacuum cleaners use a simple random algorithm, which consists of an endless loop that in turn performs:

i) drive straight until an obstacle is encountered

ii) turn a random angle, go back to i)

The cleaning quality of such behaviour is not always guaranteed. Still, from a mathematical point of view, if given infinite time, this algorithm will cover the complete cleaning area as long as the robot can physically reach it [4].

Implementing such an algorithm for a robot vacuum cleaner can be justified by considering that the area of a house is limited, its development cost is low, and the cleaning task might be performed regularly, increasing the overall cleaning result.

Modern lawnmowers often used such algorithms.However, those algorithms for our specific task do not seem optimal:

- a lot of energy is wasted
- the covered area might be pretty big
- random paths on the lawn might be visible

In this thesis, we tried to build a robot that followed objectives to cover the whole area efficiently.

2.1.1 Grid Map and Shortes Path

The primary problem that we have to solve is given two points A and B which are the intermediate point that the robot has to go through, i.e. which is the trajectory it should follow. At this point it is convenient to split the problem into even smaller pieces: for the moment we take for granted that we are able to drive a robot through an *elementary path*, i.e. a path that involves only straight line and rotations. **Section 2.3** we will deal explicitly for the solution of this particular problem.

So let us consider a given map that we want to explore: for simplicity let's say that the map is rectangular. It is natural to build a grid that covers the map and consider the center of each element of the grid, let say $m_{i,j}$ for the element at position (i, j) in the map.
We can now build an edge from each of the element $m_{i,j}$ to the next one among its neighbors which can be reached from it: In this way we have build a graph $(\{m_{i,j}\}_{i,j}, \{(m_{i,j} \to m_{n,m})\})$. It now possible to use one of the many algorithm on graphs to find a path between two vertexes $(m_{i,j})$ and driving the robot along this set of *elementary paths*[1].
A variety of algorithms exists to solve the problem [4], for example

- **Dijkstra's algorithm** solves the single-source shortest path problem with non-negative edge weight [7].
- **Bellman-Ford algorithm** solves the single-source problem if edge weights may be negative [3].
- **A* search algorithm**solves for single-pair shortest path using heuristics to try to speed up the search [11].
- **Floyd–Warshall algorithm** solves all pairs shortest paths [9].
- **Viterbi algorithm** solves the shortest stochastic path problem with an additional probabilistic weight on each node [24].

Obstacle Avoidance Subsystem

Once the mower is on its own on the fields, and it moves around, it is important to prevent that it impacts into obstacle, that it does not somersault or does not damage itself:

The subsystem devoted to this task is the *Obstacle Avoidance Subsystem*: it is designed to function as a smaller part of the whole Software Architecture and has the purpose of detecting, through various sensors, obstacles in the vehicles environment that are interpreted by the software to avoid possible collision or damages.
In the present prototype two types of sensors are used:

- five *proximity sensors*

[1] In the case presented all edges will have two classes of distances if all 8 neighbors are allowed or just a single distance if only the perpendicular neighbors are allowed. It might be consider more complex mapping. For example let the map be made up of two rooms connected with a long corridor. now the two vertexes at the end of the corridor can be simply directly connected taking into account that the wight of the edge has to be the correct one.

- and one *lidar*.

By using these two input in conjunction, we provide the data required for the lawnmower to avoid any hazards while performing its functions.

2.2 Coverage Path Planning

Coverage Path Planning (CPP) is the task of determining a path that passes over all points of an area or volume of interest while avoiding obstacles. This task is integral to many robotic applications, such as vacuum cleaning robots, painter robots, autonomous underwater vehicles creating image mosaics, demining robots, lawn mowers, automated harvesters, window cleaners and inspection of complex structures, just to name a few. A considerable body of research has addressed the CPP problem [10].

2.2.1 A Survey on Coverage Path Planning for Robotics

As identified in [10] there are 10 main topics: of these a subset might be relevant for this thesis. We will briefly present them and point out the strength and the weak aspect in relation with out task. At the end of this survay we summariese of the algorithm used in the implementation, which extends some of the concept introduced here as well as some new ideas.

Classical Exact Cellular Decomposition Methods

Exact cellular decomposition methods break the freespace (i.e., the space free of obstacles) down into sim-ple, non-overlapping regions called cells. The union of all the cells exactly fills the free space. These regions, which contain no obstacles, are "easy" to coverand can be swept by the robot using simple motions. For instance, each cell could be covered using a zigzag,"mowing the lawn" pattern [14].
In this approach the map is known in advance: in this thesis we consider however also online coverage path planning of unknown environment.
Typically, a planner based on exact cellular decomposition generates a coverage path in two steps.

- First, it decomposes the free space into cells and stores the de-composition as an adjacency graph.
- Next, it computes an exhaustive walk through the adjacency graph (i.e.,a sequence that visits each node in the graph exactlyonce).

It is worth noting that the exhaustive walk obtained is a sequence of nodes and not an actual coverage path. Therefore, an explicit path for covering each cell must be derived using simple motions as discussed above[10].

Morse-based Cellular Decomposition

In [2] the authors propose a cell decomposition of the space based on critical points of Morse functions he Morse-based decomposition has the advantage of handling also non-polygonal obstacles. By choosing different Morse functions, different cell shapes are obtained [10].
We will not dig into the ditails of the algorithm, but present the intuitive idea behind it. Let us consider a rectangular 2D map where a set ob obstacle are present and consider a vertical line which sweeps the space. As this line moves, it will intersect (or stop to intersect) an object. In this way a number of regions are identified. This construction can be make rigorous with the help of critical point of function. By changing such functions, different decompositions are obtained.
An online extention was developed in [1]: To detect the critical points, they use an directional range sensor to look for points where the surface normals of the obstacles and the sweep direction are parallel.

Grid-based Methods

Grid-based methods use a representation of the environment decomposed into a collection of uniform grid cells. This grid representation was first proposedby Moravec and Elfes [15] to map an indoor environment using a sonar ring mounted on a mobile robot. In this representation, each grid cell has an associated value stating whether an obstacle is present or if it is rather free space. The value can be either binary or a probability of finding an obstacle in the cell. Typically, each grid cell is a square, but also different grid cell shapes can be used, such as triangles.
By using a grid representations one can only approximate the shape of the target region and its obstacles, this is also referred to as approximate cellular decompositions [5]. As a result of this approximate representation, most grid-based methods are *resolution-complete*, that is, their completeness depends on the resolution of the grid map.

Example 2.2.1. $\epsilon^\star$: An Online Coverage Path Planning Algorithm In [21] a new algorithm for online coverage path planning of unknown environment is presented: $\epsilon^\star$. The algorithm is built upon the concept of an Exploratory Turing Machine (ETM), which acts as a supervisor to the autonomous vehicle to guide it with adaptive navigation commands. The ETM generates a coverage path online using Multiscale Adaptive Potential Surfaces (MAPS), which are hierarchically structured and dynamically updated based on sensor information. $\varepsilon^\star$-algorithm is computationally efficient, guarantees complete coverage, and does not suffer from the local *extrema* problem.

2.3 Vehicle Description and Mathematical Model

The previous discussion needs to be implemented on a real vehicle, which will run some kind of software to perform a task and control and monitor that everything is precisely followed. For this goal it is necessary to have an adequate mathematical model.
In particular

- model the vehicle dynamic with a model that is sufficiently representative;
- study the trajectory that such a vehicle follows for a given set of input
- reconstruct the trajectory from some sampling data.

In the following we address these three problem and propose a solution to them.

In order to find a sufficiently representative mathematical model of the system we are going to consider it necessary to describe it briefly: more elaborate description of the physical object can be find in **Chapter 3**. In particular in this section we concentrate to identifies which are the key variable that allows us to capture the dynamics of the vehicle which are studied in the next subsection.
The vehicle has four wheels: two independent driving wheels place in the back of the chassis and can rotate in other directions; two the front wheels that are only there support the structure with no active role (see **Figure 2.1**).

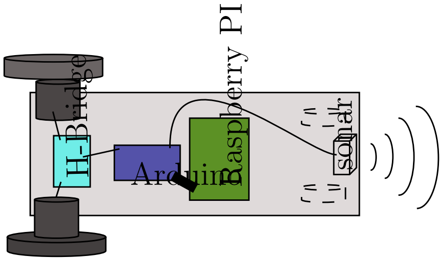


Figure 2.1: Model of the mower: two driving back wheels and two supportive front wheels. The two driving wheels are independent and can rotate in other directions.

First of all, we will consider a plane-parallel motion solid body: this will make the model a lot more easy and will not loose too much of generality since the third dimension plays a minor role.
In particular When the two wheels rotate with the same angular speed, then the vehicle follows a straight line. If the two wheels have different speed, then the vehicle rotates in such a way that two wheels describe two concentric circles.

In particular we can conclude that the dynamic of the vehicle is completely determined once we know the speed of the two wheels.

In order to introduce the equations that govern the dynamic of the car, let start with an illustrative example.

Example 2.3.1. Let consider the case in which the right motor is not rotating, while the left one spins, the car will rotate right with the left wheel describing a circle with center on the left wheel and radius equal to the distance between the wheels (see **Figure 2.2**).

In this case is straightforward to write the evolution equation for the trajectory, In particular, by indicating the position of the center of the rear axis with $P = (x, y)$, and the direction with θ, during a rotation of the car, the point P.

To identify the center of the circle, it is enough to proceed for half of the width of the mower (let it be $\frac{w}{2}$) perpendicularly to the direction θ, which is equivalent to rotate the vector $(0, \frac{w}{2})$ of $\theta + \pi$:

$$\begin{bmatrix} x_c \\ y_c \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} + \frac{w}{2} \cdot \begin{bmatrix} -\sin\left(\theta + \frac{\pi}{2}\right) \\ \cos\left(\theta + \frac{\pi}{2}\right) \end{bmatrix}.$$

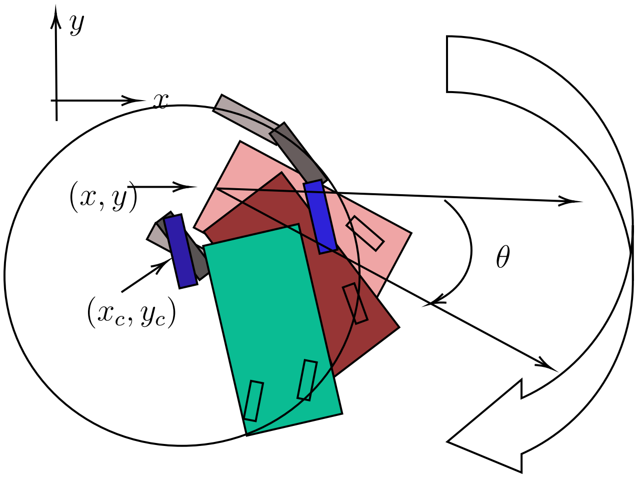


Figure 2.2: Schema of a simple rotation: the right wheel does not rotate occupies the center of the circumference that is described by the other wheel as well as the middle point of the rear axis.

In order to derive the general equations of the dynamics one has to identify the center of rotation of the vehicle rotate and its angular speed of the. Let us consider the **Figure 2.3**

The speed of a wheel is tangential to the trajectory and since we are dealing with a rigid body rotating around a fix point, all elements have the same angular speed: From the relation between angular speed (ω) and speed of a point distance r from the center, i.e.

$$\omega = \frac{r}{r}$$

we derive

$$\omega = \frac{v_r}{R + \frac{l}{2}}, \quad \omega = \frac{v_l}{R - \frac{l}{2}}$$

and solving for R and ω we have:

$$R = \frac{l}{2}\frac{v_l + v_r}{v_r - v_l}, \quad \omega = \frac{v_r - v_l}{l} \tag{2.1}$$

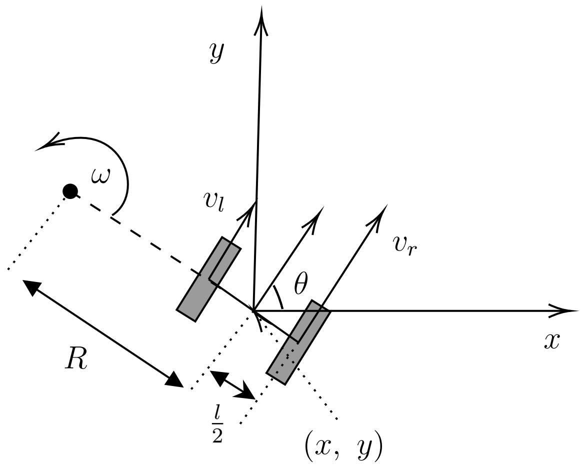


Figure 2.3: General Rotation of the vehicle: The speed of the two wheels identifies the center of rotation

In particular it is possible to identify the center of the rotation (ICC) in the same way as it was identify in the example:

$$ICC = [x - R \cdot \sin(\theta), y + R \cdot \cos(\theta)]. \tag{2.2}$$

Forward Kinematics for Differential Drive Vehicle

Consider now the situation when a robot is at position $P(t) = (x, y)$ at time t and let study what is its next position after a time δt, i.e. $P(t + \delta t)$.
From the previous section we might *simulate* the movement with the following algorithm:

1. translate the ICC to the origin of the coordinate system
2. rotate about the origin by an angular amount $\omega \cdot t$
3. translate back in the ICC

In formula this can be translate into

$$\begin{bmatrix} x(t+\delta t) \\ y(t+\delta t) \\ \theta(t+\delta t) \end{bmatrix} = \begin{bmatrix} \cos(\omega \cdot t) & -\sin(\omega \cdot t) & 0 \\ \sin(\omega \cdot t) & \cos(\omega \cdot t) & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x(t) - ICC_x \\ y(t) - ICC_y \\ \theta \end{bmatrix} + \begin{bmatrix} ICC_x \\ ICC_y \\ \omega \delta t \end{bmatrix} \tag{2.3}$$

This equation simply describes the motion of a robot rotating a distance R about its ICC with an angular velocity of ω [23].

Inverse Kinematics of a Mobile Vehicle

In general it is possible to describe the evolution of the position of an object if its dynamic is known by simply integrating them, i.e.
Let us consider a rigid body, whose angular velocity is given by $\omega(t)$ and whose speed is given by the vector $V(t)$, two functions of time, then

$$
\begin{aligned}
x(t) &= x(0) + \int_0^t V(s)\cos(\theta(s))ds \\
y(t) &= y(0) + \int_0^t V(s)\sin(\theta(s))ds \\
\theta(t) &= \omega(0) + \int_0^t \omega(s)ds
\end{aligned}
$$

and by using the formula derived previously, we have [23]

$$
\begin{aligned}
x(t) &= x(0) + \frac{1}{2}\int_0^t [v_r(s) + v_l(s)]\cos(\theta(s))ds \\
y(t) &= y(0) + \frac{1}{2}\int_0^t [v_r(s) + v_l(s)]\sin(\theta(s))ds \\
\theta(t) &= \omega(0) + \frac{1}{l}\int_0^t [v_r(s) - v_l(s)]ds
\end{aligned}
\tag{2.4}
$$

2.3.1 Trajectory of a Differential Vehicle

The previous paragraphs aimed to study the dynamics of a differential vehicle in order to try to answer the question about *how can a robot be controlled in order to reach a given configuration*: this is a known as *inverse kinematics problem* [23].

Unfortunately, a differential drive robot imposes what are called *non-holonomic* constraints on establishing its position. For example, the vehicle cannot move laterally along its axle, but can only perform two type of movements:

(F) follow a straight line

(R) rotate.

Luckily enough only this small set of allowed movements are enough to specifies which trajectory such a robot should follow to reach point B when its initial position was A.

In geometry such paths are called **Dubins path** [8].

Dubins path

In geometry, the term **Dubins path** typically refers to the shortest curve that connects two points in the two-dimensional Euclidean plane (i.e. $x - y$ plane) with a constraint on the curvature of the path and with prescribed initial and terminal tangents to the path, and an assumption that the vehicle traveling the path can only travel forward. If the vehicle can also travel in reverse, then the path follows the Reeds-Shepp curve.Since the traditional Dubins path are enough for the porpose of this thesis the more complex Reeds-Shepp curves are not considered, even though the build robot it able to perfrom such trajectories.

In 1957, Lester Eli Dubins [8] proved using tools from analysis that any such path will consist of maximum curvature and/or straight line segments. In other words, the shortest path will be made by joining circular arcs of maximum curvature and straight lines.

The optimal path type can be described using an analogy with cars of making a *right turn* (`R`), *left turn* (`L`) or driving *straight (S)*. An optimal path will always be at least one of the six types: `RSR, RSL, LSR, LSL, RLR, LRL`. In particular it always starts with a rotation. The calculation of such paths follows from geometric considerations.

Construction of Dubins Path

The model we have described above (also known as **Dubins car**) fits in the theory of *Dubins path.* In particular, let consider the case in which a path planner identifies the starting and ending point for a trajectory along with the direction it should have: we have only to calculate the corresponding Dubins path and follow them. To keep things simple, only one wheel will be activated when rotating: in this way, the circle's radius is half of the length of the rear axis.

At this point the algorithm is very simple:

1. calculate *Dubins path* between $A = (x_A, y_A, \theta_A)$ and $B = (x_B, y_B, \theta_B)$:
$$\mathcal{D}(A, B) = S_1 \circ S_2 \circ S_3,$$
where the S_i are the three different segments that constitute a path.
2. (S_1) activate the correct wheel to rotate for a given angle
3. (S_1) activate the correct wheel to rotate for a given angle or both to go straight
4. (S_1) activate the correct wheel to rotate for a given angle till the final position.

There are different approach to identify the Dubin paths connecting two point. In this section we give a intuitive geometrical description that might be instructive (See **Figure 2.4**):

1. draw the two circumferences $\mathcal{C}_A$ (with centre C_A) and $\mathcal{C}_B$ (with centre C_B) tangent to the starting and end point. The diameter is the distance between left and right wheel of the vehicle.
2. identify the line τ (or the circumference if it exists) tangent to $\mathcal{C}_\mathcal{A}$ and $\mathcal{C}_\mathcal{B}$ so that the driving direction is coherent.
3. The segments and arch or circles identified make up the possible Dubins path.
4. Compute the length of each found path and select the shortest one (in the figure the blue and red arcs and the green line.).

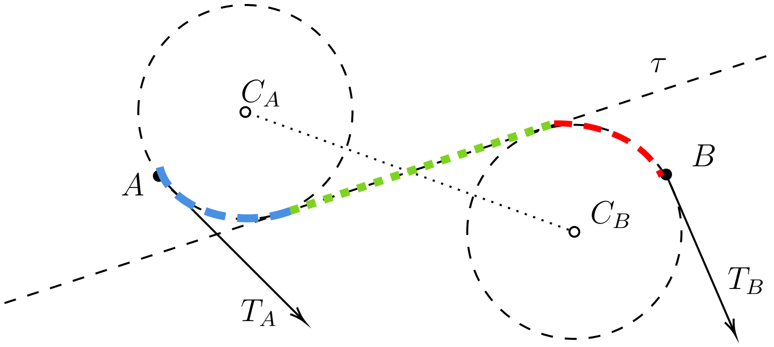


Figure 2.4: Geometric construction of a Dubins path between points A and B: First draw the circumferences tangents to the starting point and to the destination point, then identify the tangent of the two circumference.

There exists analytic solution for this problem [19], that will outperform a naive geometric implementation ([25]).

At this point the majority of theoretical and modeling aspect have been described and developed: the lawn mower needs a software, two wheels and a motor. We can now dirty our hands by building a functional prototype.

Chapter 3

Mechanical and Hardware Design

In this chapter, we briefly analyze the sensors and actuators used in the project.

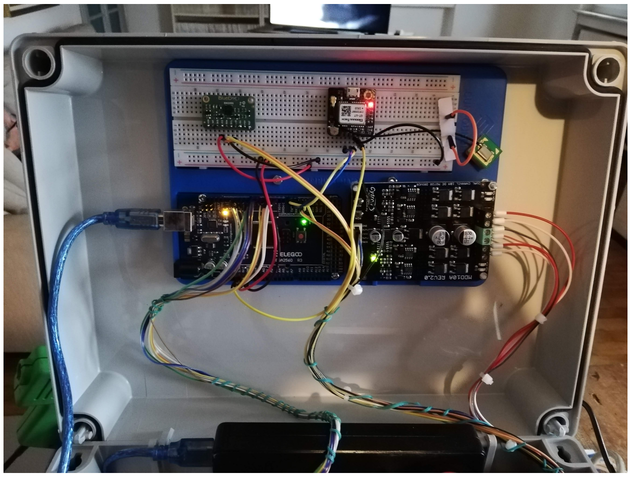


Figure 3.1: This image shows some mower sensors: compass, h-bridge, Arduino, and GPS.

3.1 Techniques and Sensors Used in Mobile Robot Positioning Systems

The robot uses a combination of methods and sensors to calculate and monitor the position of the mower. Most mobile robot positioning applications use two techniques, one from each of the following groups: [6]

- Group 1: Relative Position Measurements (Dead-Reckoning)
  - Odometry
- Group 2: Absolute Position Measurements (Reference-Based Systems)
  - Magnetic Compasses

- GPS
- Inertial Measurement Unit (IMU)

This chapter will go through some of these techniques and discuss the strengths and weaknesses of each method. The automated vehicle will use a combination of GPS, wheel encoders, magnetic compasses and Lidar (Time-of-Flight Infrared Rangefinder).

3.1.1 Odometry

Odometry is a part of SLAM problem. It estimates the agent/robot trajectory incrementally, step after step, measurement after measurement.

newstate = oldstate + step measurement

Incremental change can be measured using various sensors. We use wheel encoders.

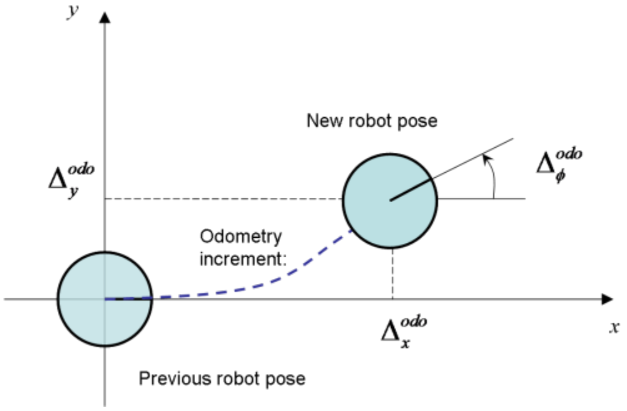


Figure 3.2: This image shows how to estimate the robot trajectory after an action with odometry [18].

Odometry provides good short-term accuracy, is relatively inexpensive and usually has very high sampling rates. The drawback of odometry is the accumulation of errors. This error becomes more significant as the distance travelled by the mower increases [18].

Non-systematic errors will result from unintended interactions between the surface the robot is traversing and the wheel. In the case of the autonomous lawnmower, non-systematic errors will be introduced into the system due to wheel slippages on a wet lawn. Thus, the equations no longer represent the actual distance travelled since the encoder may not have the means to filter out these errors. [6]

In the application, messages from Arduino sent at regular intervals provide the information necessary to update the position on the map.

3.1.2 Inertial Measurement Unit (IMU)

The lawnmower uses the `Bosh BNO055` to understand the direction where the robot is looking. This chip incorporates two features:

- An accelerometer is a tool that measures proper acceleration. This sensor is exceptionally advanced and can internally manage the formulas necessary to recover the data already calculated.
- A magnetic compass. It is intolerant to cumulative errors but is less accurate.

The `Bosh BNO055` using a combination of accelerometers, gyroscopes, and other electrical sensors, inertial measurement units, can measure orientation, acceleration rates, and rotational changes.

3.2 Obstacle Avoidance Subsystem

As mention in **Chapter 2** the *obstacle avoidance subsystem* is constitute of two set of sensors

- five *proximity sensors*
- and one *lidar*.

By using these two input in conjunction, we provide the data required for the lawnmower to avoid any hazards while performing its functions.

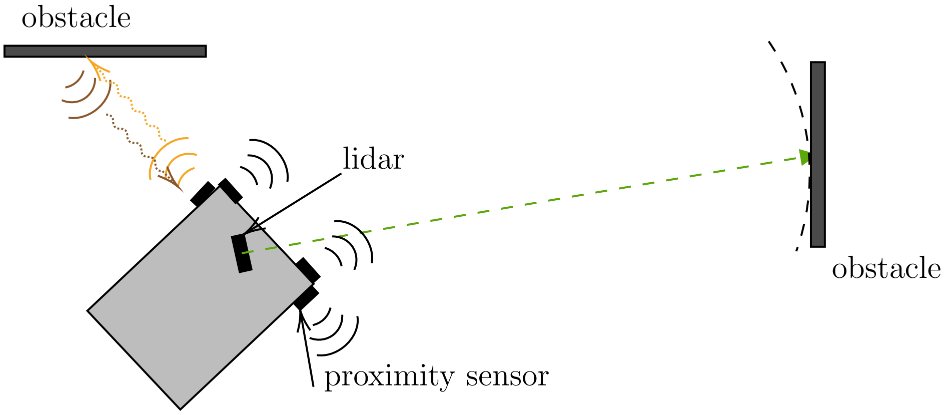


Figure 3.3: Sensor to detect obstacles: proximity sensor and lidar

The primary device used in the obstacles detection and avoidance subsystem is the lidar. This sensor will be mounted onto the front of the lawnmower so that it can detect the obstacles. For use this sensor the the mower has to stop. A stepper motor will then move one degree at a time allowing the mower to scan through 180 degrees. For each degree, the lidar sensor will measure the distance.A stepper motor will then move one degree at a time allowing the mower to scan through 180 degrees. For each degree, the lidar sensor will measure the distance.

The proximity sensor will be the secondary device used in the obstacles detection and avoidance subsystem. These sensors are also mounted on the front of the mower. The ultrasonic sensor should be able to detect objects directly in front of the lawnmower up to a distance of about 6 meters. The ultrasonic distance sensor will provide the microcontroller with real-time awareness of obstacles that are present in the path of the lawn mowers intended route. This will provide an extra layer of protection against collisions with obstructions, especially with obstacles that for whatever reason the lidar has failed to see.[6]

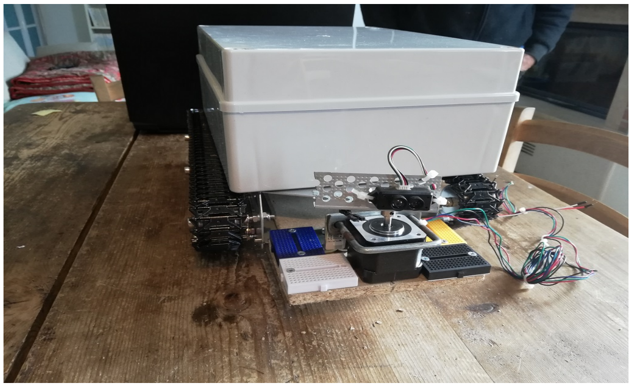
A useful sensor that could be plugged in could be a a collision sensor. The addition of a collision sensor adds an extra safety feature to the lawnmower to help prevent the mower from running over an object in case the obstacle is missed by the proximity sensor.

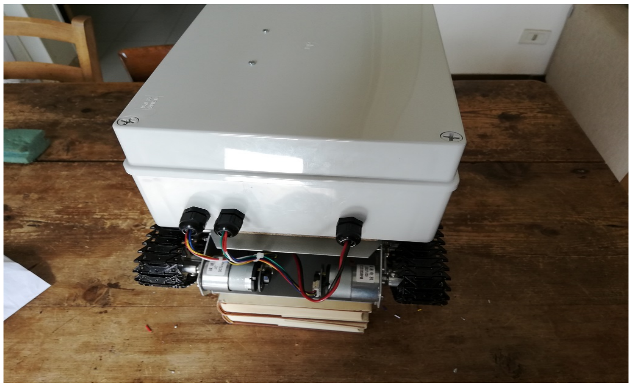
3.3 Motors

There are two types of electric motors:

- A DC motor (Two Metal `DC Geared Motor w / Encoder CQGB37Y001`) that moves the rover. A DC motor is a class of rotary electrical motors that converts direct current electrical energy into mechanical energy.
- A Stepper motor serves to have an accuracy of a degree to allow the lidar to perform reasonably accurate measurements.

Both motors run at 12V.





3.3.1 Motor controller

To adjust the speed of the robot is used an H-bridge. An H-bridge is an electronic circuit that switches the polarity of a voltage applied to a load. These circuits allow DC motors to run forwards or backwards. In addition, they allow the robot to adjust the speed by setting an analogue value from 0 to 255 (encode it digitally).

3.3.2 Stepper motor

A stepper motor is a motor that allows excellent precision. One of the most significant drawbacks, however, is that it fails to recognize its position. The engine can do x degrees, but it is useless if it cannot start from a known point.

For this project, the robot has a `TB6600 Stepper Motor Driver`. The motor uses a mechanical switch to recognize the initial position. Initially, the motor turns to the right; when it hits the contact, it stops. Then it rotates 12 degrees and moves to the starting position. Finally, the scan begins, proceeding one degree at a time and allowing the lidar to take its measurement.

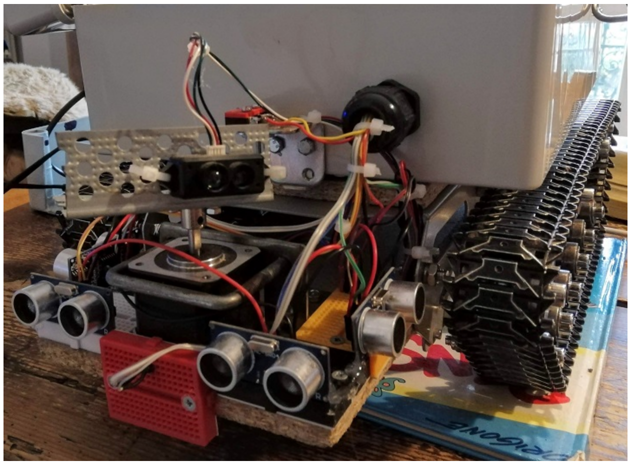
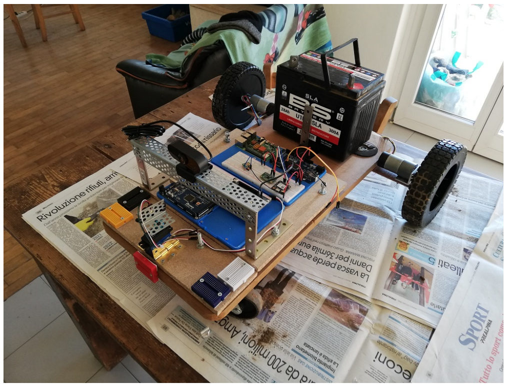
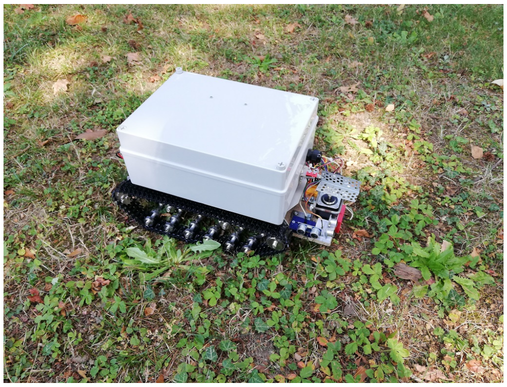


Figure 3.4: Stepper motor and ultrasonic sensors.

3.4 Mower Chassis

We try two different configurations for the project. The first was with two-wheel drive in the back. The second was a tank that was bought directly from a Chinese manufacturer.





Wheels This first prototype was built on a wooden surface, but its biggest problem was the difficulty in maintaining the trajectory on the lawn. Given the two small front wheels, every small imperfection made the trajectory deviate a lot. The important weight of the battery also did not improve the solution.

Tank This second prototype, on the other hand, thanks to the tracks, is much more able to keep the direction. The roughness of the ground affects the tank too but in a smaller way. This chassis, however, has less space for the battery which has been reduced in size by greatly reducing the autonomy. All components have been placed in a plastic box to protect them and to be easily transported.

3.5 Software subsystem

We can divide the software we are going to write into two parts:

- lower level for a microcontroller
- higher-level for a single-board computer

3.5.1 Microcontroller

Usually, the microcontroller in a robot interfaces with sensors and actuators. There are various solutions on the market, but the choice fell on `Arduino Mega`. Almost all the sensors on the market had libraries written for Arduino, which speeded up the software's writing.

Arduino board has a set of digital and analogue input/output (I/O) pins to work with the sensors. The boards can communicate with Raspberry Pi with a *Universal Serial Bus* (USB). The microcontrollers were programmed using the `C` and `C++` programming languages ([17], [22]).

3.5.2 Single-board computer

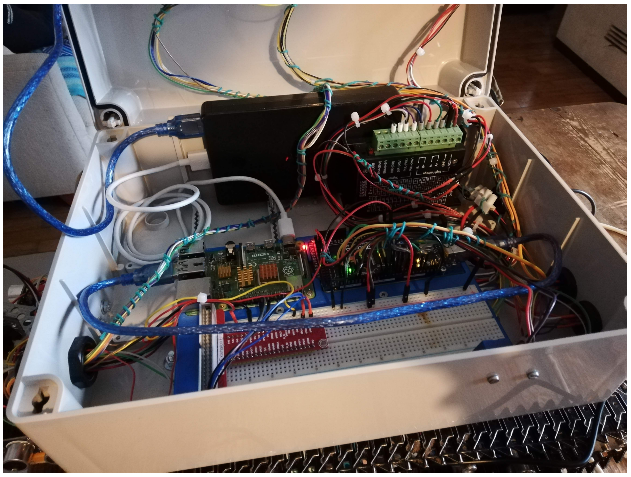


Figure 3.5: Raspberry Pi and Arduino

A single-board computer is a complete computer built on a single circuit board, with a microprocessor, memory, I/O and all the other features required for a functional computer. Single-board computers are commonly used as embedded computer controllers [26].

The computational part of this project is quite challenging. For example, the calculation of the next cell to visit and updating the map requires many calculations. For this reason, the `Raspberry Pi 4 Model B` was chosen.

`Raspberry Pi` has a `1.5 GHz 64-bit` quad-core `ARM Cortex-A72` processor, onboard `802.11ac Wi-Fi`, `Bluetooth 5`, full gigabit Ethernet, two `USB 2.0` ports, two `USB 3.0` ports, `4GB` of `RAM`, and a micro `HDMI`. The `Raspberry Pi 4 Model B` is powered via a USB-C port.

Chapter 4

Software Implementation

Contrary to common belief, the three most important topics in robotics are not Mechanics, Electronics and Software. Instead, they are Software, Software and Software![4].

In this chapter, we analyze the software used in the project. Almost all the code was written from scratch. This choice allowed us to understand every aspect that may affect the development of a robot. No specific frameworks or libraries on robotics were used.

Even if it has dramatically lengthened the development times, this choice allowed us to tailor the software for driving a lawnmower robot. In addition, we developed a simulator of the Lawnmower. A simulator gives us the environment to study the best strategy for the autonomous drive without relying on the real robot.

4.1 System Overview

Two blocks constitute the software system. One part is written for the Arduino microcontroller and mainly deals with sensors and motors. The second part runs on Raspberry PI, which manages all the complexity of localization and planning of objectives.

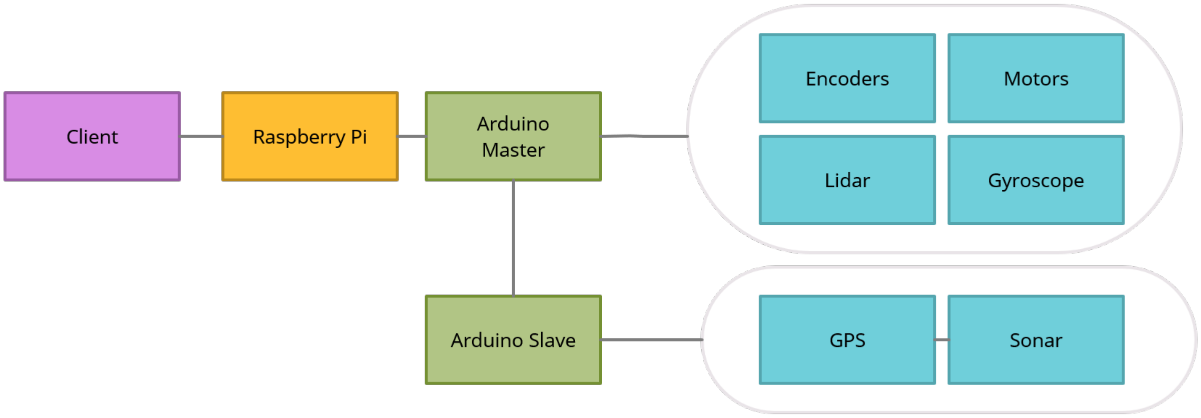


Figure 4.1

The main program was developed in python, as it is one of the most used programming languages on Raspberry pi.

Python is an interpreted high-level general-purpose programming language. Its design philosophy emphasizes code readability with its use of significant indentation. Its language constructs as well as its object-oriented approach aim to help programmers write clear, logical code for small and large-scale projects [30].

The program has about 15,000 lines of code in python. This count includes all the files in the project, including the tests. The classes that need to convert the calculated path into robot actions have the largest number of lines for both algorithms.

Source file	Total lines	Source code line	Source code lines %
dubins_path_planner.py	926	738	80%
epsilon_star_online_coverage	839	590	70%
app.py	836	504	60%
carrot_chasing.py	842	407	48%
message_plotter.py	535	406	76%
tilling	572	384	67%
....	....	...	...
Total	23047	15599	68%

The `Javascript` used for the graphical interface and the `C` used in `Arduino` are both 2000 lines of code.

4.2 App.py - Program Overview

The program is launched by running the *python3 app.py* command. Once the program has started, it waits for the user to start with a new mission or load an old map.

After the choice, the program enters the actual robot management. Figure 4.3 show the primary classes used in app.py.

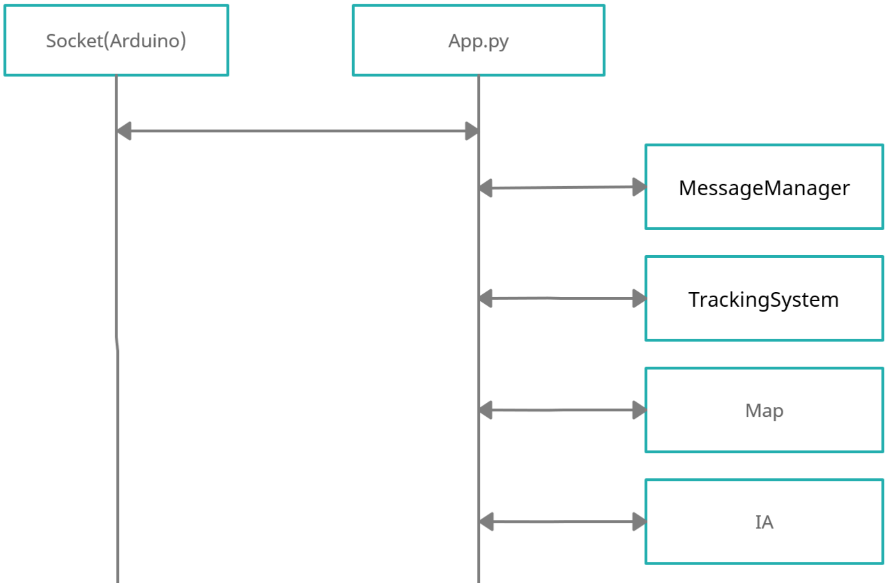


Figure 4.2

The architecture is based on the Observer Pattern [12]. The observer pattern is a software design pattern in which an object, named the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods. It is mainly used for implementing distributed event handling systems, in *event driven* software. In those systems, the subject is usually named a *stream of events* or *stream source of events*, while the observers are called *sinks of events*. The stream nomenclature alludes to a physical setup where the observers are physically separated and have no control over the emitted events from the subject/stream-source. This pattern then perfectly suits any process where data arrives from some input that is not available to the CPU at startup, but instead arrives *at random* (HTTP requests, GPIO data, user input from keyboard/mouse/..., distributed databases and blockchains, ...). [29]

With this pattern, every class interested in an event, such as an Arduino message, will subscribe to it. This pattern will ensure that the result code will be more readable. For example, a one-to-many dependency between objects should be defined without making the objects tightly coupled. Furthermore, It should be ensured that when one object changes state, an open-ended number of dependent objects are updated automatically. [29]

4.3 Configuration

A complex system has several values that can be changed. The program read from a file **applicationConfiguration.xml** all the information that is often modified. Among these choices, we can find:

- the tracking type
- the algorithm to use to explore the map
- the starting point
- algorithm parameters

Editing the XML file is more manageable: the code always remains the same, avoiding recompiling the whole solution. This choice is handy during the test on the field.

```
<?xml version=”1.0” encoding=”UTF-8”?>
<configuration>

        <tracking_type>
                <selected>1</selected>
                <options>
                <option id=”0”>CALCULATED</option>
                <option id=”1”>NAIVE</option>
                </options>
        </tracking_type>

        <tracking_type_sampling_rate>20</tracking_type_sampling_rate>

        <simulation_mode>
                <selected>1</selected>
                <options>
                <option id=”0”>True</option>
                <option id=”1”>False</option>
                </options>
        </simulation_mode>

         ...

</configuration>
```

4.4 Socket

The program receives inputs via socket. However, two simulators have been created to test the algorithms more quickly in addition to the Arduino. It was necessary to develop an interface to always use the same methods for any occurrence. In figure 4.3, it is possible to see all the methods of the interface.

```python
from abc import ABC, abstractmethod
from typing import List

class ISerial(ABC):

    def __init__(self):
        pass

    @abstractmethod
    def open_connection(self):
        pass

    @abstractmethod
    def is_serial_available(self) -> bool:
        pass

    @abstractmethod
    def write(self, msg: bytes):
        pass

    @abstractmethod
    def check_for_new_messages(self) -> bool:
        pass

    @abstractmethod
    def get_messages(self) -> List:
        pass
```

Particular attention has been paid to reassembling the messages that arrive. A message can arrive broken into several parts. The task of the class **MessageChunkHelper** is to recompose the messages arrived in several pieces. The program used a unique character to mark the end of the message: $. MessageChunkHelper can understand if a message arrived corrupt or if the rest of the message has yet to be decoded, thanks to this unique character.

4.5 Messages

Managing messages between Raspberry Pi and Arduino is not an easy task. Messages can get lost from both Arduino and Raspberry Pi. Messages may also not be handled readily and accumulate in a queue if Raspberry Pi has to compute other functions.

First implementation

Initially, the program written in python managed all the movements. The distance between the messages was just 300 milliseconds. The program had to be able to handle all of its computation in this range.

During these 300 milliseconds, the program had to manage:

- update the map
- check if the trajectory is respected
- send a message if necessary

Raspberry Pi had to handle all the movement. For example, if the robot had to go 2 meters and then turn left, Rasperry Pi started communicating with the message ”go forward”. In the meantime, as the robot advanced, Arduino sents report messages informing the program about the ticks the encoder had counted. By adding up all the various reports, the program knew if the two meters had been covered.

Unfortunately, the program could accumulate delays and then delay communication. The result is a list of messages accumulated on a queue. A robot not responsive to commands led to choosing another way to implement communication.

In this implementation, each command (forward, left, right, backward) created a sub-class from MovingHelper.Each sub-class was responsible for understanding if its goal had been achieved.

Second implementation

A piece of logic has been moved from Raspberry Pi to Arduino to avoid the problem of a robot not being responsive. The message in this second implementation has a parameter, for example, right-50. Once the message has been received, Arduino will autonomously rotate 50 degrees to the right.

For each message received, Arduino sends a NAK or an ACK message if it was able to decode it. Thus, each message has a unique ID. When Arduino responds to a command, this ID is reported. Thus, the program can understand if the command was executed.

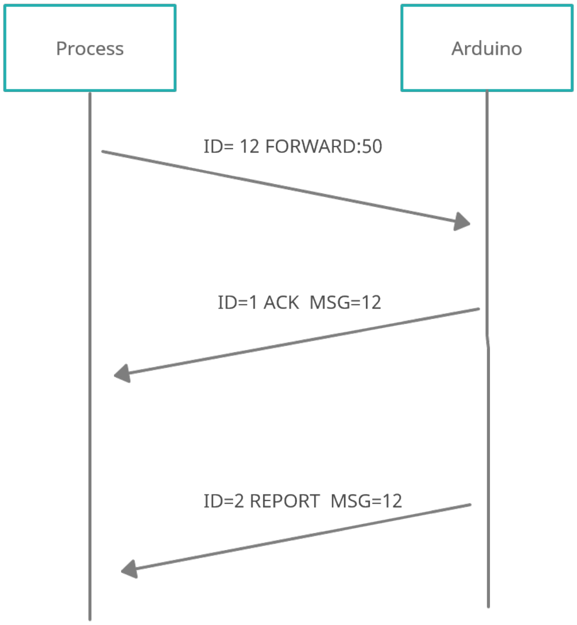


Figure 4.3

The MessageMenager class handle the communication. There are various scenarios to control:

- If all goes well, Arduino responds with an `ACK`, and once the command is completed, it sends the report. The report has the encoder ticks and the degrees read by the sensor.
- Arduino cannot decode the message. A `NAK` is sent, and the message is re-sent.
- A message is lost. It can be lost the report, the command or the ack. After waiting for a timeout to ensure the message is truly lost, the system goes into error mode.

A crucial method for handling robot motion is *can sent()*. The program can proceed with the planned strategy only if:

- the report expected has arrived from Arduino
- the system went into timeout error

The MessageMenager class is also responsible for sending commands to Arduino. When the message is sent, there is a well-defined time in which the system must react.

```
class MessageManager:

def __init__(self, my_socket_helper: SocketHelper, mover_status: LawnMowerStatus):

def can_sent(self, message: ArduinoMessage_Output)->bool:

def check_arduino_replay(self, messages: List[ArduinoMessage_Output]):

def __set_system_in_error(self):

def __check_for_report(self, messages: List[ArduinoMessage_Output])->bool:

def sent_message_to_arduino(self, message: HttpMessage)->bool:
```

The system uses the JSON format to make the communication more readable. Unfortunately, this choice adds a couple of milliseconds of delay. However, the most significant advantage is its great flexibility, which is very useful in this prototyping phase.

ArduinoMessageInput

The ArduinoMessageInput class models the commands that can be sent to Arduino. Messages are divided into two types: commands and setup.

- Commands are messages that indicate the robot's action, such as going forward for 30,000 ticks.
- The setup messages are used to set the values in Arduino. For example, the `RESET_THETA` message brings the compass back to the value 0.

An example of a message:

```
{
    "id":           "13",
    "key":          "COMMAND",
    "value":        "FORWARD",
    "parameter":    "3500"
}
```

The messages are encoded in JSON. The program then adds the `ID` field, which represents a unique number that identifies the message.

To manage the status of the message, the ArduinoMessageInput class has a variable of type ArduinoMessageInputStatusHelper: message_status. The purpose of the message_status variable is to save the states in which the message goes through.

```
class ArduinoMessageInputStatusHelper:

def __init__(self, msg_id: int, command: HttpCommand):
        self._id = msg_id
        self._command = command
        self._is_send: bool = False
        self._is_ack_arrived: bool = False
        self._is_nak_arrived: bool = False
        self._is_report_arrived: bool = False
        self._is_timeout_acknowledge: bool = False
        self._is_timeout_report: bool = False
        self._is_in_error: bool = False
        self._is_arduino_stuck: bool = False
```

The life cycle of the message is:

- Send
- `ACK`, `NAK` or lost
- Report
- Timeout

ArduinoMessageOutput

The ArduinoMessageOutput class models the various answers that can come from Arduino. There are five types of messages that Arduino can send.

- **ACK**: the message was successfully received and coded.
- **NAK**: the message was received but could not be coded.
- **REPORT**: the robot has finished carrying out the requested action. Arduino sends all the recorded data to the program.
- **ALIVE**: the robot is ready to execute another command.
- **SCAN**: the result of the lidar scan. This message contains 180 values.

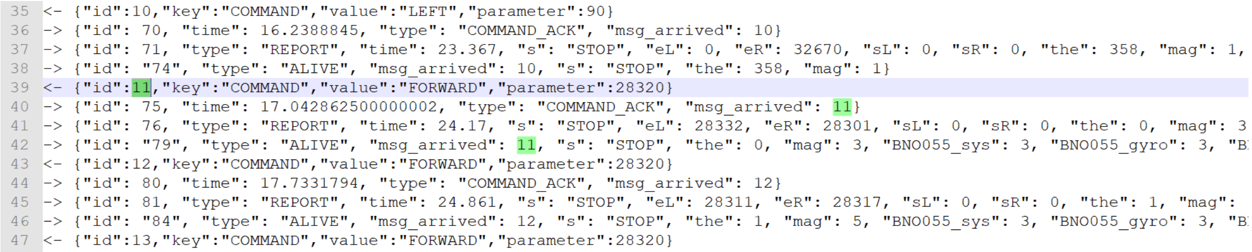
An example of report message::

```
{
        ”id”:    86,
        ”type”: ”REPORT”,
        ”time”:  25.502,
        ”s”:    ”STOP”,
        ”eL”:   28306,    ”eR”: 28332,
        ”sL”:   0,        ”sR”: 0,
        ”the”:  0,        ”mag”: 3,
        ”msg_arrived”: 13,
        ”res”: true,
        ”lat”: ”180.759934849”,
        ”lon”: ”181.759934849”,
        ”gps”: 182, ”sat”: 21,
        ”s_R”: −1, ”s_Fr”: −1, ”s_Fl”: −1,
        ”s_L”: −1, ”s_D”: −1
}
```

ArduinoMessageOutput is a Data Class. A data class refers to a class that contains only fields and crude methods for accessing them (getters and setters). These are simply containers for data used by other classes. These classes don't contain any additional functionality and can't independently operate on the data that they own. [20]

Messages.json

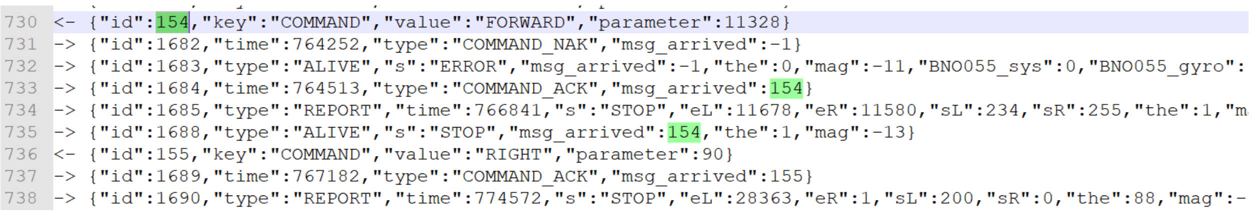
When the program starts, it creates the message.json file. In this file are saved all the messages exchanged between the program and the Arduino. This file is handy for understanding how the system reacted and finding out if there was a communication error.



```
<- {"id":10,"key":"COMMAND","value":"LEFT","parameter":90}
-> {"id": 70, "time": 16.2388845, "type": "COMMAND_ACK", "msg_arrived": 10}
-> {"id": 71, "type": "REPORT", "time": 23.367, "s": "STOP", "eL": 0, "eR": 32670, "sL": 0, "sR": 0, "the": 358, "mag": 1,
-> {"id": "74", "type": "ALIVE", "msg_arrived": 10, "s": "STOP", "the": 358, "mag": 1}
<- {"id":11,"key":"COMMAND","value":"FORWARD","parameter":28320}
-> {"id": 75, "time": 17.042862500000002, "type": "COMMAND_ACK", "msg_arrived": 11}
-> {"id": 76, "type": "REPORT", "time": 24.17, "s": "STOP", "eL": 28332, "eR": 28301, "sL": 0, "sR": 0, "the": 0, "mag": 3
-> {"id": "79", "type": "ALIVE", "msg_arrived": 11, "s": "STOP", "the": 0, "mag": 3, "BNO055_sys": 3, "BNO055_gyro": 3, "B
<- {"id":12,"key":"COMMAND","value":"FORWARD","parameter":28320}
-> {"id": 80, "time": 17.7331794, "type": "COMMAND_ACK", "msg_arrived": 12}
-> {"id": 81, "type": "REPORT", "time": 24.861, "s": "STOP", "eL": 28311, "eR": 28317, "sL": 0, "sR": 0, "the": 1, "mag":
-> {"id": "84", "type": "ALIVE", "msg_arrived": 12, "s": "STOP", "the": 1, "mag": 5, "BNO055_sys": 3, "BNO055_gyro": 3, "B
<- {"id":13,"key":"COMMAND","value":"FORWARD","parameter":28320}
```

Figure 4.4: This figure shows how the flow of the messages without error.

The program sends the command with `id = 11`. Arduino responds with an `ACK` reporting the `ID`. When the command has been executed, Arduino sends the `REPORT` message with all the collected data. Finally, Arduino sends the `ALIVE` command to communicate that it is available.



```
<- {"id":154,"key":"COMMAND","value":"FORWARD","parameter":11328}
-> {"id":1682,"time":764252,"type":"COMMAND_NAK","msg_arrived":-1}
-> {"id":1683,"type":"ALIVE","s":"ERROR","msg_arrived":-1,"the":0,"mag":-11,"BNO055_sys":0,"BNO055_gyro":
-> {"id":1684,"time":764513,"type":"COMMAND_ACK","msg_arrived":154}
-> {"id":1685,"type":"REPORT","time":766841,"s":"STOP","eL":11678,"eR":11580,"sL":234,"sR":255,"the":1,"m
-> {"id":1688,"type":"ALIVE","s":"STOP","msg_arrived":154,"the":1,"mag":-13}
<- {"id":155,"key":"COMMAND","value":"RIGHT","parameter":90}
-> {"id":1689,"time":767182,"type":"COMMAND_ACK","msg_arrived":155}
-> {"id":1690,"type":"REPORT","time":774572,"s":"STOP","eL":28363,"eR":1,"sL":200,"sR":0,"the":88,"mag":-
```

Figure 4.5: This figure shows how the flow of the messages with a NAK error.

The program sends the command with `id = 154`. Arduino responds with a `NAK`. Arduino sends a `NAK` message when it fails to decode the input. Then Arduino sends the `REPORT` message with the error field set to true. When the program receives a `NAK`, resent the message with the same `ID`. This time the message is decode correctly and the mission can go on.

4.6 Maps

The robot relies on different maps to make its decisions. Maps are matrices (`numpy`) and represent reality through a grid.

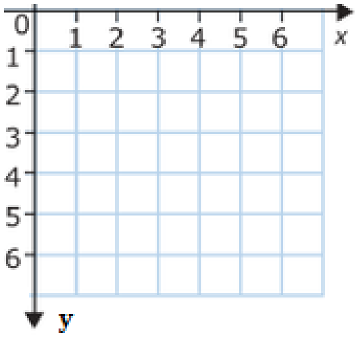


Figure 4.6: The origin of the axes is the top-left point in the map.

Obstacle Maps

This map is used to represent the environment where the robot acts. Therefore, it is the most important map. All the other maps derive from it and must be as precise as possible. Each px of the map represents a square in the real world. The default resolution is 10 cm, so a px in the map represents a 10x10cm square.

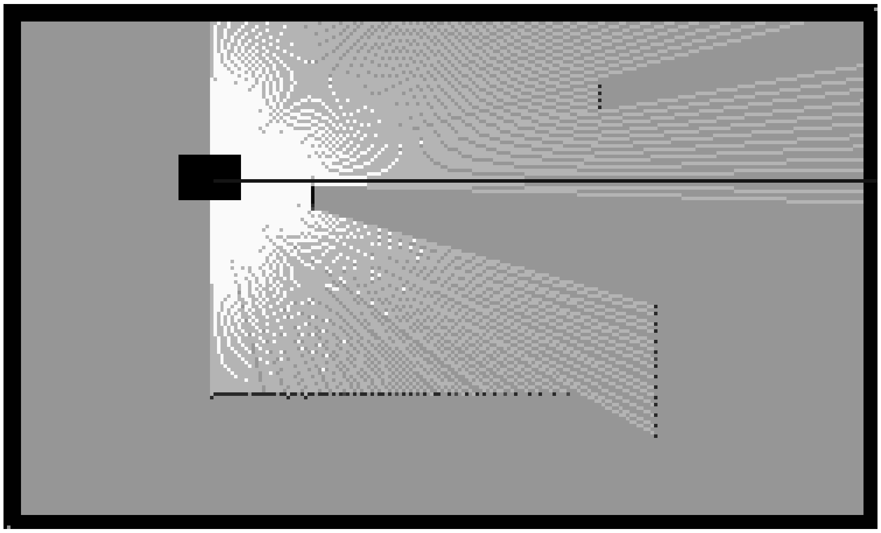


Figure 4.7: Example of an obstacle map. Light grey represents the free ground. The dark grey represents the still unknown part. Black, on the other hand, represents obstacles.

While each cell in the obstacle map can have one of 255 different values, the underlying structure that it uses is capable of representing only three. Specifically, each cell in this structure can be either free, occupied, or unknown. [13]

The program set the threshold to 120. If the cell value is higher than 120, the cell is free. If the value is lower than 120, the cell is occupied. The closer the value is to zero, the more confident that the cell represents an obstacle.

Map Updates When new data comes from the sensors, the program uses the Bayes formula to update the map. We want to find out the probability of the cell being occupied, given the new sensor reading s, and also knowing our prior probability of the cell being occupied. [16] The formula of Conditional Probability

$$P(H|s) = \frac{P(s|H)P(H)}{P(s|H)P(H) + P(s|\neg H)P(\neg H)} \tag{4.1}$$

substituting *Occupied* for H becomes

$$P(\texttt{Occupied}|s) = \frac{P(s|\texttt{Occupied})P(\texttt{Occupied})}{P(s|\texttt{Occupied})P(\texttt{Occupied}) + P(s|\texttt{Empty})P(\texttt{Empty})}$$

Notice that the probability $P(s|\texttt{Occupied})$ and $P(s|\texttt{Empty})$ are known from the sensor model, while the other are unconditional probabilities.

Lidar Image

The robot is equipped with a lidar mounted on a motor, capable of scanning the surrounding terrain. Lidar is a method for determining ranges (variable distance) by targeting an object with a laser and measuring the time for the reflected light to return to the receiver. It has terrestrial, airborne, and mobile applications. Lidar is an acronym of *light detection and ranging* [28]

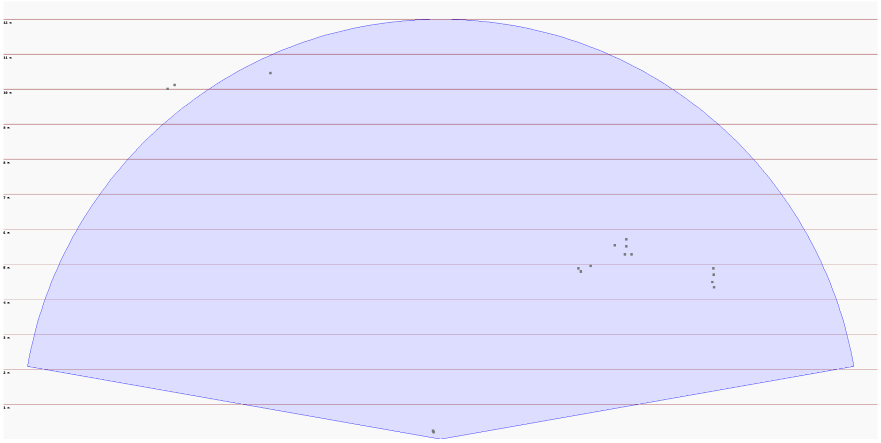


Figure 4.8: lidar scan example

Once the scan is complete, the program can create an image to observe the objects found. The data returning from Arduino are two arrays. The array index indicates the angle we are referring to. One of the two arrays indicates the distance. The other array indicates the strength of the measurement. These arrays are also used for updating the obstacle map.

The program uses Bresenham's line algorithm to update the map and create an image for the user interface. Bresenham's line algorithm is a line drawing algorithm that determines the points of an n-dimensional raster that should be selected in order to form a close approximation to a straight line between two points. It is commonly used to draw line primitives in a bitmap image (e.g. on a computer screen), as it uses only integer addition, subtraction and bit shifting, all of which are very cheap operations in standard computer architectures. It is an incremental error algorithm. It is one of the earliest algorithms developed in the field of computer graphics. [27]

Visited Maps

Visited Maps keep track of the path the robot has made. Each time the robot visits a cell, the cell counter is incremented. Thus, it is easy to see how many times a cell is visited based on the intensity of the white.

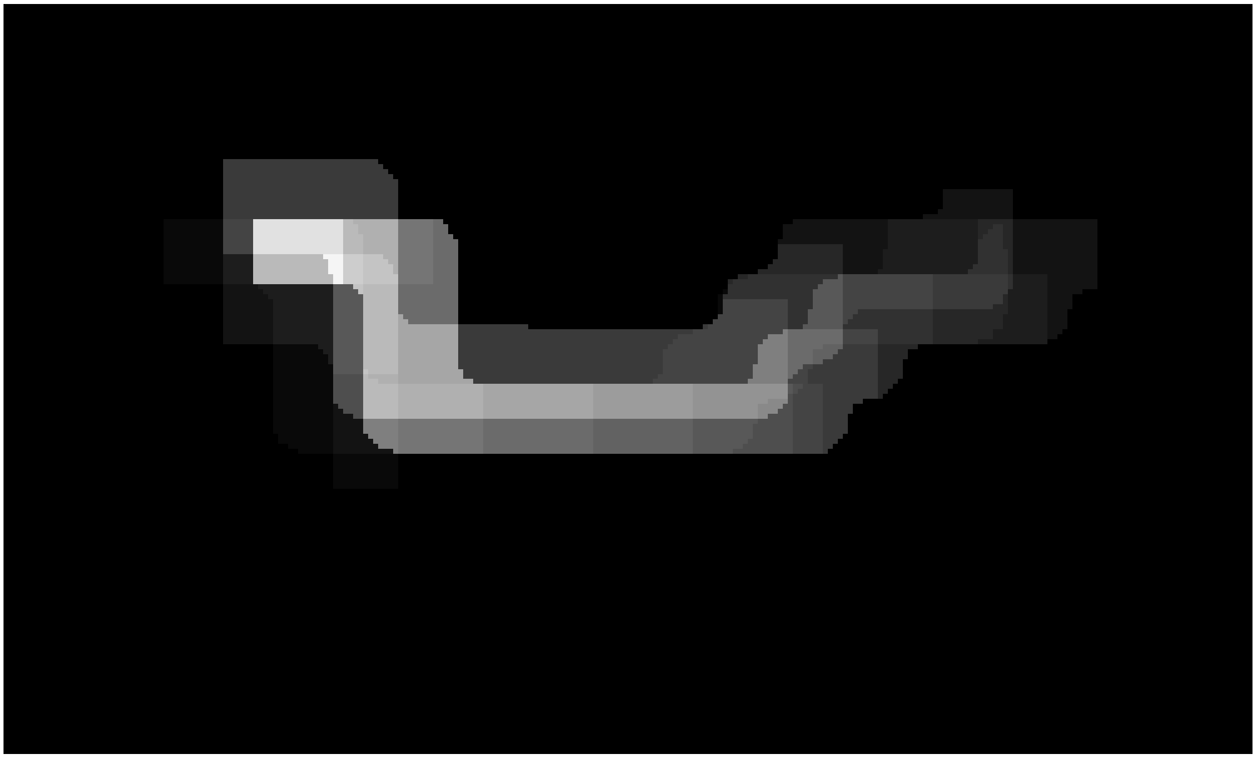


Figure 4.9: A visited map example. The robot started in the upper right corner and arrived in the upper left corner, avoiding an obstacle.

4.7 Tracking System

This class manages the position of the robot on the map. The robot's position is estimated on the information it receives from the encoders and the compass. There are currently two ways to keep track of the position. One is based solely on reading the encoders, the other based on the compass as well.

For each message that arrives from Arduino with new data, the tracking system class updates the position.

Managing forward movement is simple, thanks to the Pythagorean theorem. The program uses the formula shown in chapter two to mimic the rotation.

```
def __turn(self, command_sent, theta_msg_grad) -> Position:
    # ======================================== #
    if command_sent == HttpCommand.RIGHT:
    # ======================================== #
        theta_msg_rad = math.fabs(math.radians(theta_msg_grad))
        center_of_rotation = self.get_right_center_of_rotation()
        c = self.get_current_position().clone() - center_of_rotation
        return c.clone()
                .rotate(theta_msg_rad)
                .translate(center_of_rotation)
    # ======================================== #
    elif command_sent == HttpCommand.LEFT:
    # ======================================== #
        theta_msg_rad = math.fabs(math.radians(theta_msg_grad))
        center_of_rotation = self.get_left_center_of_rotation()
        c = self.get_current_position().clone() - center_of_rotation
        return c.clone()
                .rotate(-theta_msg_rad)
                .translate(center_of_rotation)
```

The tracking system works in cm. The position on the map instead is expressed in px by an approximation. Each time the program switches from the position in the map to position in the tracking system, it performs a conversion based on the resolution.

By adding GPS and an IA camera, the tracking system class can be improved in the future.

4.8 Web Server

The program lunch a new thread that runs a web server at startup. This server listens on port 8080. The server runs on the rover and waits for the user commands. Instead, the client lunch a simple HTML page(StartProject.html) that allows sending commands and reading the rover's status directly in the browser.

Therefore, in order to communicate, the client must be on the same LAN as the server.

```
msg_type: HttpMessageType = read_post_message_type(form)

if msg_type == HttpMessageType.STAR_PROJECT_MAP:
        msg_id = self.handle_star_new_project_by_map(form)

if msg_type == HttpMessageType.SEND_COMMAND:
        msg_id = self.handle_command_message(form)

if msg_type == HttpMessageType.STAR_PROJECT:
        msg_id = self.handle_star_new_project(form)

if msg_type == HttpMessageType.OPTIONS:
        msg_id = self.handle_options(form)
```

The server's job is to receive commands for:

- create a new map (set size and resolution)
- read the manual commands sent by the client (`STOP`, `GO`, `LEFT`, ...)
- provides the client with real-time information on what the robot is doing.

The messages sent from client to server and the replays are formatted in JSON.

Polling is a technique where the user check for fresh data over a given interval by periodically making API requests to a server. The polling interval is set to 2 seconds. The request for new information is made automatically on the `ShowMap.html` page. Once the new data arrives, the interface is updated.

4.9 Motion Control

Initially, the class `MovingHelper` in python dealt with managing the movement of the robot. For example, if the robot moved one meter forward, this class was responsible for sending the stop signal when the goal was reached.

With the arising of problems related to sending and receiving between Arduino and Raspberry Pi, the control logic was slowly moved to Arduino. Arduino needs to perform three actions in order to manage the robot movements:

1 - Read the message

Connection is via USB cable and Arduino communicates with the program via a socket. The first action Arduino does is to check is if there is a message. The program sends the command as JSON, so if a message is present, Arduino needs to decode it. Only if the decode works Arduino can execute the command.

Check is if there is a message:

```
// Input from RasperryPi
if (Serial.available() > 0) {
    count_stop_report_message = 0;
    bool isDeserializeJsonDone = DeserializeJson();
    if (isDeserializeJsonDone == false) {
        mowerStatus.ChangeState(ERROR);
        SetMotorSpeed(STOP);
        command_result = false;
    } else {
        SetVariablesFromMessage();
    }
    SentACK(isDeserializeJsonDone);
}
```

Arduino try to decode the message. In Arduino, it is not recommended to work with strings, so the input that arrives is copied character by character, as seen in the ”for loop”. This function returns True if the message is formatted correctly, False if it arrived corrupt.

```
bool DeserializeJson() {
    int availableBytes = Serial.available();
    if (availableBytes == 0) { return false; }
    for (int i = 0; i < availableBytes; i++) {
        serial_rasperrypi_input[i] = Serial.read();
    }
    return deserializeJson(msgInput, serial_rasperrypi_input);
}
```

2 - Set the variables

```
void SetVariablesFromMessage() {
    message_received_id = json_message_input["id"];
    const char* key     = json_message_input["key"];

    ArduinoMessageType msg_type = ConvertToArduinoMessageType(key);
    bool par_ok = json_message_input.containsKey("parameter");
    bool value_ok = json_message_input.containsKey("value");

    if (msg_type == COMMAND) {
        if (par_ok && value_ok) {
            const char* v = json_message_input["value"];
            Commands message_command = ConvertToCommand(v);
            if (message_command == FORWARD || ...){
                // ...
            }
            if (message_command == SCAN) {
                mowerStatus.ChangeState(STOP);
                is_scanner_message_arrived = true;
            }
        }
    }

    if (msg_type == SETUP) {
        // ....
    }
    json_message_input.clear();
}
```

This function set the variables according to the message that has arrived. The message can be:

- a command such as *turn right by 40*
- a setup message

3 - Execute the commands

This part of the program takes care of running the command:

```

// message has been sent
if (execute_command == true) {

// set new state
if (change_status == false) {
    SetMotorSpeed(mowerStatus.GetState());
    change_status = true;
}

// Enter in the ”turn mode”
if (mowerStatus.GetState() == LEFT || RIGHT) {
    ReadCompass();
    int destination = CalculateDestinationsTheta(...);
    MotorWheel motor = GetMotor(mowerStatus.GetState());
    command_result = StartTurn(motor, destination);
    mowerStatus.ChangeState(STOP);
    motor.SetSpeed(0);
    execute_command = false;
}

if (mowerStatus.GetState() == FORWARD ) {
    max_value_encoders = max(abs(...), abs(...));
    if (max_value_encoders >= encoder_step) {
        mowerStatus.ChangeState(STOP);
        LeftMotor.SetSpeed(0);
        RightMotor.SetSpeed(0);
        execute_command = false;
    }
}
```

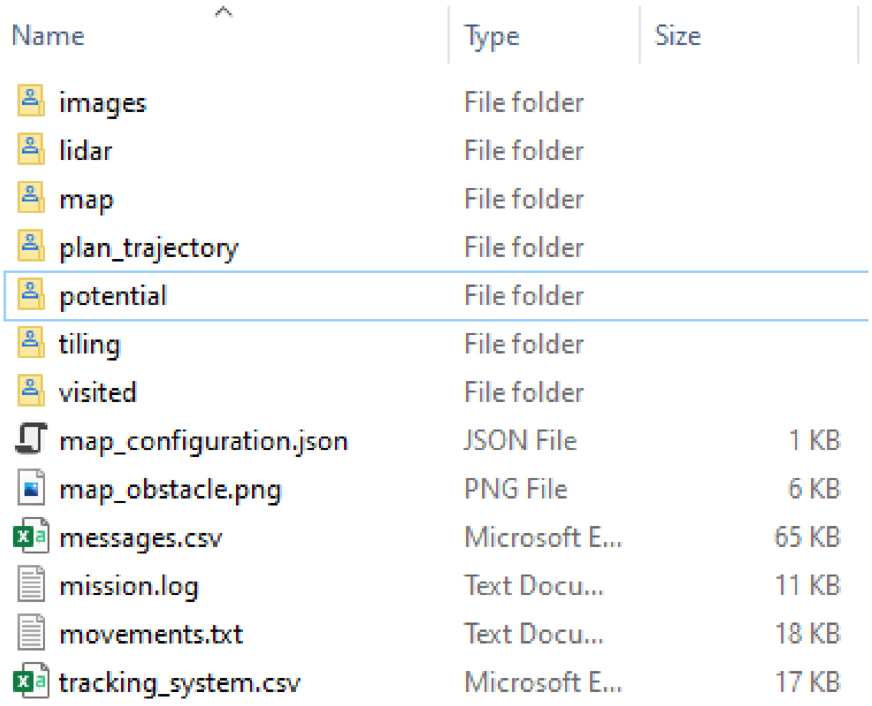
If the robot has to go on, the program checks that the encoder tick count exceeds the set threshold. The threshold was set by reading the value from the message. The value of the encoders is reset at each message and increase each time the motor rotates.

If the robot has to turn instead, the first action performed is calculating the target angle.Then the robot starts spinning. As it approaches the target, the speed decreases to reduce the error.

4.10 Missions

To simplify debugging and to be able to carry out multiple tests, it was decided to save each test in its own folder. A test is called in this case mission. In the mission folder you can therefore find:

- the configuration file
- the images of the map
- the images of the scans made with lidar
- the images taken with the camera
- the images of the potential map
- the images of the visited map
- the images of the plan trajectory map
- The log with errors (mission.log)
- The log file with almost all the decisions that the AI has made (moviments.txt)
- A file with all the messages exchanged



Name	Type	Size
images	File folder	
lidar	File folder	
map	File folder	
plan_trajectory	File folder	
potential	File folder	
tiling	File folder	
visited	File folder	
map_configuration.json	JSON File	1 KB
map_obstacle.png	PNG File	6 KB
messages.csv	Microsoft E...	65 KB
mission.log	Text Docu...	11 KB
movements.txt	Text Docu...	18 KB
tracking_system.csv	Microsoft E...	17 KB

Figure 4.10

4.11 AI

In this thesis two different strategies are tested. An interface was therefore created that allows you to easily change the main algorithm and thus test both solutions.

```

class AIGeneral(ABC):
”””
This class model the general inside the Robot,
which receives information and send out commands
”””

def __init__(self,obstacle_map,tracking_system,mower_status):
    self._obstacle_map: ObstacleMap = obstacle_map
    self._tracking_system: TrackingSystem = tracking_system
    self._mower_status: LawnMowerStatus = mower_status
    self._is_new_plan_trajectory_available: bool = False
    self._status = MowerStatus.IDLE
    self._maps: List[LastMapAvailability] = []
    # ....

def get_map(self, map_type: MapType) -> LastMapAvailability:
    for item in self._maps:
        if item.get_map_type() == map_type:
            return item
    raise Exception(”Sorry, MapType is not found”)

def get_next_command(self, last_report) -> HttpMessage:
    raise Exception(”Function to override”)
```

The AI uses all of the information available to choose the following command to execute. The output of the get_next_command method is the following command to execute.

4.11.1 Grid based coverage path planning

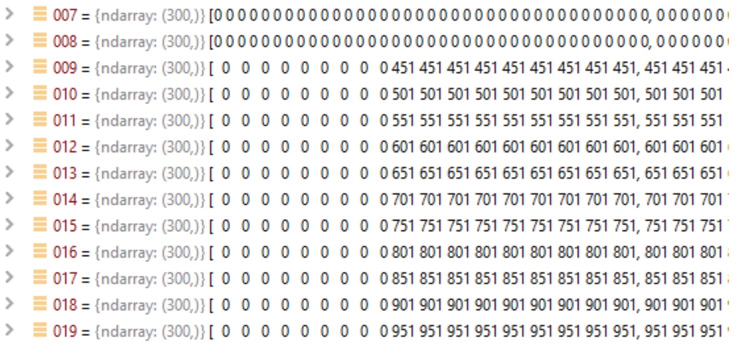
The algorithm that deals cover the area the map is divided into three parts.

1- Potential Maps

This map is for helping to plan the next goal. This map is created as a NumPy array matrix. The idea behind this map is to divide the cells into:

- cells not visited: which have a lower potential.
- cells already visited: which have a higher potential than cells not visited
- obstacles: which have zero potential and are, therefore, not reachable

To allow the map to be explored line by line, the potential is increased as the number of lines increases. The algorithm then looks for the lowest value and that will be the rover's next target.

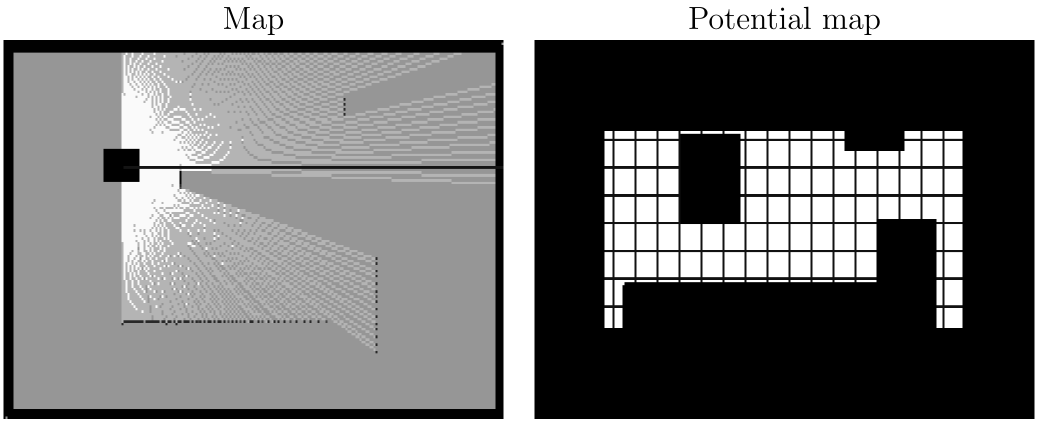


```
> 007 = {ndarray: (300,)} [0000000000000000000000000000000000000, 000000
> 008 = {ndarray: (300,)} [0000000000000000000000000000000000000, 000000
> 009 = {ndarray: (300,)} [ 0  0  0  0  0  0  0  0  0 451 451 451 451 451 451 451 451 451, 451 451 451
> 010 = {ndarray: (300,)} [ 0  0  0  0  0  0  0  0  0 501 501 501 501 501 501 501 501 501, 501 501 501
> 011 = {ndarray: (300,)} [ 0  0  0  0  0  0  0  0  0 551 551 551 551 551 551 551 551 551, 551 551 551
> 012 = {ndarray: (300,)} [ 0  0  0  0  0  0  0  0  0 601 601 601 601 601 601 601 601 601, 601 601 601
> 013 = {ndarray: (300,)} [ 0  0  0  0  0  0  0  0  0 651 651 651 651 651 651 651 651 651, 651 651 651
> 014 = {ndarray: (300,)} [ 0  0  0  0  0  0  0  0  0 701 701 701 701 701 701 701 701 701, 701 701 701
> 015 = {ndarray: (300,)} [ 0  0  0  0  0  0  0  0  0 751 751 751 751 751 751 751 751 751, 751 751 751
> 016 = {ndarray: (300,)} [ 0  0  0  0  0  0  0  0  0 801 801 801 801 801 801 801 801 801, 801 801 801
> 017 = {ndarray: (300,)} [ 0  0  0  0  0  0  0  0  0 851 851 851 851 851 851 851 851 851, 851 851 851
> 018 = {ndarray: (300,)} [ 0  0  0  0  0  0  0  0  0 901 901 901 901 901 901 901 901 901, 901 901 901
> 019 = {ndarray: (300,)} [ 0  0  0  0  0  0  0  0  0 951 951 951 951 951 951 951 951 951, 951 951 951
```

Figure 4.11

This map is created from the obstacle map. All obstacles are reported. In addition, an extra border is added around obstacles. This extra edge is added to increase safety.

In order to create trajectories where the number of curves is minimized, an important step is to divide the potential map as if it were a chessboard.



Map

Potential map

The map of the visited cells can be passed as an argument in the constructor of the class. The potential of a cell is increased every time the robot visits the cell. The robot is encouraged to visit still unexposed cells.

2- Plan Trajectory

The robot's goal is to explore the whole map. The robot starts in the lower-left corner then tries to reach the lower-right corner. Once the goal is achieved, a new goal is calculated. The robot will continue to go from right to left.

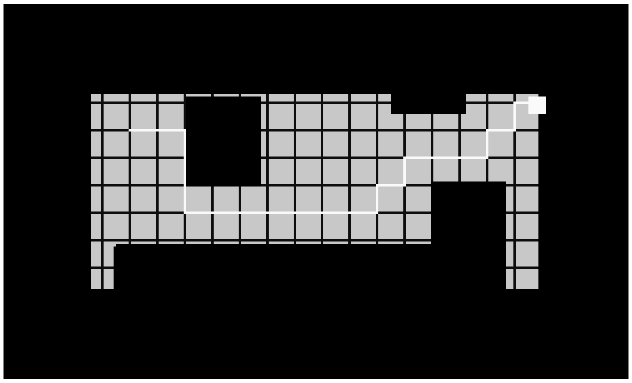


Figure 4.12

In these constant movements, the robot is encouraged to travel through unexposed cells, which have the lowest potential. The a-star algorithm is used to calculate the route between the starting point and the endpoint. To convert the path into robot movements, the CarrotCasing class was written.

3- Calculation of Trajectory: CarrotCasing

The robot needs about 11 cm to make a curve. We must therefore calculate this space when we carry out the conversion.

In the image, you can see the points that the robot must reach. It can be seen that the robot stops 11 cm earlier than the path near a curve. The red dot after the curve indicates where the robot should be after turning.

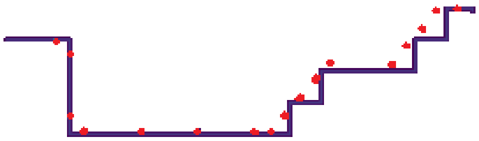


Figure 4.13

To avoid deviations due to uneven ground after many tests it was decided to advance a maximum of 40 cm at a time. After translating the path into robot movements, the movements that exceed the 40 cm threshold are divided. The plan trajectory map is made only for the user to see on the screen the next moves of the robot and see the entire path towards the next goal.

$\epsilon\star$: An Online Coverage Path Planning Algorithm

One of the implementation used is based on the algorithm $\epsilon^\star$ [21]. In this section we go through some of the implementation details and some of the choice made in the selection of the best path.

Briefly the algorithm is based on the **Multiscale Adaptive Potential Surfaces** (**MAPS**) model, which is a hierarchically data structures, which is dynamically updated based on sensor information . It starts from the construction of a so called ϵ-tilling $\mathcal{T}_\epsilon$ on a map $\mathcal{M}$, i.e. a cover of the map with square cell of side equal to ϵ with a value indicating if the cell is visited, empty or contains an obstacle

$$\mathcal{M} \subseteq \bigcup_{i=0}^{N_\epsilon} \mathcal{U}_\epsilon^i := \mathcal{T}_\epsilon, \quad \mathcal{U}_\epsilon \in \{x_{visited}, x_{empty}, x_{obstacle}\}$$

along with a potential B that is used to give the hint of the trajectory to be followed and that is dynamically updated with the information about the environment, i.e. if a new obstacle is discovered by scanning the area in front of the mower: for example $B(x_{visited}) = 0$ and $B(x_{obstacle}) < 0$.

In the default run, the algorithm compute a ordered sequence of waypoints **wp** to be visited: these follow the shape of the potential, i.e.

$$\mathbf{wb} = \{w_1, ..., w_N\}, \qquad B(w_{j+1}) \leq B(w_j).$$

In case the list of waypoints is empty, recursively a list of tilling is generated where the length of the square (i.e. ϵ) double at every level and the value of every cell is the average of the corresponding cells in the lower level. At this point a new cell $\mathcal{U}_{2^n \cdot \epsilon}$ with positive potential is search. If one is found, let say U, then the mower should move to one of the empty cell corresponding to U, if no such square exists, the next level is calculated and so on. In the levels are all consumed, without finding a cell with positive epsilon, the map has been completly explored (refer to [21] for better and deeper explanation).

The class that implement this algorithm is `EpsilonStarOnlineCoverage`.

During the initialization the main parameters that have to be specified are the value of ϵ (the number of map grid to be merged together), the type of potential (grow/decrease vertically/horizontally) and size of the forbidden area, i.e. cells neighbor of an obstacle that are also marked as inaccessible (we follow the original paper, assigning $B = -2$ for the obstacle and $B = -1$ for forbidden squares).

Typically the grid of the *Obstacle map* maintained during a mission has a finer resolution then the smallest tilling, so that the first tilling is generated by making a coarser grid: in this phase if any of the grid cells contains an obstacle the corresponding tilling cell will be marked as $x_{obstacle}$, and similarly for visited and forbidden cells.
The second step is to construct the potential: according to the passed parameter iterating through the columns (or rows) from bottom (or from the top) the value of the potential is fixed taking into account the nature of the corresponding tilling cell.

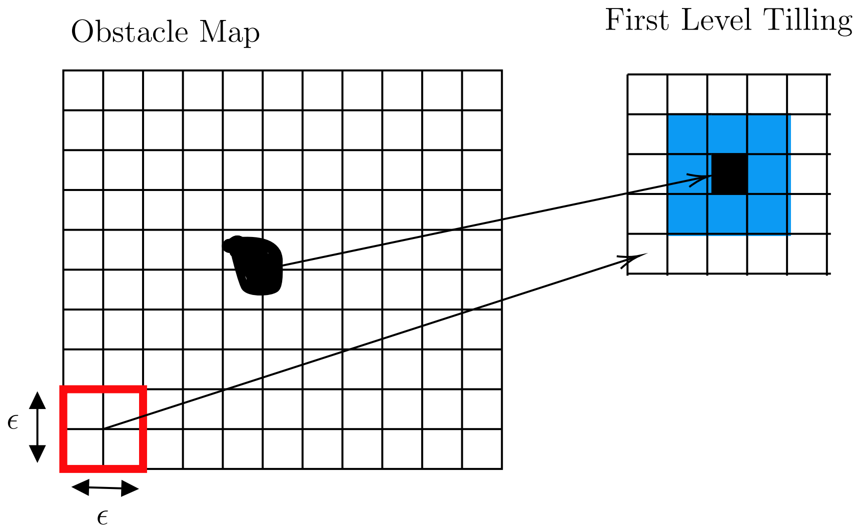


Figure 4.14: Initialization of the tiling from the obstacle map

At this point the AI engine is properly initialized and the calculation of a list of waypoints can be requested.
The search of new waypoint differ from the original paper in small details that we present here, mostly to avoid small zigzag: this might produce a side effect, i.e. not all squares to be actually visited.

Calculation of Waypoints

Let's start at square $s_{i,j}$:

1. identify the empty neighbors of the current square: these can be parametrize allowing only *North*, *South*, *East* and *West* or also *North-East*, *North-West etc.*.
   In case no such square exists, then the neighborhood of the neighborhood are considered, discarding squares that have to got though inaccessible regions. Let call $\mathcal{N}_{i,j}$ these available squares. If $\mathcal{N}_{i,j}$ is empty alternative strategy are used to escape this (possible) *local minimum*.
2. From $\mathcal{N}_{i,j}$ create the set with cells with maximum potential, $\mathcal{N}^B_{i,j}$
3. The next waypoint is selected as the element that in $\mathcal{N}^B_{i,j}$ maximize a special select function typically dependent on the directions of the mower in the different squares.

4. Add the waypoints to **wp**:
5. Perform various checks
    - check if a scan is needed, i.e. the cells around the last waypoint are empty or unknown
    - check if a shortcut is possible, if yes expand it (see below)
    - check the presence of zigzags, if yes take action (see below).

If it is possible to keep going go to 1. otherwise break.

Example 4.11.1. Selection of next waypoint Consider that the neighbors of a squares are all the 8 squares around it: one of the implementation for the selection of the next waypoint among candidates with equal potential is as follow: Consider the current direction $\theta_{i,j}$ and the direction to reach the next square $\theta_{k,j}$, the next waypoint is

$$\theta := \arg\max_{\phi \in \mathcal{N}^B} f(cos(\theta_{i,j} - \phi))$$

$$f_1(x) = \begin{cases} 2 - x & \text{if } 0 \leq x \leq 1 \\ 1 + x & \text{otherwise} \end{cases}$$

In this case it is preferred to go straight perpendicular to the current direction instead 45 degrees.

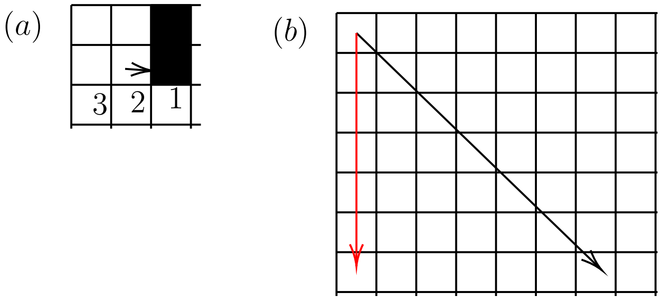


Figure 4.15: (a) local search for waypoints: squares 1, 2 and 3 have all the same potential. (b) possible global strategies

For example in **Figure 4.15** supposed the points marked as 1, 2, and 3 have all the same potential, and the potential grows with the y axis: Using as metric the function f_1 the list of the waypoints will be the red arrow, using a function like $f_2(x) = x$ the list of waypoints will produce the black line with slope of 45 degrees.
Consider the starting point in (b) is the area behind a wall the run parallel to the x axis and another wall goes straight along the y axis downwards. Using the function f_2 will result in going through the free area diagonally while the function f_1 will impose to reach the bottom of the square and then proceeds *normally* forwards and backwards.

Use shortcuts

As mention in the introduction, the grid of the obstacle map is finer then the one in of the first tilling, which might lead to straight walls not perfectly aligned to the direction of the grid: this will create in the potential/tilling broken lines. The same is true in case the mower cross a neighbor cell of the calculated waypoints (due to not perfectly aligned driving angle) or there is some error in the measurements: in all these cases the irregularity created might make the trajectory much more complex to follow.
At price to miss to cover some small areas, when a new waypoint is added to **wp** a backward check is done to identifies such cases and perform a shortcut. The algorithm is very simple and straightforward: check the direction of the previous waypoints and count the change of direction as well as the length of straight line parallel to the current direction, this is enough to identify such cases (see **Figure 4.16**):

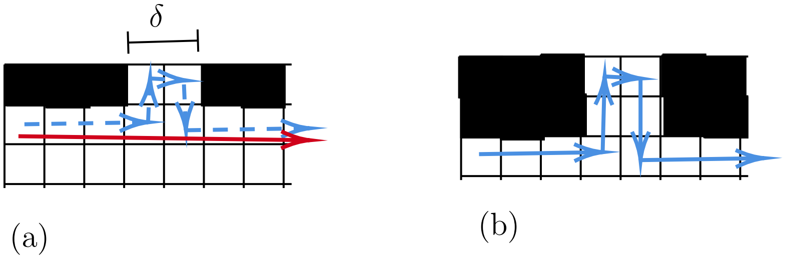


Figure 4.16: (a) The trajectory in blue is replace with the one in red. (b) The trajectory cannot be transformed by a shortcut.

If the length δ is short enough, and if the deviation from a straight line is small enough (one square in (a) vs. 2 squares in (b)), then the waypoints are merged to produced the shortcut drawn in red in (a).

Deal with Zigzags

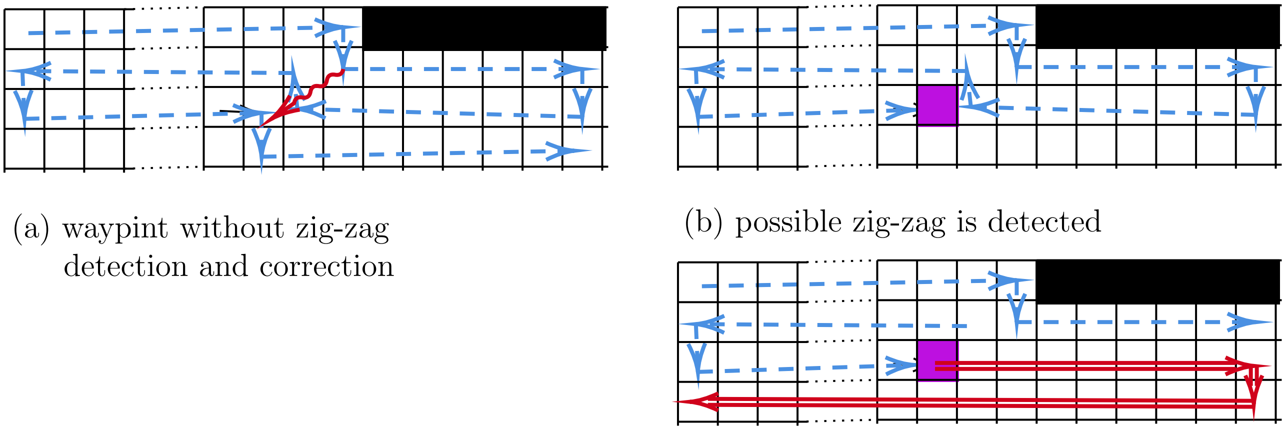
Consider the shape in **Figure 4.17**
In case no action is taken the *shape* of the obstacle will be *translate* across the area while more and more trajectories are calculated (red line in (a)). This might be not the desired path, since the insertion of not necessary turns might increase the noise and the errors.
This optimization works only in case of standard potential is used, i.e. vertical or horizontal.
During the calculation of the waypoints **Step 5** a possible zigzag needs to be detected.

Detection

Let consider we have just added the waypoint w_1 and $\theta_{w_1} \neq \theta_{w_0}$ then

1. prolong the trajectory from w_0 in direction θ_0 and check no obstacle will be met in n-cells, where n is a parameter from the user. If all condition are met, go to 2., otherwise break.
2. calculate the theoretical potential in this direction, i.e. $B(w_0+t\cdot\theta_0)$. If $B(w_0) \geq B(w_0)$ no real improvement is found, then break

3. otherwise break the generation of waypoints: in this case we have reached the pink square in subplot (b).



(a) waypint without zig-zag detection and correction

(b) possible zig-zag is detected

(c) the new trajectory goes through already visited cells.

Figure 4.17: Correction of ZigZags: instead of repeating the shape of an obsatcles, the waypoints try to prefer straight line, even though they go through visited

The next time the waypoints are requested, the algorithm will try to fix it.

Fix zigzag

Let consider we are at position p in square w_0:

1. check that the neighbor $\mathcal{N}_{w_0}$ of w_0 are not all visited
2. check that the mower is not driving *as a salmon*, i.e. is not going in the direction of growing potential.
3. repeat the zigzag detection: during the previous consumption of the strategy the arrival point might be different from the one calculated in the previous waypoints.

   3.a prolong the trajectory from w_0 in direction θ_0 and check no obstacle will be met in n-cells, where n is a parameter from the user.

   3.b calculate the theoretical potential in this direction, i.e. $B_0 = maxB(w_0 + t \cdot \theta_0)$. If $B(w_0) \geq B_0$ no real improvement is found, then break

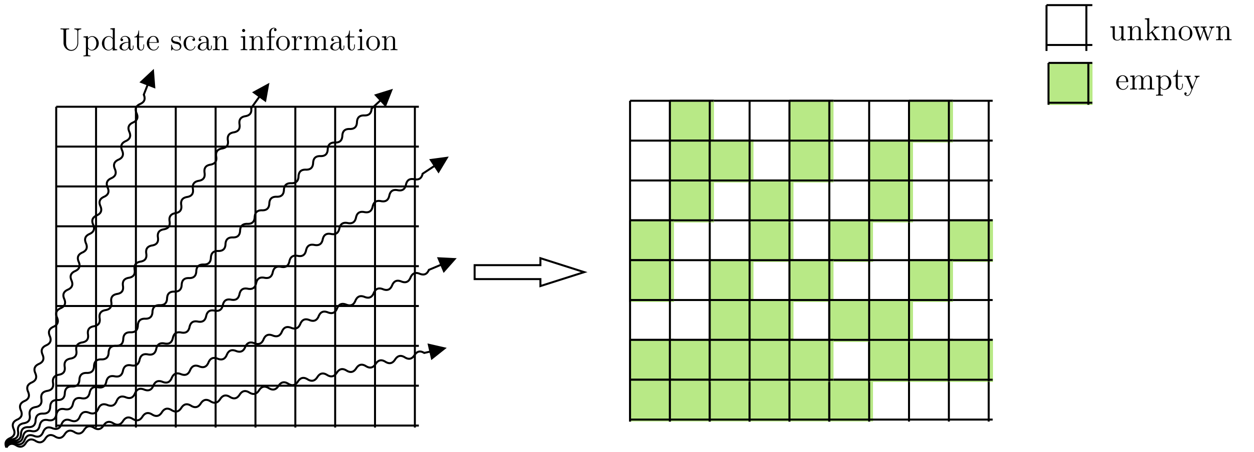
4. check that the parallel above and below the current direction do not contains any obstacle
5. If all the conditions above are met, we temporary overwrite the values in the tiling and potential with the theoretical one and go on with the normal calculation of waypoints.

We note that the information about visited cells is not lost by overwriting the values in the tilling, in fact the obstacle map is not modified. The data structure used where the

Tiling logic is implemented (*HierarchicalMultiScaleTiling*) is regularly recreated from the Obstacle map.

Check if a scan is needed

Working with the tilings $\mathcal{T}_\epsilon$, the underline obstacle map is only indirect used: each $\mathcal{U}$ in $\mathcal{T}$ select the prominent characteristic from the obstacle map grid in the corresponding cells ($\mathcal{O}_j^{\mathcal{U}_\epsilon}$)and use it directly, in particular for the presence of obstacles or to mark a cell as visited. When we consider if a $\mathcal{U}$ is empty or unknown, i.e. if we need to issue a scan command we need to look directly into the obstacle map, taking into account how the information derived from a scan are saved into this map.



(a) set the cell empty alng the direction of the laser

(a) result of a scan in an area without obstacle

Figure 4.18: Update known (empty) and unknown cells in the Obstacle Map after a scan: not all cells are labeled.

When a scan is performed the cells along the bean are marked empty, while the one with obstacle are marked this way. Let us consider a square $\mathcal{U}$ without obstacle: clearly it is improbable that all cells $\mathcal{O}_j^{\mathcal{U}_\epsilon}$ are marked empty: so when check for the next waypoint if the ratio of unknown cells and the empty one exceeds a threshold (in the experiments it is 0.4): if this is the case, the construction of the waypoints list is interrupted and consequently all waypoints are consumed, a *scan* command is going to be requested.

Escape From Local Minima

In case no way-points is found, the algorithm try two different strategy.
First try to detect if it has encountered a cul-the-sack: in this case it tries to go backwards till a new way point is possible.
In case the the previous solution is unacceptable (i.e. it needs to go backwards for a very large line, it is not safe, etc.) then (as presented in [21]) higher tillings are created and examined to identified regions not yet explored. To reach such spots we employed the **A*** search algorithm.

Typically when searching for unexplored cells, more candidate are available: to select one, these regions are sorted according to the distance needed to reach it and the number of not visited cells in corresponding first layer. In case less then 3 cells in the first layer are present, then this position is discarded from the set op possible endpoints. The strategy might also go through unknown regions, and in these regions a scan is performed before complete the task, if possible, otherwise a new strategy is calculated.

4.11.2 Calculation of Trajectory: Connect Middle Points

Once the waypoints **wp** have been identified, the actual trajectory that the mower should follows is calculated:
This algorithm take as input a list of waypoints **wp** as well as the current direction of the mower, θ. Let index the waypoints relative to the current waypoint under consideration: p is the current position and θ its direction, while w_j is the j-the waypoint from w_0:

1. calculate the middle point of w_1 in real units: $p_j = (x_{p_1}, y_{p_1})$
2. extract the angle that the mower must have when transiting from p to p_i, let call it θ_i
3. get the intersection point q_0 where the mower will exit the cell corresponding to p in direction θ_1 and q_1 for the next one.
4. Calculate the Dubin paths[1]

$$t_1 = \mathcal{D}((x_p, y_p, \theta) \mapsto (x_{q_1}, y_{q_1}, \theta_1))$$

   and

$$t_2 = \mathcal{D}((x_p, y_p, \theta_0) \mapsto (x_{q_0}, y_{q_0}, \theta_1)) + \mathcal{D}((x_{q_0}, y_{q_0}, \theta_1) \mapsto (x_{q_1}, y_{q_1}, \theta_1))$$

   and select the shortest trajectory.
5. set $p = x_{q_1}$ and $\theta = \theta_1$ go to 1. starting from the next waypoint till none is left.

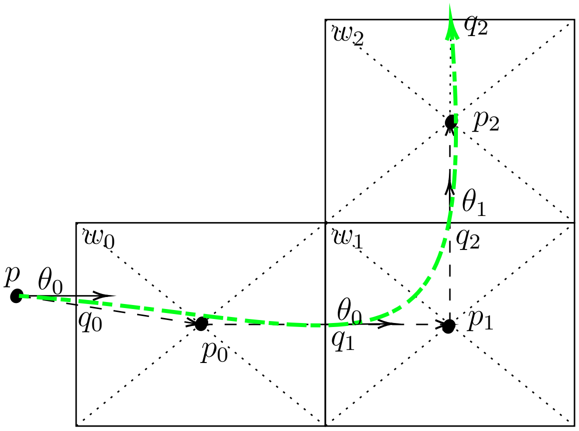


Figure 4.19: calculation of trajectory from waypoint using middlepoints: after identifying the middle points of the waypoints p_j, the direction needed to connect two consecutive squares (θ_j) and the intersection of the segment connecting the middle points (q_j), appropriate dubin are calculated: the one with smallest length is selected.

[1] $\mathcal{D}(A \mapsto B)$ is the Dubin path from A to B.

When using this algorithm to produce the trajectory it is important that ϵ is big enough so that the mower width is smaller then the size of a tilling.

4.12 Arduino

Usually, microcontrollers are used to read data from the sensors and to control the various actuators. Two mega Arduino are used in this project. The choice to use two Arduino is dictated by the fact that sonars are sensors that take a long time. While the Arduino is reading the sonar result, the system is not responsive.

The solution was to move the sonars to an Arduino slave which communicates with the Arduino master whenever it has updated data.

Master

This Arduino does practically all the work required to be able to command a rover. Among his tasks, we find:

- Read the commands that come from RasperryPi
- Read the sensor values
- Go forwards and backwards or turn according to the command
- Balance the speed to try to go straight
- Communicate with RasperryPi and send the collected data.

Paragraph 4.9 introduced how Arduino manages movements. This paragraph analyzes the `StartScan()` and `BalanceSpeed()` functions

StartScan()

To scan the area in front of the rover, Arduino uses the Lidar and the stepper motor.

```
void StartScan(int angle)
{
 while (count_angle < servo_max_angle) {
  StepperMakeOneDegree(LEFT);
  if (count_step != 0) {
        GetLidarDistance();
        lidarscan_distance[count_angle] = lidar_distance;
        lidarscan_strength[count_angle] = lidar_strength;
        count_angle++;
        count_step--;
        } else {
         count_step = skip_step_for_error_scale;
        }
 }
}
```

The motor initially rotates the lidar until the switch. The switch indicates when the motor reaches a know position. Then, the motor moves about 19 degrees to the initial position. Then the scan starts. The stepper motor allows making a rotation of 1 degree. For each degree, Arduino saves the measurement value into an array.

BalanceSpeed()

The robot is not always able to keep the trajectory due to the unevenness of the ground. The BalanceSpeed() function uses the feedback from the two motor encoders to decrease the speed of the faster motor. These adjustments are in the order of milliseconds.

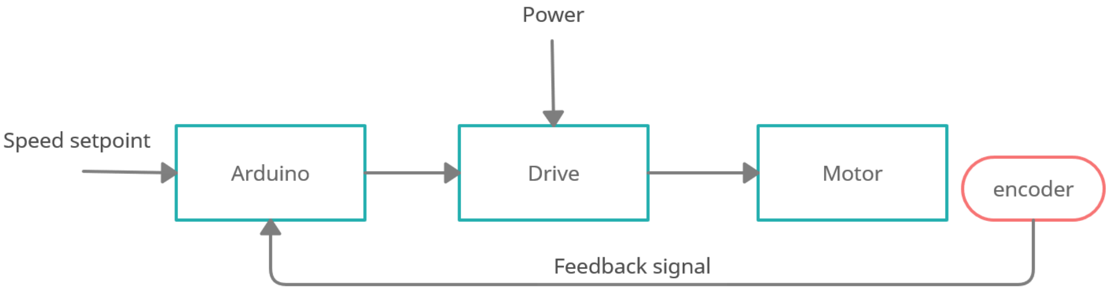


Figure 4.20

The first version of this function destroyed the joint between the wheel and the motor. Unfortunately, a bug in the software that manages mechanical components can be costly.

```
// # Right engine faster
if (motor_right_faster == 1) {

 //# Try to lower its speed
 if (motorRight_speed - incremental_value > min_speed)
 {
    new_motorRight_speed = (int)(motorRight_speed - incremental_value);
 }
 else
 {
   // # If I am already at max I raise the speed of the other motor.
   if ((motorLeft_speed + incremental_value) < max_speed) {
     new_motorLeft_speed = (int)(motorLeft_speed + incremental_value);
   }
   else
   {
     new_motorLeft_speed = max_speed;
   }
...
```

Slave

This second Arduino is needed to support the Arduino master. This Arduino reads the data of the fivers sonar and GPS. Once it has collected the data, it sends it to the master Arduino.

The tasks of the second Arduino are simple. The second Arduino was necessary because the reading from the sonar sensors took a long time and would have slowed down the main program too much.

```
void loop()
{
 is_message_to_send = true;

 //# GPS
 ResetGPS();
 ReadGPS();

 if (new_data_GPS) {
  GetGPValues();
 }

 //# Sonar
 ResetSonar();
 ReadSonarDistances();

 //# Serial
 if (is_message_to_send == true) {
  SentReportArduinoMaster();
  is_message_to_send = false;
 }
 delay(100);
}
```

4.13 User Interface

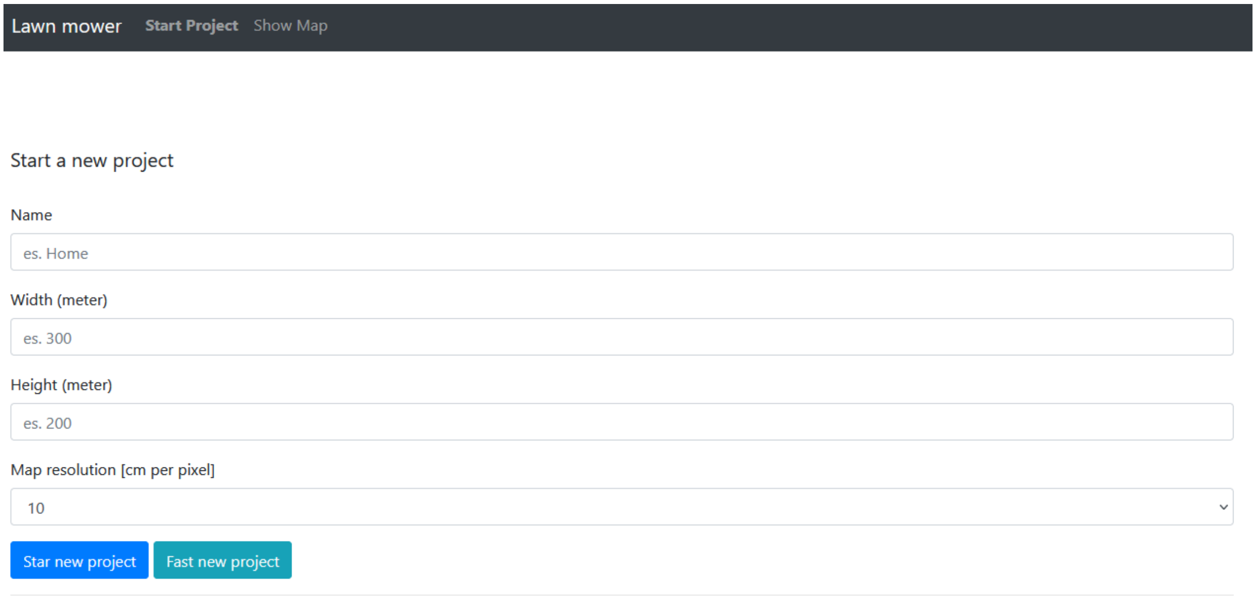
In order to allow the dialogue between the application running in the robot and the user, two simple web pages have been created. These allow you to start the mission and check its progress.

New Map

This page offers the user three choices:

- Create a new map
- Load an old map
- Upload a custom map

If you want to create a new map, you can specify both the length and the height. Another customizable aspect is the resolution, by default set at 10cm = 1px



Start a new project

Name
es. Home

Width (meter)
es. 300

Height (meter)
es. 200

Map resolution [cm per pixel]
10

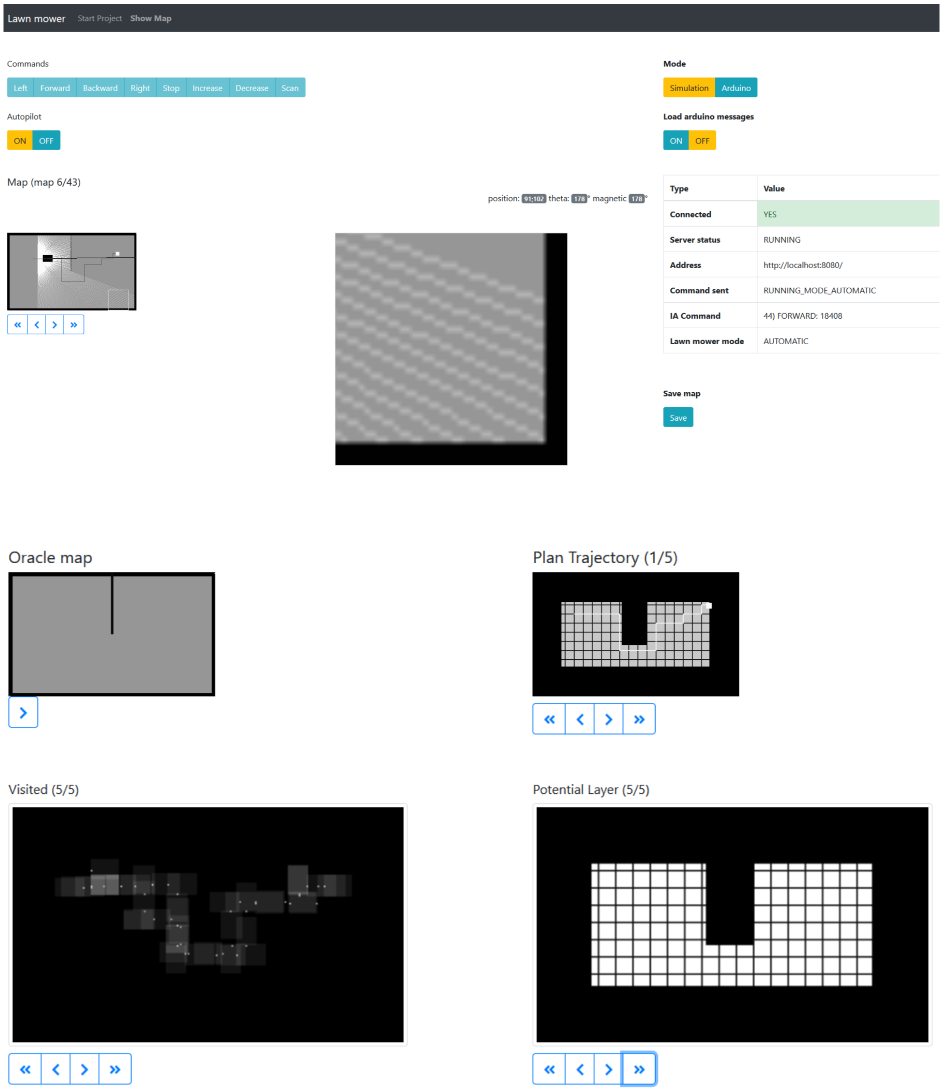
Figure 4.21: Home page, the user can choose the mission to start.

Dashboard

This page allows to control the robot and shows all the actions made by the robot. There are several maps: obstacles map, trajectory map, for the next goal, visited map potential map

The robot can run manual and automatic. In the UI, the buttons allow driving the robot manually.

On the page, there is a polling timer set to 1s. When this timer triggers, a request to the webserver is made. The web server responds with all updated information in a JSON format. This information contains, for example: the last command sent, plan strategy, update map, ...



Type	Value
Connected	YES
Server status	RUNNING
Address	http://localhost:8080/
Command sent	RUNNING_MODE_AUTOMATIC
IA Command	44) FORWARD: 18408
Lawn mower mode	AUTOMATIC

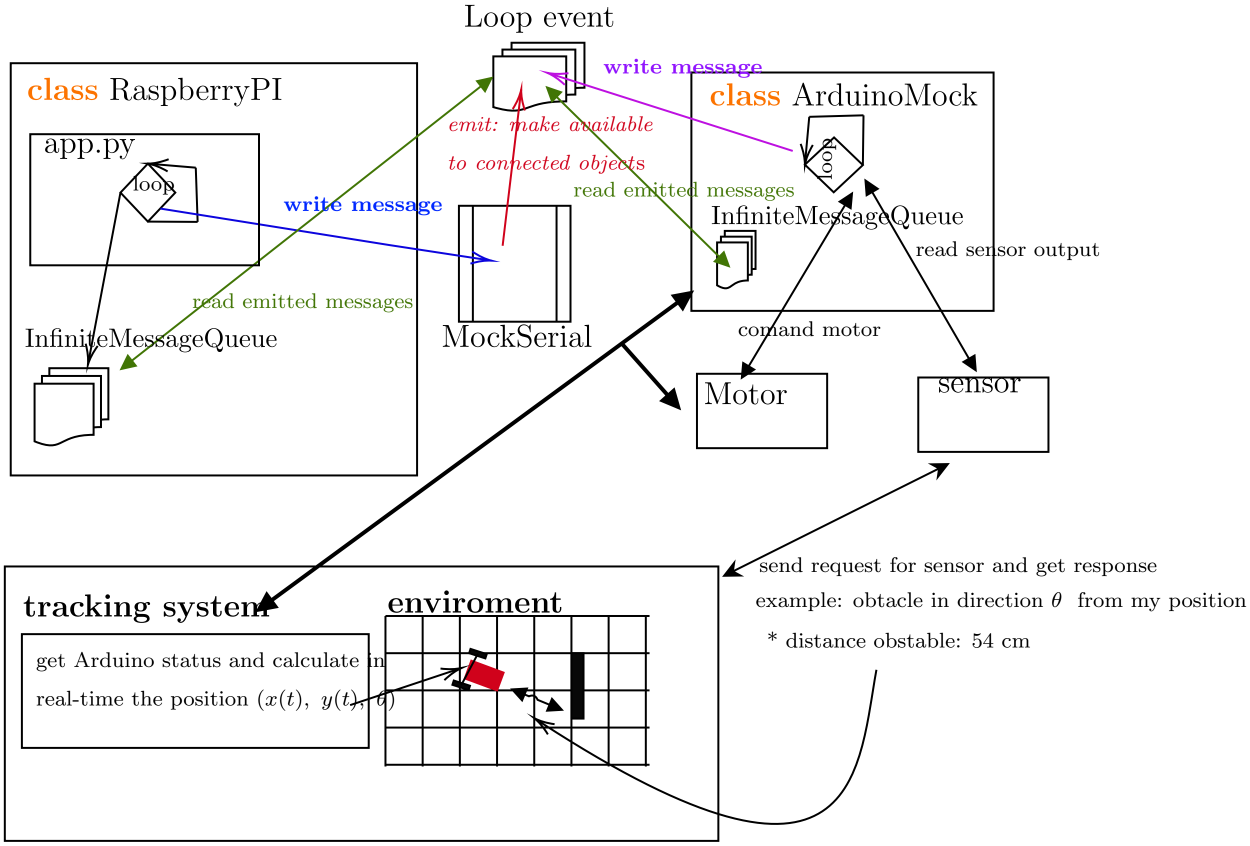
Figure 4.22: Mission summary page: graphs and maps.

4.14 Mower Simulator

To speed up development times and to be able to debug the code, a robot simulator was created. The simulator receives the commands and moves the robot virtually on the map. All sensors are simulated to provide adequate response messages. A fundamental parameter is the amount of noise present in the data collected by the sensor to be able to obtain sufficiently robust algorithms.

```python
class Simulator:
  def __init__(self, args):
        params = Parameters()
        params.dims.encode_noise_motor = args.encode_noise_sim
        params.dims.encode_noise = args.encode_noise
        params.dims.sigma_noise = args.path_noise
        params.dims.errorLidar = float(args.lidar_noise)
        params.dims.frequency_invalid_messages = args.
            frequency_invalid_message
        ...
```

The logic workflow of the prototype can be found below:



4.15 Mission Analysis

In order to analyze experiments and simulations we developed different tools to gather and display the huge amount of information produced during in a convenient form.

4.15.1 Report

Once the mission is over, the *report* program allows analyzing how the test went. The reports help to compare the two algorithms developed with the same metrics. To choose the sections to be printed in the report, the program uses the operating system's console.

Listing 4.1: The various options available in the report

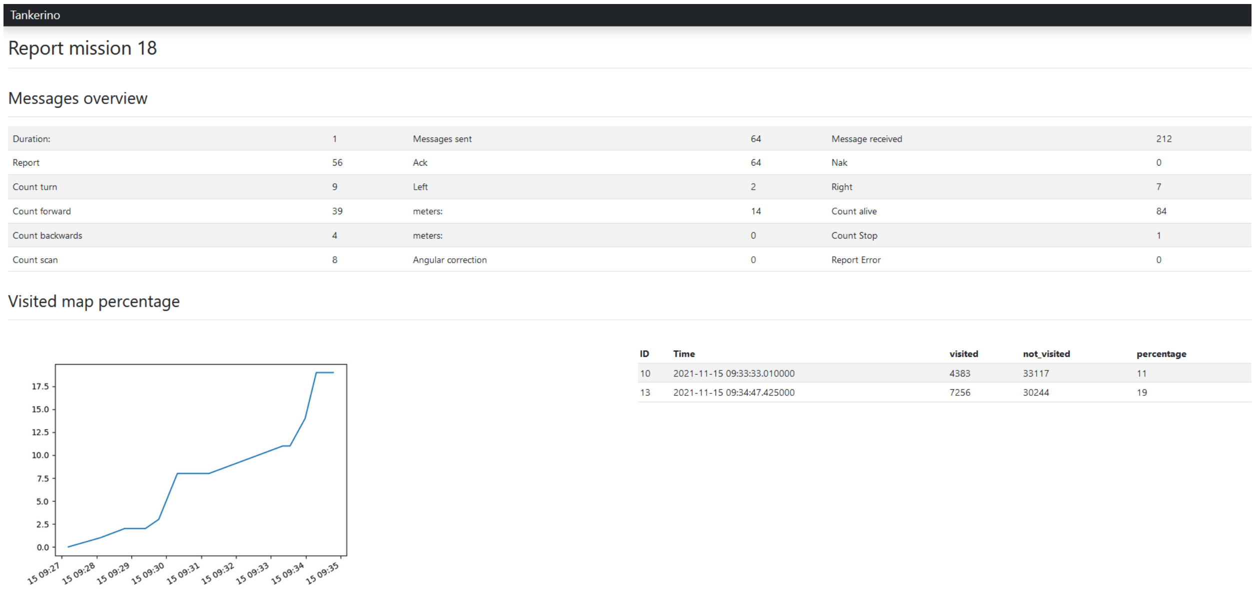
```
---Missions Analyse ---
1) Choose the mission. There are 19 missions.
   You must choose. But choose wisely
> 19
You have chosen... wisely

---Missions Analyse Main Menu ---
1)  Messages overview        [ ]
2)  Visited map percentage   [x]
9)  Select all
10) Save report
11) Exit
Choose what to include in the report
>
```

The program inserts the mission values into predefined templates. These templates are displayed or hidden depending on the user's choices.

```python
def SaveReport(options: List[int], folder_path: str, mission_id: int):
    current_folder: str = os.getcwd()
    mission_folder = os.path.join(current_folder, "missions")
    # ...
    file = open(template_file)
    template = file.read()

    if options[1] == 1:
        count_actions: CountActions = CountActions(folder_path)
        template = template.replace("{phMsg}", count_actions.get_table
            ())
    else:
        template = template.replace("{SectionMessagesOverview}",
            css_collapse)

    # ....

# Titles
template = template.replace("{ReportMissionID}", str(mission_id))
```

In this report example, it is possible to see the salient statistics of the mission:

- sent messages
- received messages
- count of turns
- how many meters the robot has travelled
- ACK and NAK
- ...



Tankerino

Report mission 18

Messages overview

Duration:	1	Messages sent	64	Message received	212
Report	56	Ack	64	Nak	0
Count turn	9	Left	2	Right	7
Count forward	39	meters:	14	Count alive	84
Count backwards	4	meters:	0	Count Stop	1
Count scan	8	Angular correction	0	Report Error	0

Visited map percentage

ID	Time	visited	not_visited	percentage
10	2021-11-15 09:33:33.010000	4383	33117	11
13	2021-11-15 09:34:47.425000	7256	30244	19

Figure 4.23: Report example

4.15.2 Telemetry

Running the telemetry tools it is possible to display all the reads from the sensors as well as the requested strategy and the calculated trajectory.

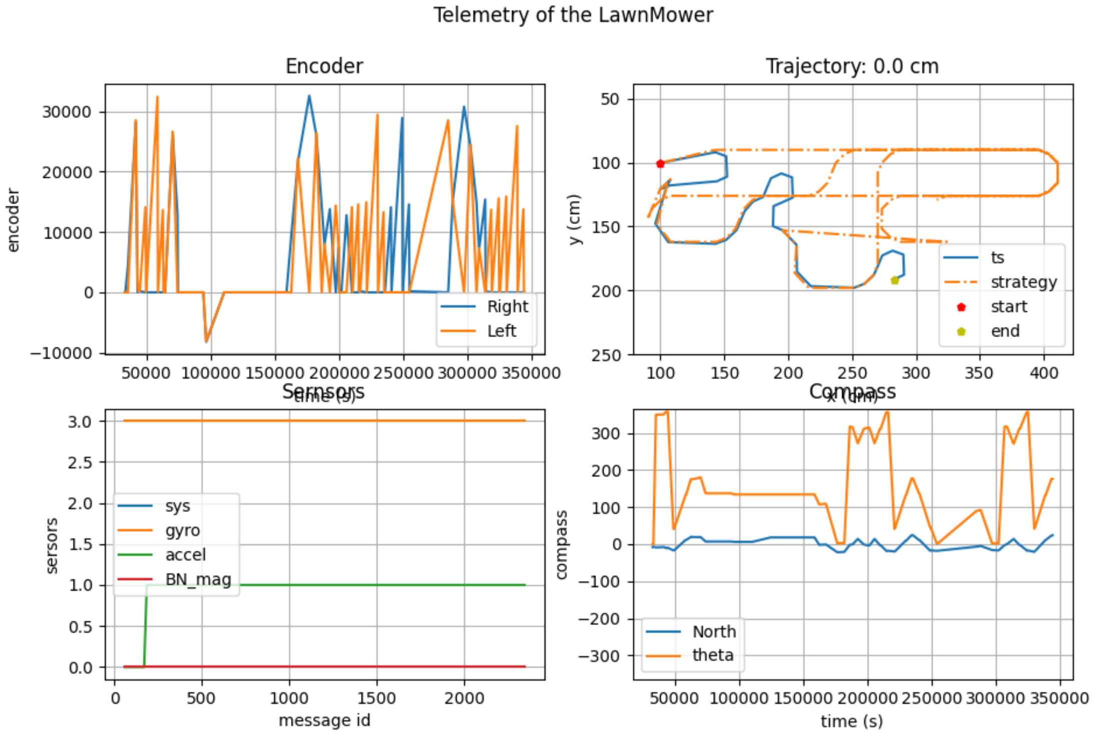


Figure 4.24: Example of the telemetry of an Experiment: (top left) value of the read encoder. (top right) calculated trajectory and requested strategy. (bottom left) sensor reads. (bottom right) angles and magnetic reads

This view is useful when analyzing the result of a mission, in particular to find bugs and errors. For example in the **Figure 4.24** we notice that the straight line (orange) going from the red dot has not been completely performed.

Chapter 5

Simulation and Experiments

In this chapter we will present some experiments performed: these are of two types:

- *Simulation*: these runs are useful on one hand to test the system with more control and on the other to experiment and evaluate different implementation of *coverage strategy*.
- *Experiment on the field*: these are the main goal of the thesis: the mower is left alone to complete its mission.

5.1 Simulations

The simulations can be classified in two different categories:

- *Ideal Mower*: in these simulations the mower follows exactly the trajectory calculated by the AI without any imperfection: useful to validate the implemented algorithms.
- *Realistic Mock*: the messages and responses are generated by the simulator presented in the **Section 4.14**.

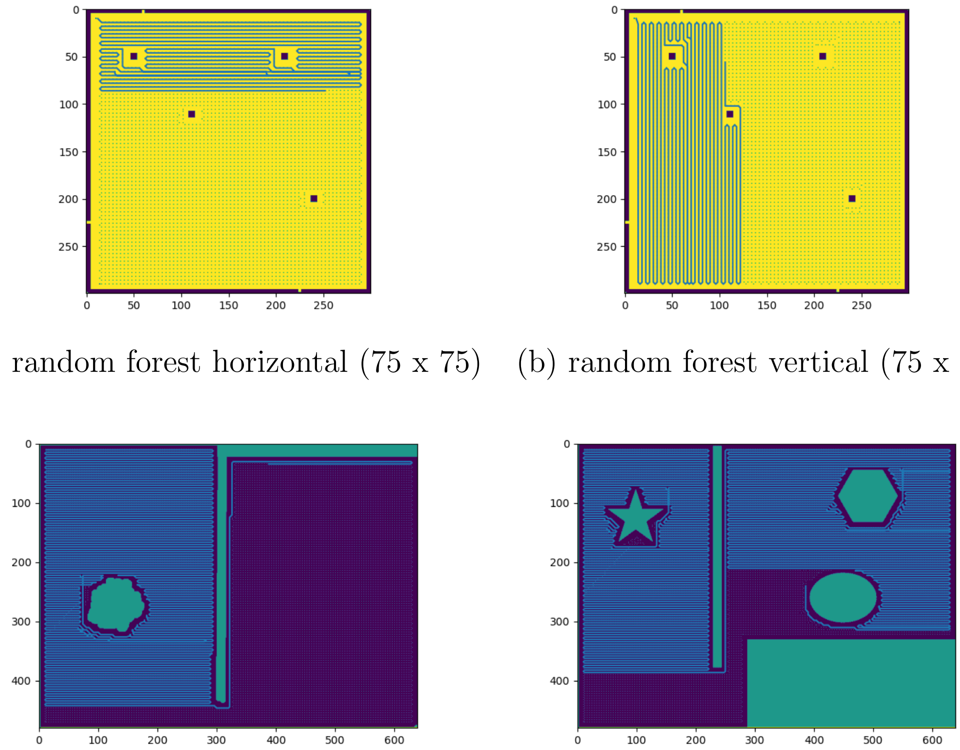
5.1.1 Ideal Mower

This simulation tests and validate only the AI part of the software. The flow is as follows:

1. *Obstacle Map Generation*: a map with different obstacle of different shape is generated. In these simulation no scan is performed since the obstacles are known.
2. *Strategy Calculation*: examining the map, the AI calculate a strategy.
3. *Update Obstacle Map*: the strategy is used to marked as visited the cells of the map crossed by the calculated trajectory. Go to point 2.

In order to evaluate different AI implementation it is convenient to defined a metric, in our case this is:

value	**description**
size map	total number of cell in the map. This depends on the algorithm
positive	number of cell not visited but visitable
negative	number of cell inaccessible (i.e. contining obstacles)
visited	number of visited cells
length path	length of path in cells units
direction changes	number of change of direction performed
theoretical best path	inferior limit of the shortest possible path



(a) random forest horizontal (75 x 75)

(b) random forest vertical (75 x 75)

(c) Bush and wall (120 x 160)

(d) geometric garden (120 x 160)

Figure 5.1: Ideal Mower in action: the dotted line is the calculated trajectory, while the continuous line is the trajectory covered up to the present point in time while running the simulation.

An important value to take into account is the number of direction change: since the mower should try to avoid zigzags, it is important to decrease the number of of turns. Here are reported some experiments done to evaluate the algorithm $\epsilon^{\star}$

maps	**(a)**	**(b)**	**(c)**	**(d)**
size	75 x 75	75 x 75	120 x 160	120 x 160
visited	4820	4820	16484	12783 (-8)
obstacle	805	805	2716	6409
visitable	4820	4820	16484	12791
length path	4836	4838	16580	13524
direction changes	220	223	548	972

Table 5.1: Results of the experiments to evaluate and improve the algorithm $\epsilon^\star$

The results of these experiments are encouraging, in particular the strategy cover almost the complete keeping the length of the path close to the lower limit.

5.1.2 Realistic Mock

In this section we are going to discuss and present one of the experiments done with the simulator.

We consider a somewhat complicated environment (**Figure 5.2**) to take the opportunity to describe possible improvements that need to be consider and implemented.

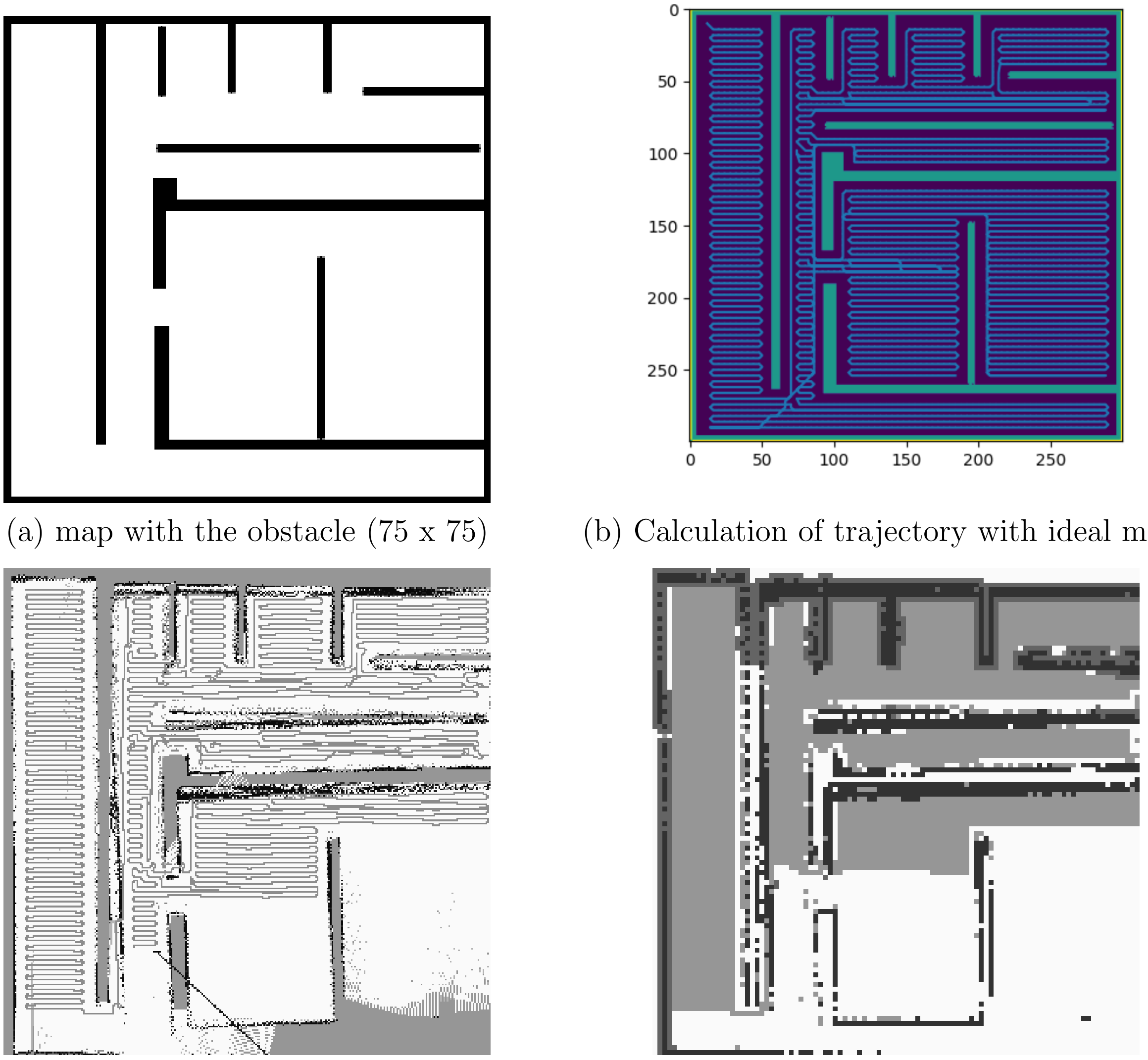
The parameters used in the $\epsilon^\star$ algorithms are as follows:

1. $\epsilon = 30$ cm
2. potential is horizontal, growing with y
3. *tolerance* is set to 10 cm, i.e. when the calculated trajectory deviate from the planed one for more then 10 cm a recalculation of the strategy is forced.
4. the implementation for the calculation of new way-points is `WayPointsNoZigZag`.
5. A strategy can contain at most 20 consecutive way-points.

The Realistic Mock has the possibility to accelerate the simulation: in this simulation a time stamp of 0.05 seconds has been used, but the actual time has been 20 time faster, i.e. if a command in the real mower would take 20 seconds to complete, in the simulated environment would just take 1 second: although the data volume that the `RaspberryPi` has to process in the time unit is much bigger then the real experiment.
This simulation has been performed on a laptop (`Lenovo`) with two processors `Intel(R) Pentium(R) CPU 3550M @ 2.30GHz`, 6 Gigabytes of `RAM`, 2 Gigabytes of Swap Space running `Linux 5.11.0-40-generic #44 20.04.2-Ubuntu x86_64 GNU/Linux`

Take a closer look at **Figure 5.2.b)** we notice that the trajectory calculated by the *Ideal mower* is not optimal: it does not conclude to visit a *room* before passing to the next one, but if exiting the door at the middle of the wall (coordinate ca. (100, 175)) it goes down the narrow corridor and visit the bottom of the map, and only at the end it completes the big room in the center - the length of the path is much longer then the ideal one (**Table 5.2**).

The same *difficulties* are registered in the simulation with the *Mock Mower*: the trajectories are almost the same. However we notice that the detected walls are not overlapping the obstacle map. This is due to the presence of small *approximations* that in such a long run add up, and make the simulated trajectory and the one calculated from the *simulated sensors* diverge from each other.
To better see this, it is useful to analyze the telemetry of the mission **Figure 5.3)**, where the two trajectory deviate by a large amount from each other. To solve this Simultaneous localization and mapping (SLAM) technique should be considered and implemented.



(a) map with the obstacle (75 x 75)

(b) Calculation of trajectory with ideal mower

(c) progress of the simulation

(d) tilling map for (c)

Figure 5.2: Realistic Mower in action: we notice that shape of the walls is not vertical, meaning that the simulation and the tracked path are not the same

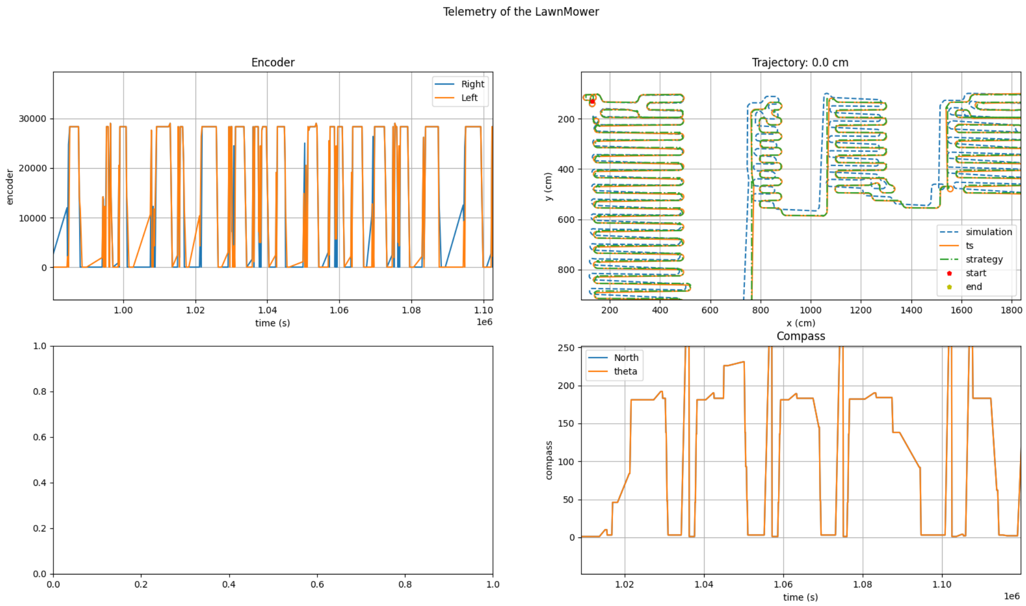


Figure 5.3: Telemetry corresponding to the simulation in **Figure 5.2**: we can notice the difference of the simulated trajectory and the calculated in the main process.

Another important aspect that we need to underline comes from the comparison of the metric between the ideal coverage path and the simulated one. In particular the number of turns expected (502) and the one actually performed (over 1185). Most of the turning are consequences of the correction of the trajectory, i,e, the trajectory deviates more then 10 cm from the planed one and the path is recalculated to move the mower closer to the center of a tilling-cell.

metrics	**Ideal Mower**
size	75 x 75
visited	3567
obstacle	2058
visitable	3567
length path	4029 (+462, 112%)
direction changes	502

Table 5.2: Results of the experiments with ideal mower $\epsilon^{\star}$ - map in **Figure 5.3**

It should also be note the high number of scans performed: in the reality every scan is an expensive operation (in term of time) and this number should be reduced.

Duration:	1h	Messages sent	6219	Message received	872056
Report	5444	Ack	6219	Nak	0
Count turn	1185	Left	583	Right	602
Count forward	3541	meters:	1192	Count alive	859752
Count backwards	31	meters:	7	Count Stop	772
Count scan	641	Angular correction	46	Report Error	0

Table 5.4: Summary of the report for the Mock Mower simulation

5.2 Experiments on the Field

In this chapter, we will describe how the robot behaves in the field. Initially, we started with a simple map 5 meters by 3 meters with no obstacles.

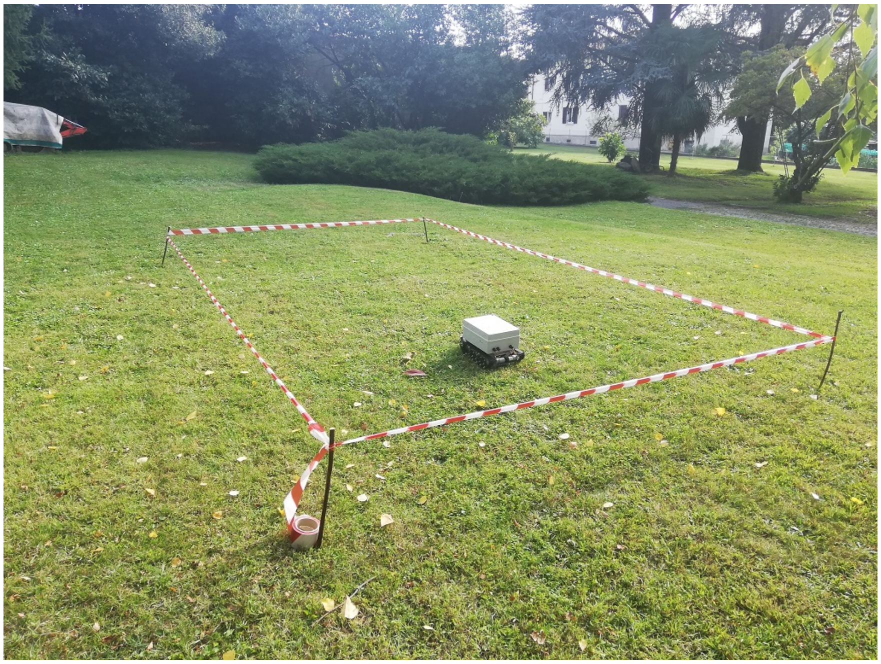


Figure 5.4: Robot in the field.

5.2.1 Measures

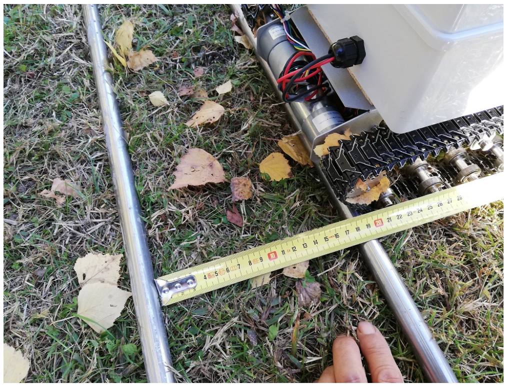
The first operation carried out was to check that the formulas used in the simulator and the code have a confirmation also in reality. There are three movements of the tank: go forward, go back and turn.

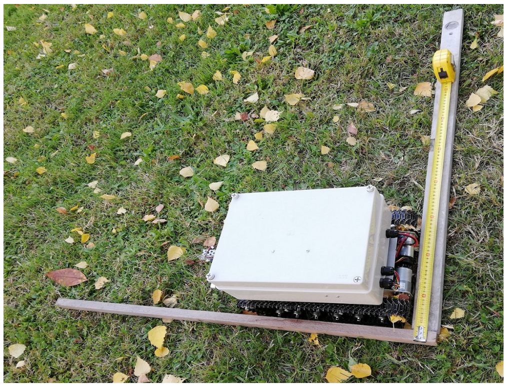
To calculate the distance travelled forward or backwards we rely on the encoders. The variables used to turn are:

- the measurement of the gear wheel connected to the motor

- the length and height of the track

Wheel


Tick


Turn start

Turn end

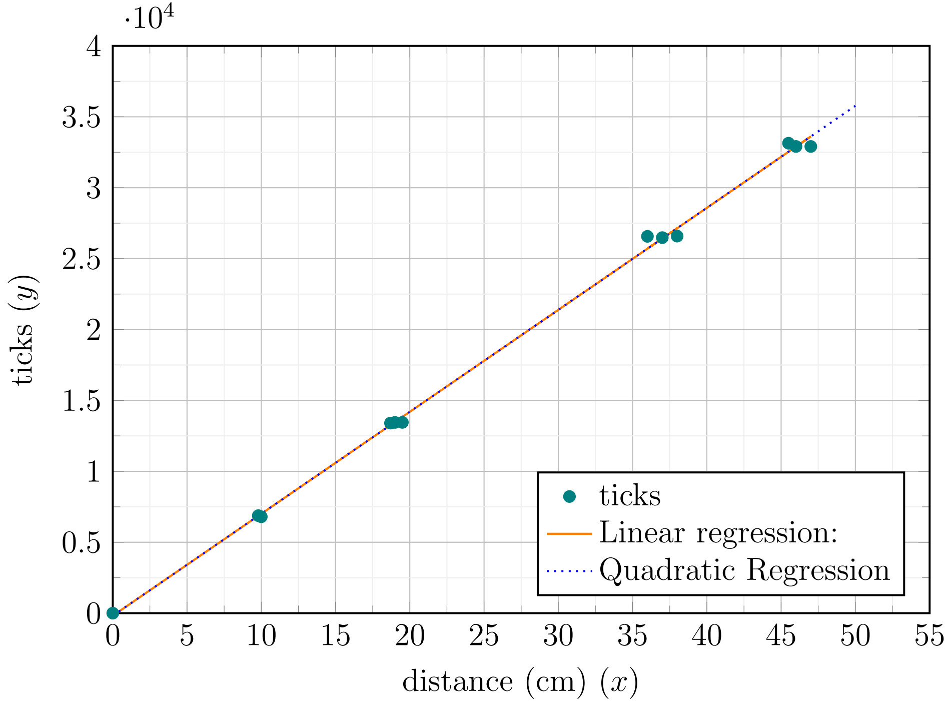
Forward-Backward

To correlate the distance between the cm travelled and the values read by the encoders, several tests were carried out. The actual value is often different from the value declared by the manufacturer.

Among the various causes we can find the confirmation of the ground or if the wheel has lost grip on a particular point. The shape of a tank helps mitigate these problems but problems can still arise.

The result of the experiments is the following table:

ms id	planned cm	measured cm	json parameter	tick Left	tick Right	Mean
5	10	10	6550	6804	6793	6798,5
7	10	9,8	6550	6876	6878	6877
11	20	19,5	13100	13461	13443	13452
13	20	19	13100	13450	13450	13450
15	20	18,7	13100	13426	13381	13403,5
17	40	38	26200	26587	26588	26587,5
19	40	37	26200	26485	26477	26481
21	40	36	26200	26572	26571	26571,5
27	50	45,5	32750	33135	33142	33138,5
31	50	47	32750	32914	32906	32910
32	50	46	32750	32914	32906	32910



To infer the ticks for centimetre we can use simple linear regression, since the distance and the number of ticks are linearly dependent.
The calculation leads to

$$y = 719.09 \cdot x - 183.72, \tag{5.1}$$

with

$$\texttt{Error} = \sqrt{\sum_{i=0}^{N} (y_i - \hat{x}_i)^2} = 53.23.$$

Using a quadratic regression

$$y = 0.00780244x^2 + 718.693x - 180.478 \tag{5.2}$$

with

$$\texttt{Error} = \sqrt{\sum_{i=0}^{N} (y_i - \hat{x}_i)^2} = 13.77.$$

Notice that the both linear and quadratic regression predict that at 0 centimetre the number of ticks is negative. Performing more measurement between 0 and 10 cm we obtain extremely noisy values and for this reason it is necessary to divide the behaviour between up to 10 cm and beyond this distance.
From these experiments we could conclude that the best fit is the quadratic one. Our approximation, however, is not perfect, but good enough from a practical point of view, i.e. the actual run of the mower on the field justifies it.
Furthermore, the value found by this approximation, which correspond to a turning wheel of diameter 3.9 cm, agree also, in first approximation, with the diameter of the wheel that moves the tank (3.8 cm).

Turn

When we analyzed the prototype with the wheel, the identification of the center of a rotation was straightforward, since we block one of the wheel, the center of rotation was in the middle of the blocked wheel.

To identify the center of rotation of the tank version we fixed a coordinate system and measure the position of at least one points before and after the turn, in particular we rotate of 90 degree and measured that the point $P = (19, 27.5)$ has been transformed to the point $Q = (33, 39)$.

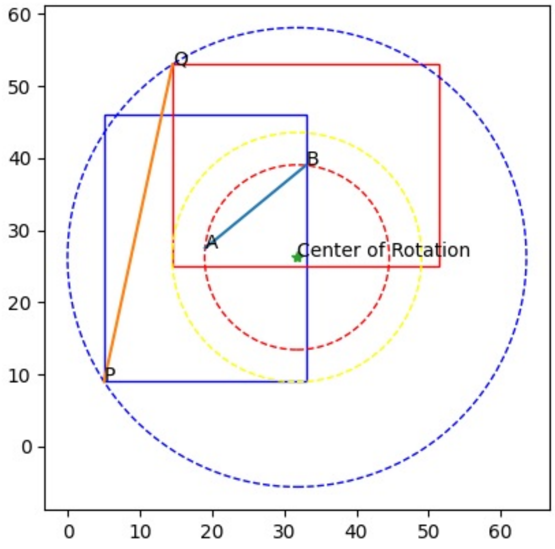


Figure 5.5: Graph for calculating the centre of rotation.

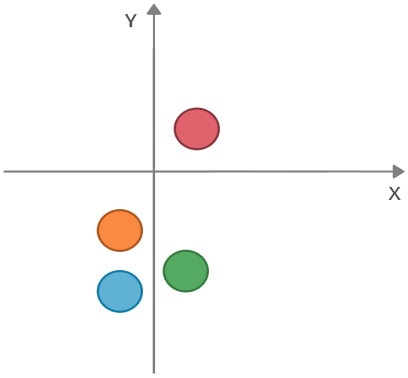
By using the fact the the mower is a rigid body, we calculate the center of rotation for the measured points: we found that the center is in the middle (width and length) of the the tank trunk, which agrees with the expected result.

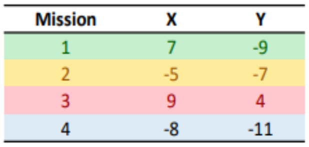
5.2.2 Maintaining Straight Path and Executing 180° Turn

To test the drive system, the tank first needs some basic tests. Testing the mower's ability to maintain a straight path and execute a 180 degree turn is beneficial to prove the functionality of the drive system. To accomplish this, a testing procedure is provided.

1. At the beginning of the run, measure the absolute position and orientation of the vehicle
2. Run the vehicle for four meters, in ten steps of 40cm. After 40 cm adjust the trajectory going a few degrees to the right or left, if necessary
3. Make a 180 degree turns on the spot
4. Return to the starting area, measure the absolute position and orientation of the vehicle
5. Compare the absolute position to the robot's calculated position, based on odometry

The experiment is running four times. The result is the following graph:





Mission	X	Y
1	7	-9
2	-5	-7
3	9	4
4	-8	-11

Table 5.5: Experimental results: each circle represents measured position at the end of a single experiment.

In the graph we can see the differences between the real position of the rover in the experiment with respect to the planned point.

The most obvious errors are found on the y-axis. This is because even a few degrees of difference from the desired trajectory lead to some errors in the measurements. On the other hand, the encoders that are responsible for measuring the horizontal movement behave quite accurately.

5.2.3 Obstacle detection

Another early experiment was recognizing obstacles and then deflecting strategy accordingly.

To recognize obstacles, the robot could use all three types of sensors it has. It mounts:
- ToF (Time of Flight) LiDAR
- five ultrasonic sonar
- camera

The lidar is very accurate in measurements and has an action radius of about 12m and. In the first tests carried out, it is used alone to test the pros and cons of this technology. It is mounted at about 20 cm height.

Setup

A wooden panel is placed at 160 cm, which simulates the obstacle to be overcome. The following photo is taken by the camera mounted on the robot.

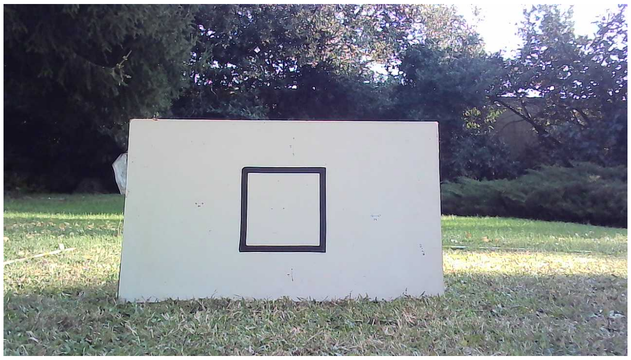


Figure 5.6: Obstacle to overcome

The first operation that is performed is the scan of the area to discover any obstacles. Once the scan has been performed, the map is updated with the new data. An image is produced to show the result of the measurements is possible.

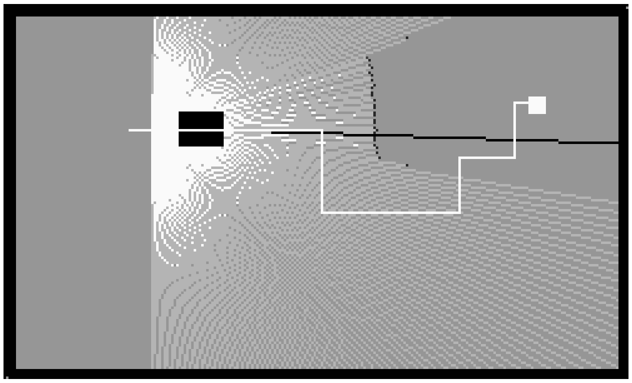


Figure 5.7: This figure shows the trajectory with an obstacle inside the field.

The panel can be clearly recognized in the image, the scanner works pretty well. From this image, it can be seen that the sensor also introduces noise. There are several imaginary points. They do not exist in reality but they are classified as objects. There are several techniques to overcome this problem and will be implemented as the project progresses.

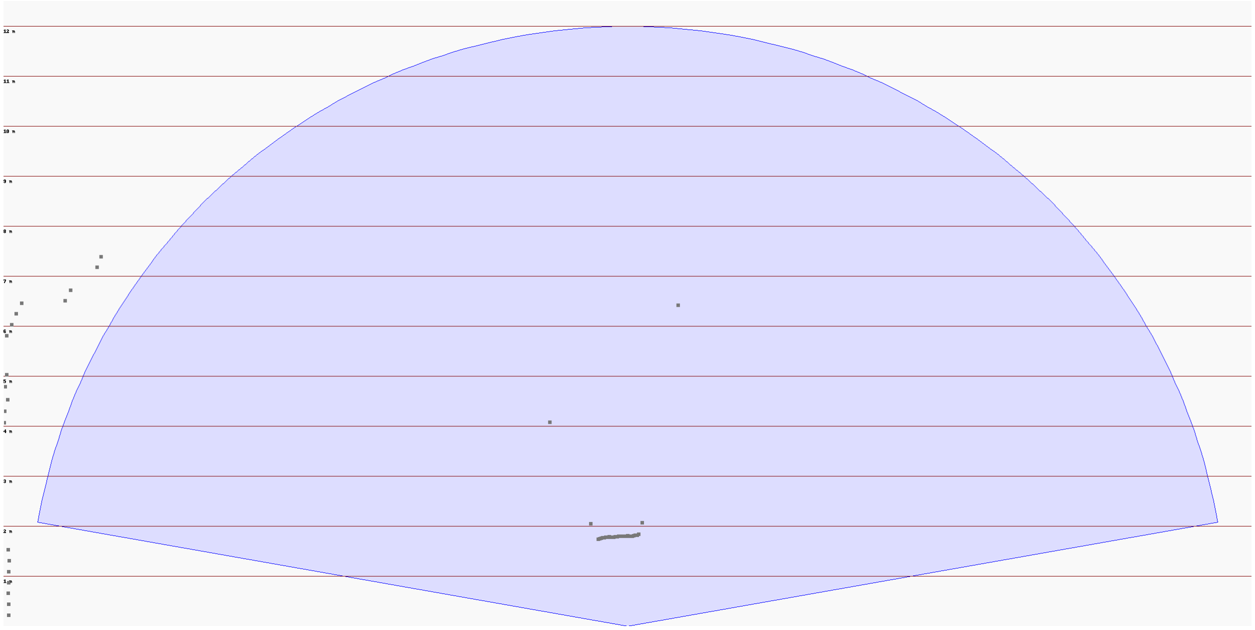


Figure 5.8: This image shows the lidar scan. It is possible to recognize the obstacle in front of the robot.

As we saw in chapter 4 in detail the formula updates the map with the scan results. Once this operation has been completed, it is possible to create a strategy to reach the next goal. In this test, the rover is going from the lower-left corner to the lower right corner.

As we saw in chapter 4, the A-star algorithm is used to find the way to the goal. Map creation is explained in detail in Chapter 3.

As can be seen from figure 5.5, unreachable space is added around obstacles. This operation allows you to add a fixed safe distance from the trajectory to the obstacles. The robot's path can only go through free cells.

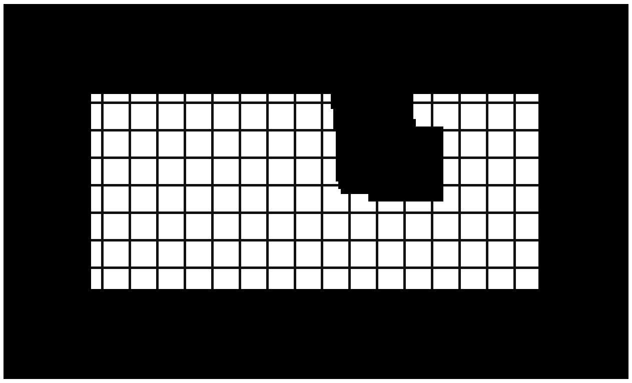


Figure 5.9: In this image, it is possible to see an extra space around the obstacle.

5.2.4 Go around

Another experiment is exploring the map and checking that the commands are executed correctly.Initially, the map will be free of obstacles so it will be easier to follow the robot's movements.

The robot starts from the 100x100 (cm) position. The origin of the axes is the top left. Once the trajectory has been calculated, the list of commands is displayed in the graphic interface. The list of commands allows you to check in real-time if the strategy is executed.

Listing 5.1: Example of the planned robot movements

```
STRATEGY 1 —— count step 8
—> go   [FORWARD ] for 40 cm  – wished_compass_angle = 0
—> go   [FORWARD ] for 18 cm  – wished_compass_angle = 0
—> turn [LEFT    ] for 90 deg – wished_compass_angle = 270
—> go   [FORWARD ] for  0 cm  – wished_compass_angle = 270
—> go   [RIGHT   ] for 90 deg – wished_compass_angle = 0
—> go   [BACKWARD] for  4 cm  – wished_compass_angle = 0
—> go   [RIGHT   ] for 90 deg – wished_compass_angle = 90
—> go   [BACKWARD] for 10 cm  – wished_compass_angle = 90
```

Once the robot has reached the goal in the upper right corner a new strategy must be calculated.

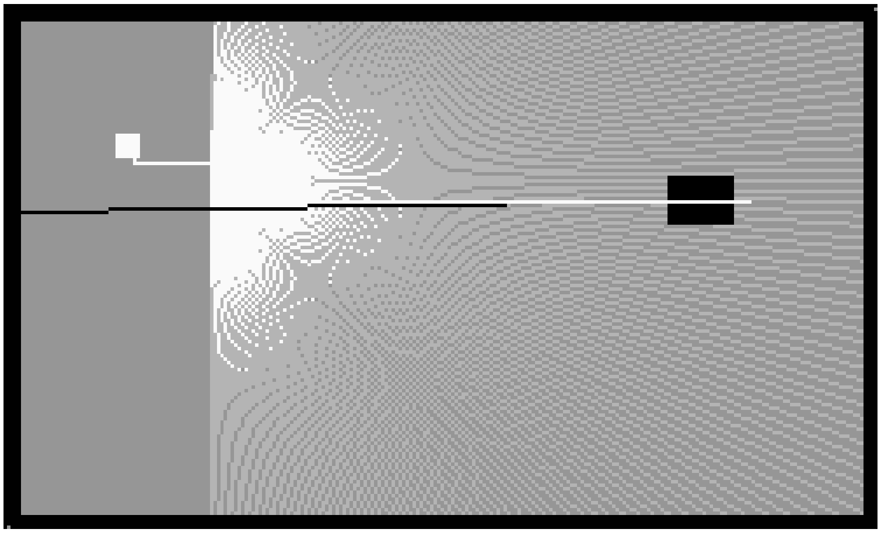


Figure 5.10: Position of the robot after the first turn.

The new strategy includes a 180-degree rotation. The goal is placed about 20 cm away from the starting position. In this experiment, the robot continues to make these back and forth movements for about 5 minutes.

Once the robot has been stopped, the result of the cells visited is in figure 5.8. The black represents the points not seen, the white colour the points visited. If a cell has a more intense colour it means that the cell has been visited several times.

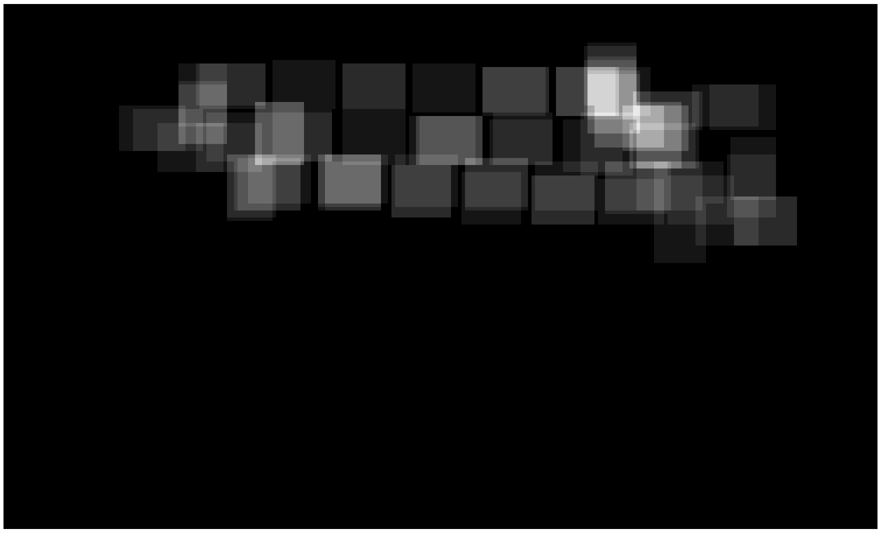


Figure 5.11: Cells visited by the mower

Once the robot was stopped, we were also able to detect its real position. The actual position from where the robot thought it was differed by about 5cm on the x-axis while it differed by 40cm on the y-axis. More similar experiments were performed but with similar results. This behaviour is not acceptable and therefore some SLAM algorithm should be implemented.

5.2.5 Comparison of implemented AI-Algorithms

This paragraph compares the result of the two developed algorithms that drive the mower. The parameters taken into consideration will be those elaborated by the report. The experiments to be carried out will be two. The first experiment is carried out on a completely blank map. The second experiment is carried out on a map with an obstacle in the middle. The map will always be 5x3 meters in size. In these experiments, both the simulator and the robot will be tested for a total of eight reports:

Report name	Mode	Map type	Algorithm
1	simulator	blank map	grid based coverage path planning
2	simulator	blank map	e-star
3	simulator	map with obstacle	grid based coverage path planning
4	simulator	map with obstacle	e-star
5	mower	blank map	grid based coverage path planning
6	mower	blank map	e-star
7	mower	map with obstacle	grid based coverage path planning
8	mower	map with obstacle	e-star

The reports are enclosed in the thesis in the homonymous chapter.

The two algorithms developed are very different from each other. E-star is much more innovative when it comes to handling complicated map situations. Its purpose is to visit the map as quickly as possible by optimizing the route. On the other hand, the "Grid-based coverage path planning" algorithm is much simpler as it tries to go from left to right in the map until it is completely covered.

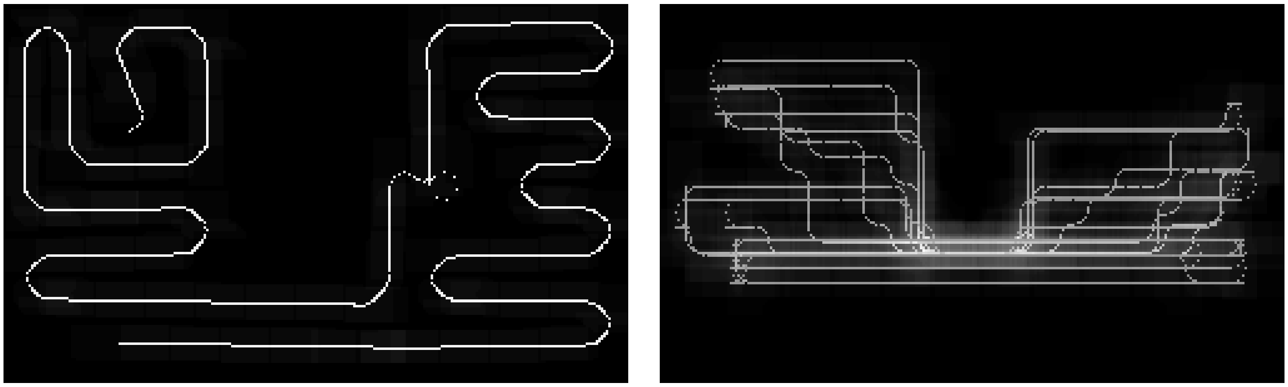


Figure 5.12: The two images compare the two algorithms in the obstacle map in the simulator.

In the simulations, both algorithms do their job well. They manage to cover the map entirely and avoid obstacles.

The problem arises when the noise, present in the real world, is introduced into the simulator. As a result, the commands are not always executed perfectly. For example, a wheel can slip, or the trajectory can be slightly incorrect from uneven terrain.

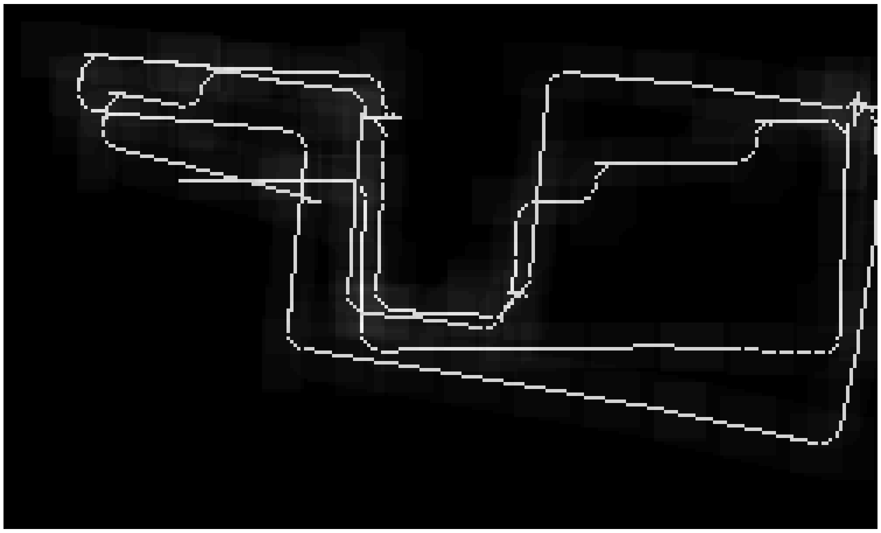


Figure 5.13: In this image, it is possible to see how the noise affects the trajectory.

Chapter 6

Conclusions

The goal of this project is to produce an autonomous lawnmower prototype. The lawnmower should use sensory devices to detect the aspects of the mowing field and use this information to navigate the area with no user input, mowing as much of the field as possible while avoiding obstacles and staying on the mowing area. The objective of the thesis is very ambitious; all commercial lawnmowers use the perimeter wire.

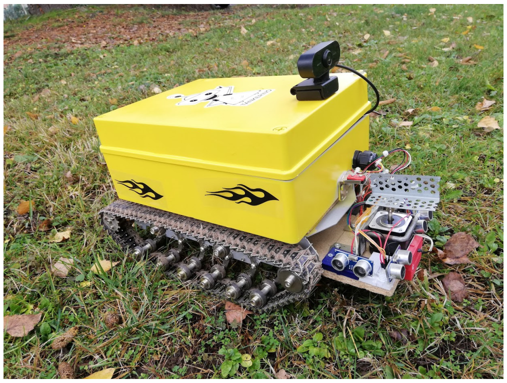


Figure 6.1: Tankerino

The project goals were not fully achieved. Nevertheless, the final result is remarkable. The mower can perform up to 20 minutes an optimal way. During those 20 minutes, the robot visited 30% of the total area without going out of the edges or hitting any obstacle. But then the sum of many minor errors affects the robot's location. After about 20 minutes and up to about 30 minutes, it is possible to see some localisation errors, which could still be considered acceptable. After approximately 30 minutes, the robot begins to exit the test area. This problem, well known in the literature, can be avoided by implementing SLAM techniques. The cheap and not always accurate sensors increase the problem of localization.

Writing all the code forces us to deal with many obstacles. For example, there were both software problems and hardware difficulties. Building a robot framework was not easy; real-time applications are usually complicated.

Therefore, this choice has significantly lengthened development times. Using ROS (The Robot Operating System) would undoubtedly have decreased development and hidden most of the complex.

Our framework:

- it consists of a number of customizable interfaces that can be switch at will via simple configuration parameters: we can for example change the implementation of the message type used to communicate between the microcontrollers and the main process, change the type of vehicle used, by just changing the tracking system that consumes the incoming data, or the generator of the strategy as well as the translation between the graph representing the high level path at the grid level of the map to the actual trajectory or succession of commands to be sent to the robot.
- it contains two simulators to debug and experiment with different scales of realism.
- It is easily usable with its UI and produces readable reports to analyse statistics and telemetry.
- The design is kept simple, but it is easily extendable to include new features, such as processing data from the camera.
- working prototype.

Having built a fairly reliable framework could now increase the speed of development.

The future works to be implemented are using a SLAM algorithm and moving some part of the logic in Arduino to speed up the robot. Another way forward to help the robot locate it could be the implementation of DenseDepth algorithms.

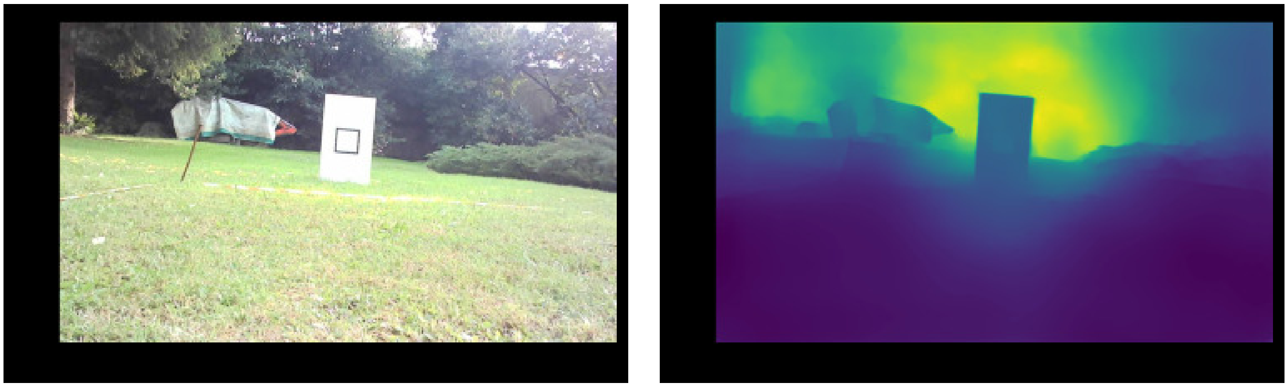


Figure 6.2: Result of the image with a Dense Depth algorithm taken by the robot camera. The program can detect the obstacle by its different colours.

Chapter 7

Budget and Finance Discussion

This project does not have a sponsor, so the cost is of great concern. All the sensors used are commercial and easy to retrieve.

The highest cost was for the chassis. Analyzing better the price 85 € are for the tank. The remaining 115 € are between taxes and shipping costs. The second most expensive item is the Raspberry Pi. It is the robot's heart and allows you to manage everything with sufficient power. All other components are pretty inexpensive.

The final price could be in line with commercial lawnmowers that have a price range ranging from € 1000 to € 2500€. The table with the main components used is attached.

Name	Pz	**Price€**
Chassy	1	200€
Raspberry Pi 4	1	95€
Motor w/Encoder	2	27€
Stepper Motor	1	22€
Lidar	1	45€
Ultrasonic Sensor	5	11€
Battery	1	12€
Rele	1	9€
Mini Switch	1	7€
Webcam	1	15€
BNO055	1	28€
Arduino Mega	2	16€
Breadboard Kit	2	15€
H-bridge	1	27€
Total	-	**556€**

Chapter 8

Reports

This chapter includes the reports of the eight tests carried out. Each report is a pdf file that contains the principal information elaborate after the mission end.

The first four reports are performed using the simulator. The next four are the field trials with the robot. The reports help us compare the two algorithms and understand how they behave in the two tested situations.

Both algorithms have excellent behaviour in the simulator test, exploring about 80% of the area. Even in the presence of an obstacle, the two algorithms perform well.

The reports map immediately helps us to detect two differences:

- $\epsilon^{\star}$ covers the edge. The grid-based algorithm instead builds a safety edge on the perimeter.
- The grid-based algorithm is less cautious near obstacles.

The user can easily modify the safe distance from the map edges and the obstacles editing the XML. For these simulations, both were set to 20cm.

The robot took about 30 minutes to explore half of the map in field tests. As written several times, the noise coming from the sensors is one of the biggest problems faced.

In the first field test, the noise from the lidar made the program misclassifying some areas. As a result, the robot changes its trajectory because of those fake obstacles.

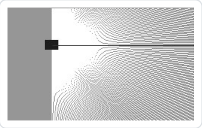
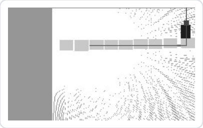
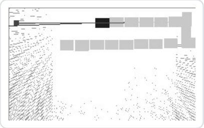
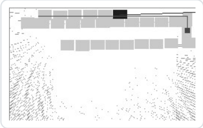
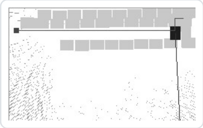
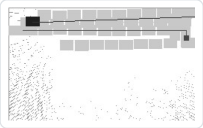
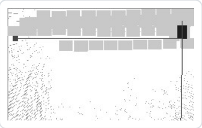
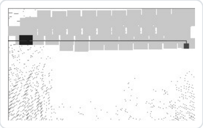
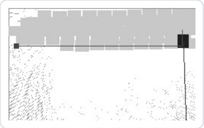
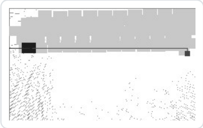
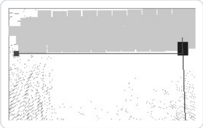
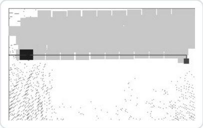
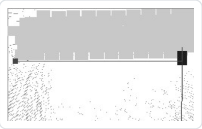
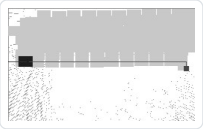
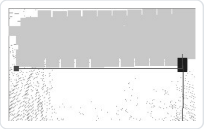
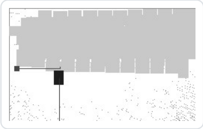
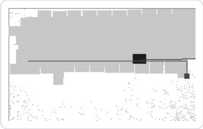
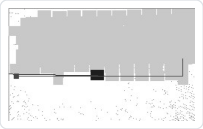
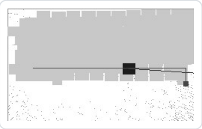
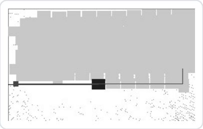
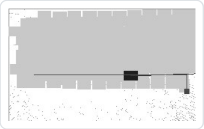
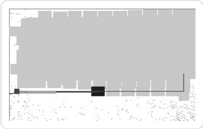
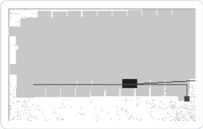
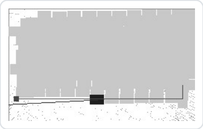
Some messages are not decoded in the field test. This problem also lengthens the time it takes to complete the mission.

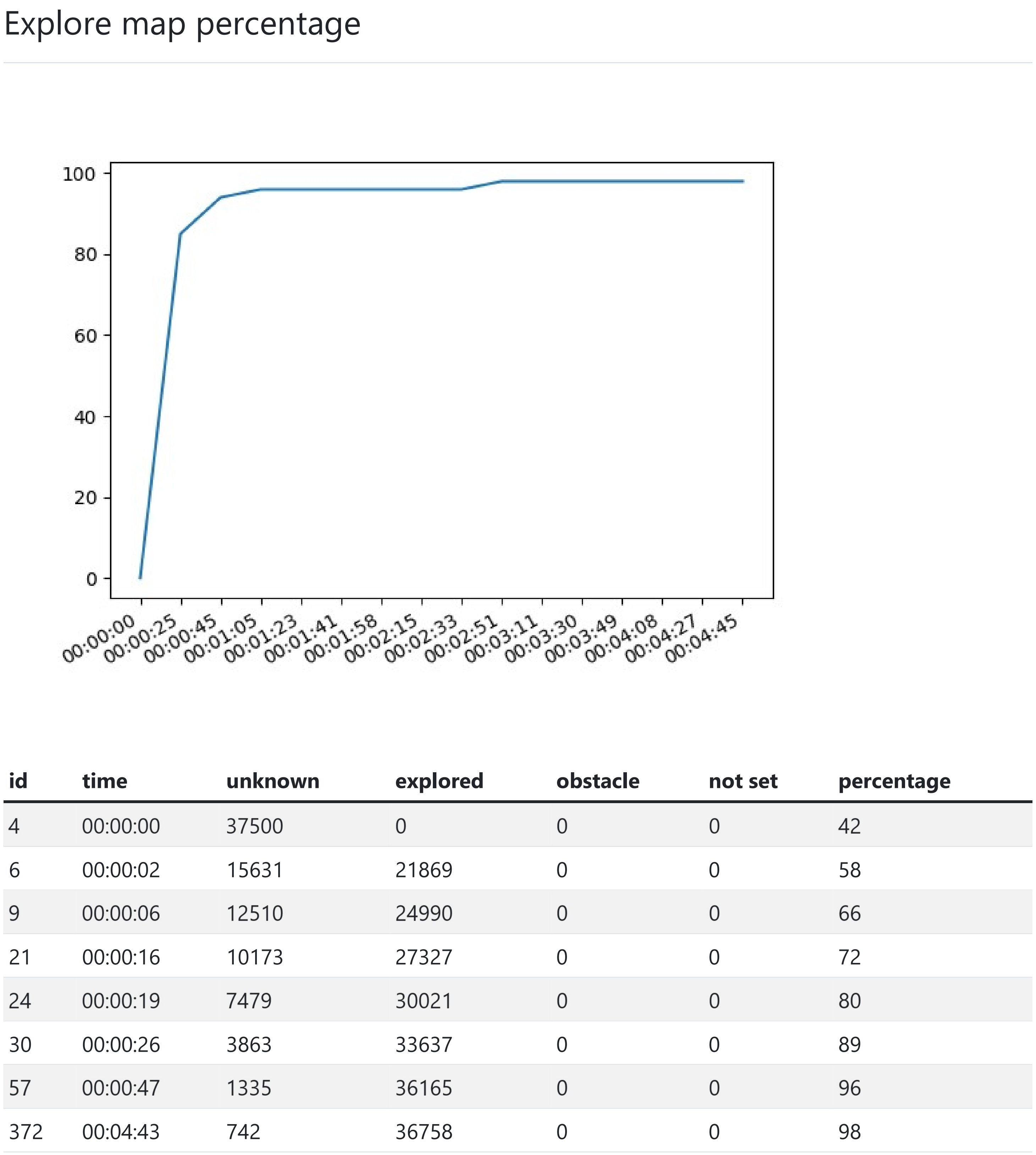
Gridbase - empty map - simulation

Messages overview

Duration:	00:04:44	Messages sent	374	Message received	1141
Report	362	Ack	374	Nak	0
Count turn	49	Left	8	Right	41
Count forward	254	meters:	98	Count alive	393
Count backwards	26	meters:	2	Count Stop	1
Count scan	12	Angular correction	29	Report Error	0

Maps overview

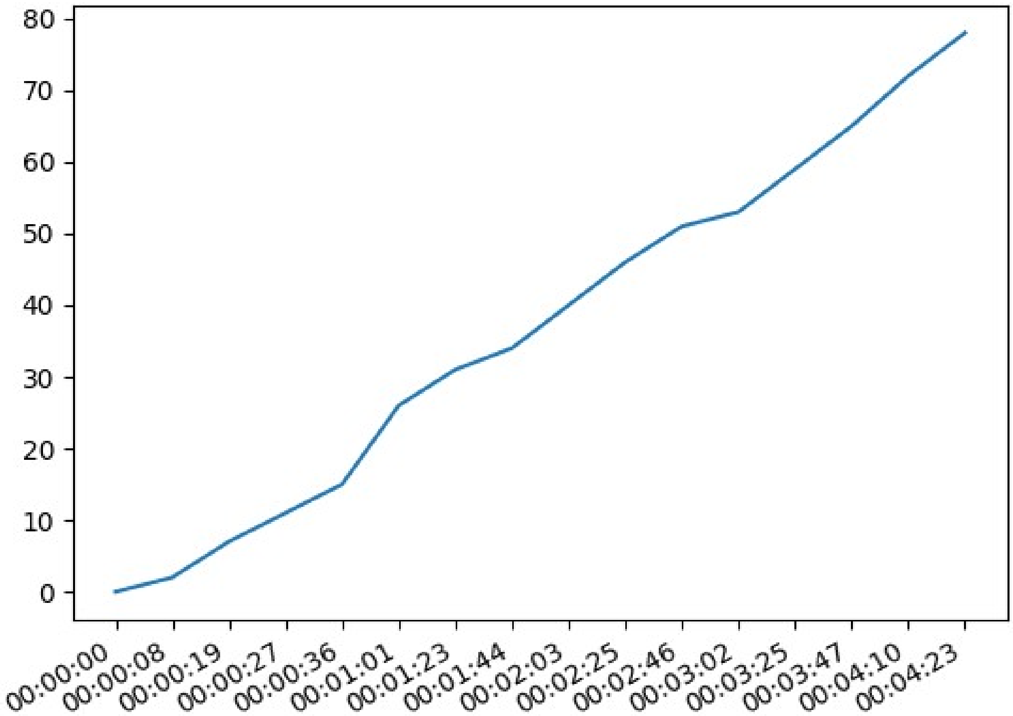



























Explore map percentage

id	time	unknown	explored	obstacle	not set	percentage
4	00:00:00	37500	0	0	0	42
6	00:00:02	15631	21869	0	0	58
9	00:00:06	12510	24990	0	0	66
21	00:00:16	10173	27327	0	0	72
24	00:00:19	7479	30021	0	0	80
30	00:00:26	3863	33637	0	0	89
57	00:00:47	1335	36165	0	0	96
372	00:04:43	742	36758	0	0	98

Visited map percentage



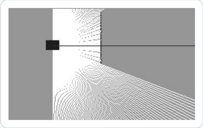
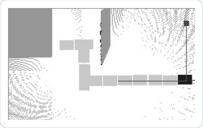
ID	Time	visited	not visited	obstacle	percentage
1	00:00:00	0	37500	0	0
10	00:00:27	4163	33337	0	11
20	00:01:25	11716	25784	0	31
30	00:02:34	17703	19797	0	47
40	00:03:47	24707	12793	0	65
45	00:04:23	29535	7965	0	78

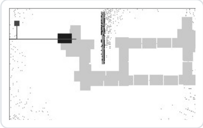
Gridbase - map with obstacle - simulation

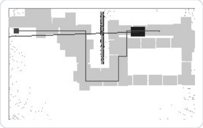
Messages overview

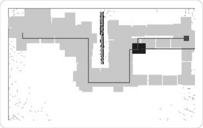
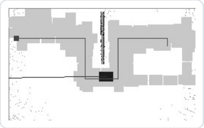
Duration:	00:07:23	Messages sent	532	Message received	1662
Report	518	Ack	532	Nak	0
Count turn	141	Left	65	Right	76
Count forward	294	meters:	104	Count alive	598
Count backwards	40	meters:	4	Count Stop	2
Count scan	14	Angular correction	38	Report Error	0

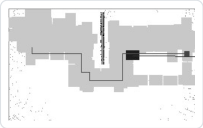
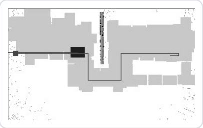
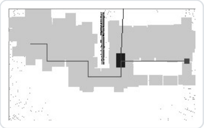
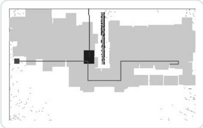
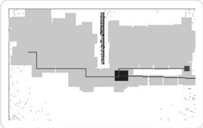
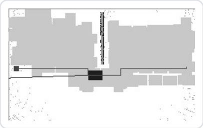
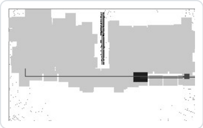
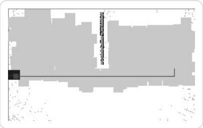
Maps overview

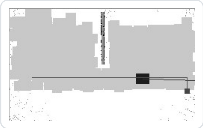
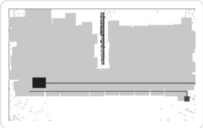
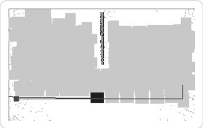





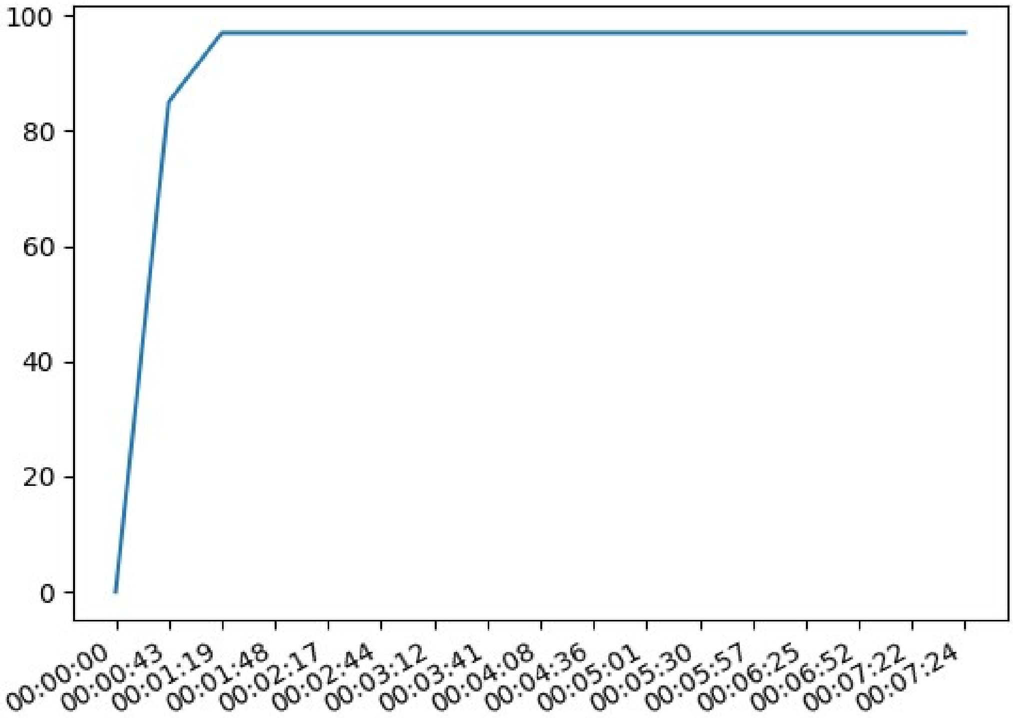


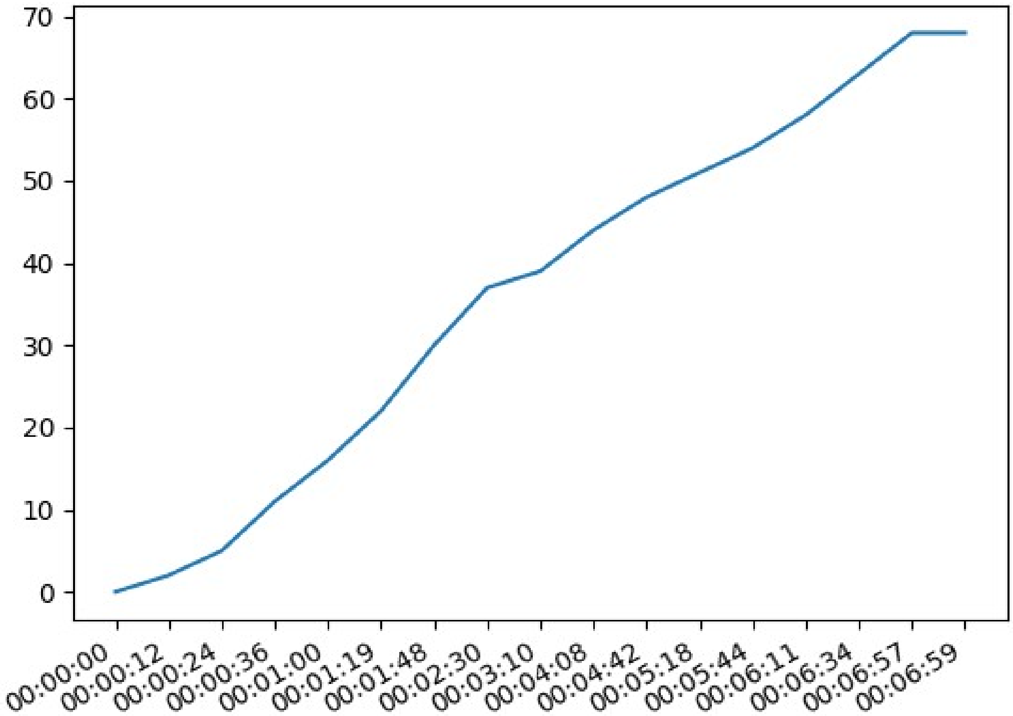




Explore map percentage



id	time	unknown	explored	obstacle	not set	percentage
4	00:00:00	37500	0	0	0	27
10	00:00:08	20504	16931	64	1	45
13	00:00:12	17323	20112	64	1	53
16	00:00:16	11319	26156	25	0	69
18	00:00:20	8231	29244	25	0	77
27	00:00:31	5983	31463	53	1	83
52	00:00:55	3285	34133	81	1	91
66	00:01:12	973	36424	97	6	97
530	00:07:22	809	36577	109	5	97

Visited map percentage



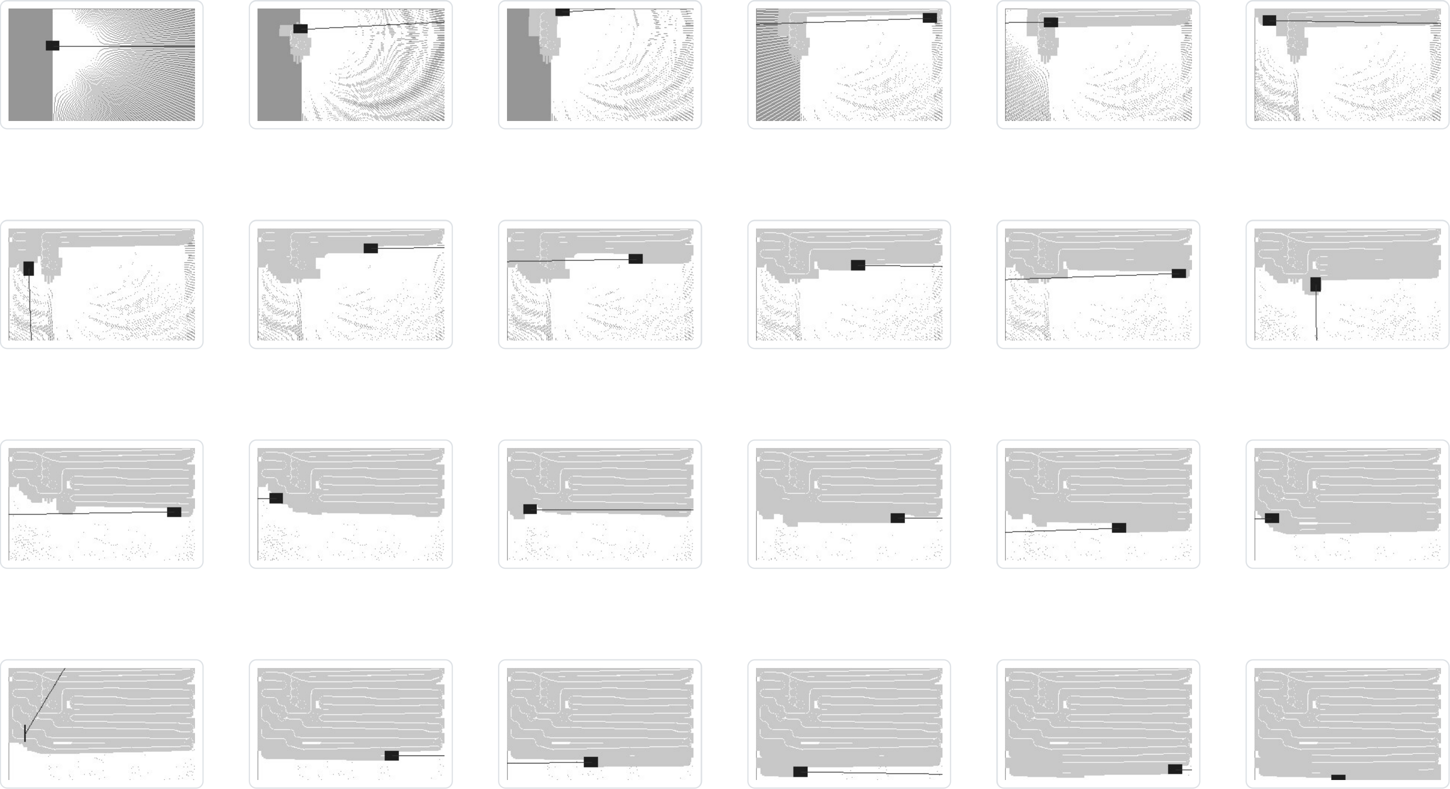
ID	Time	visited	not visited	obstacle	percentage
1	00:00:00	0	37391	109	0
10	00:00:36	4286	33105	109	11
20	00:02:06	12644	24747	109	33
30	00:04:26	17851	19540	109	47
40	00:06:11	21975	15416	109	58
47	00:06:59	25781	11610	109	68

eStar - empty map - simulation

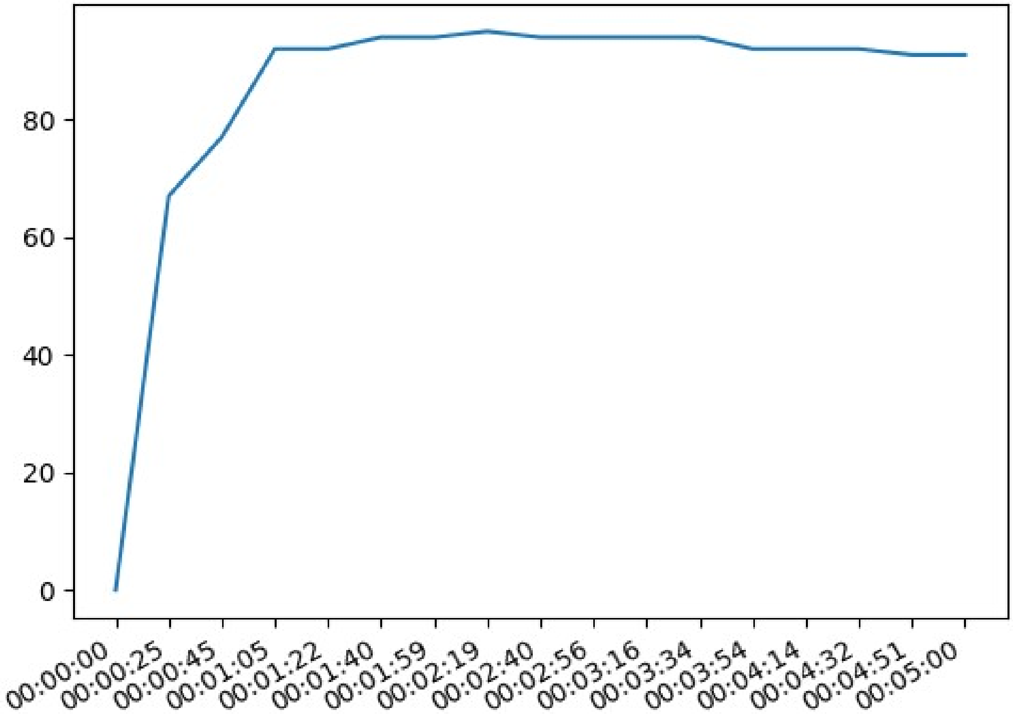
Messages overview

Duration:	00:04:59	Messages sent	348	Message received	1069
Report	311	Ack	347	Nak	0
Count turn	79	Left	36	Right	43
Count forward	163	meters:	57	Count alive	375
Count backwards	0	meters:	0	Count Stop	57
Count scan	36	Angular correction	10	Report Error	0

Maps overview

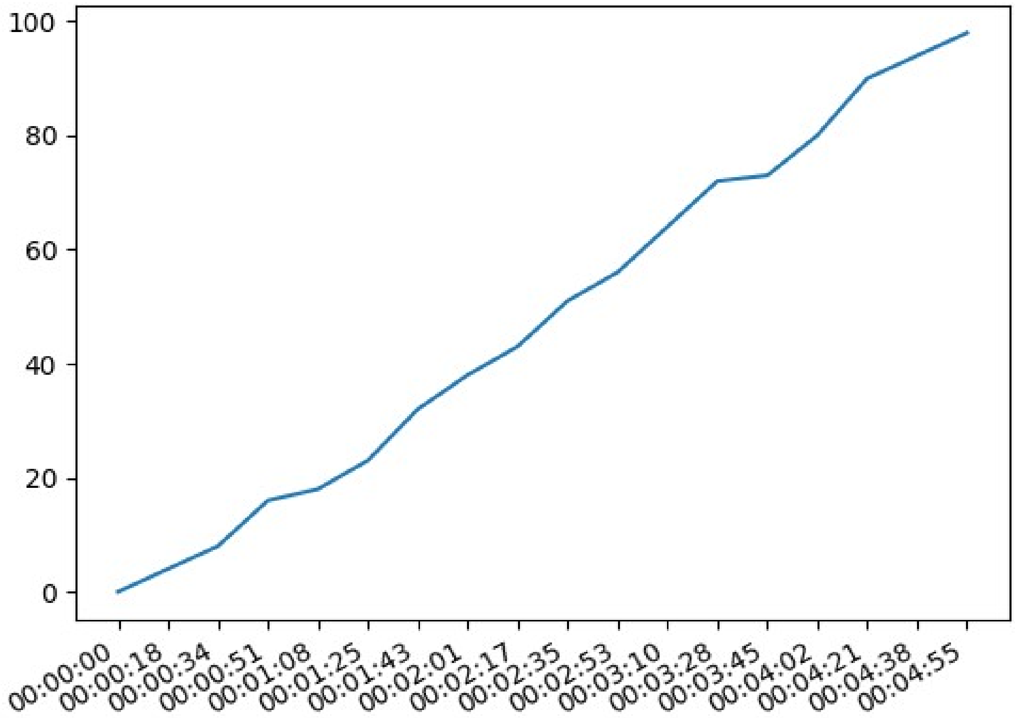


Explore map percentage



id	time	unknown	explored	obstacle	not set	percentage
4	00:00:00	37500	0	0	0	42
10	00:00:06	16461	21039	0	0	56
16	00:00:12	13631	23852	0	17	63
28	00:00:25	11162	26290	0	48	70
44	00:00:39	8048	29390	0	62	78
58	00:00:51	3396	33843	0	261	90
170	00:02:28	636	36070	0	794	96
346	00:04:58	401	34239	0	2860	91

Visited map percentage



ID	Time	visited	not visited	obstacle	percentage
1	00:00:00	0	37500	0	0
10	00:00:31	2384	35116	0	6
20	00:01:04	6448	31052	0	17
30	00:01:39	11100	26400	0	29
40	00:02:13	16401	21099	0	43
50	00:02:49	20805	16695	0	55
60	00:03:25	26866	10634	0	71
70	00:03:58	29588	7912	0	78
80	00:04:34	35388	2112	0	94
86	00:04:55	36873	627	0	98

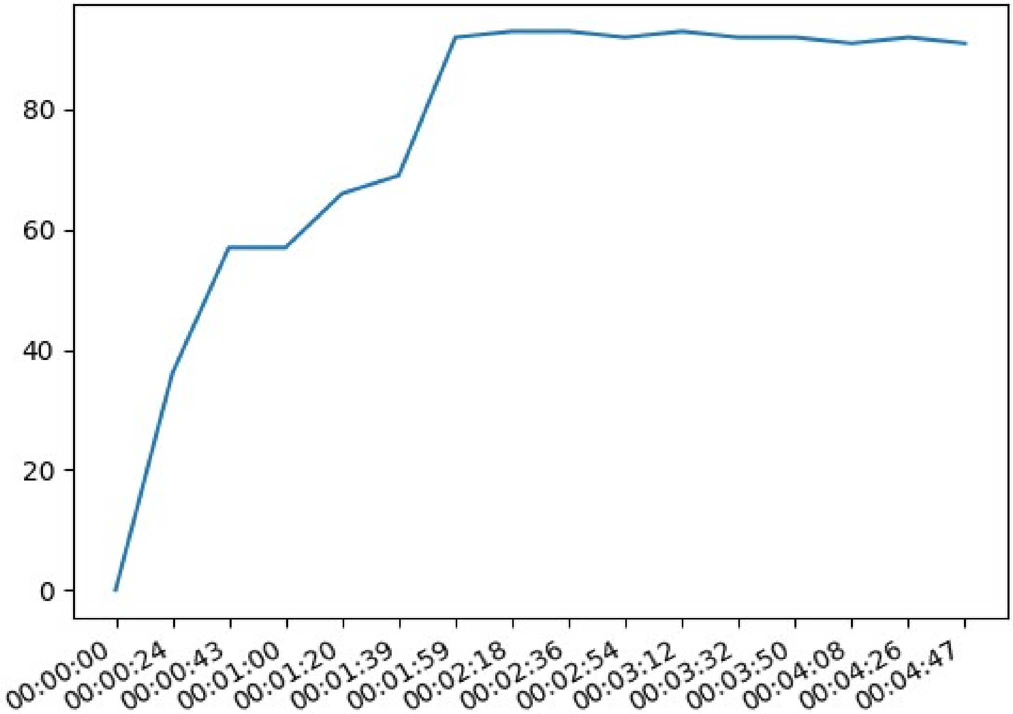
eStar - map with obstacle - simulation

Messages overview

Duration:	00:04:47	Messages sent	350	Message received	1076
Report	315	Ack	349	Nak	0
Count turn	92	Left	45	Right	47
Count forward	150	meters:	47	Count alive	378
Count backwards	0	meters:	0	Count Stop	59
Count scan	34	Angular correction	12	Report Error	0

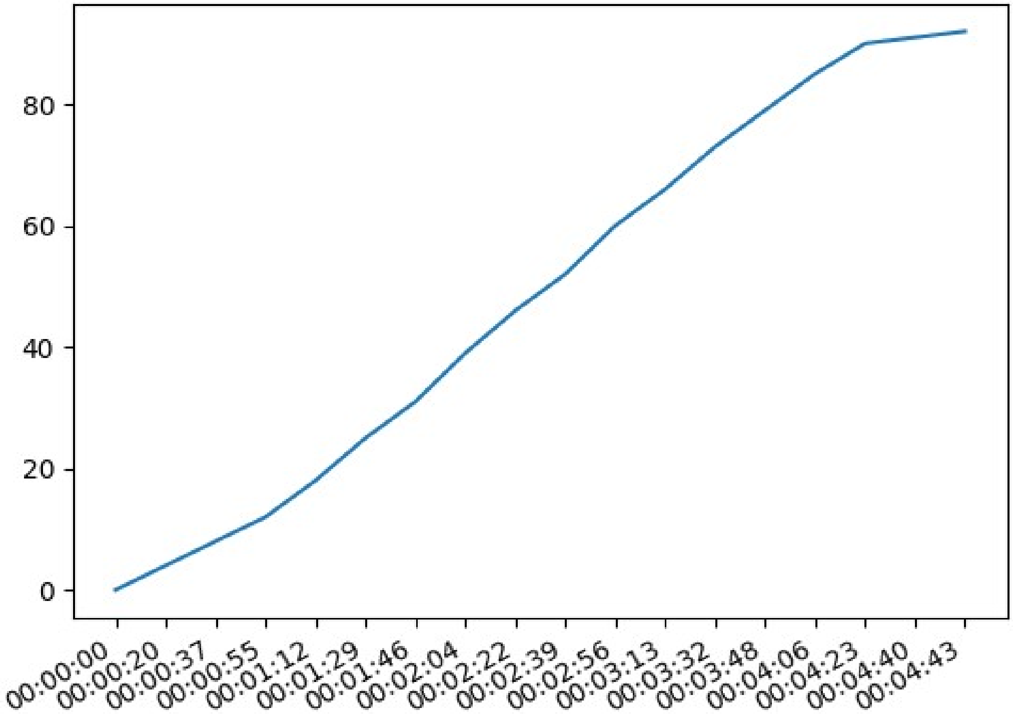
Maps overview

Explore map percentage



id	time	unknown	explored	obstacle	not set	percentage
4	00:00:00	37500	0	0	0	27
10	00:00:05	24952	12479	69	0	33
39	00:00:32	16871	20491	75	63	54
73	00:00:59	14339	22935	73	153	61
115	00:01:33	10969	26280	72	179	70
127	00:01:43	4114	32829	112	445	87
153	00:02:06	1623	35107	112	658	93
348	00:04:45	545	34430	23	2502	91

Visited map percentage



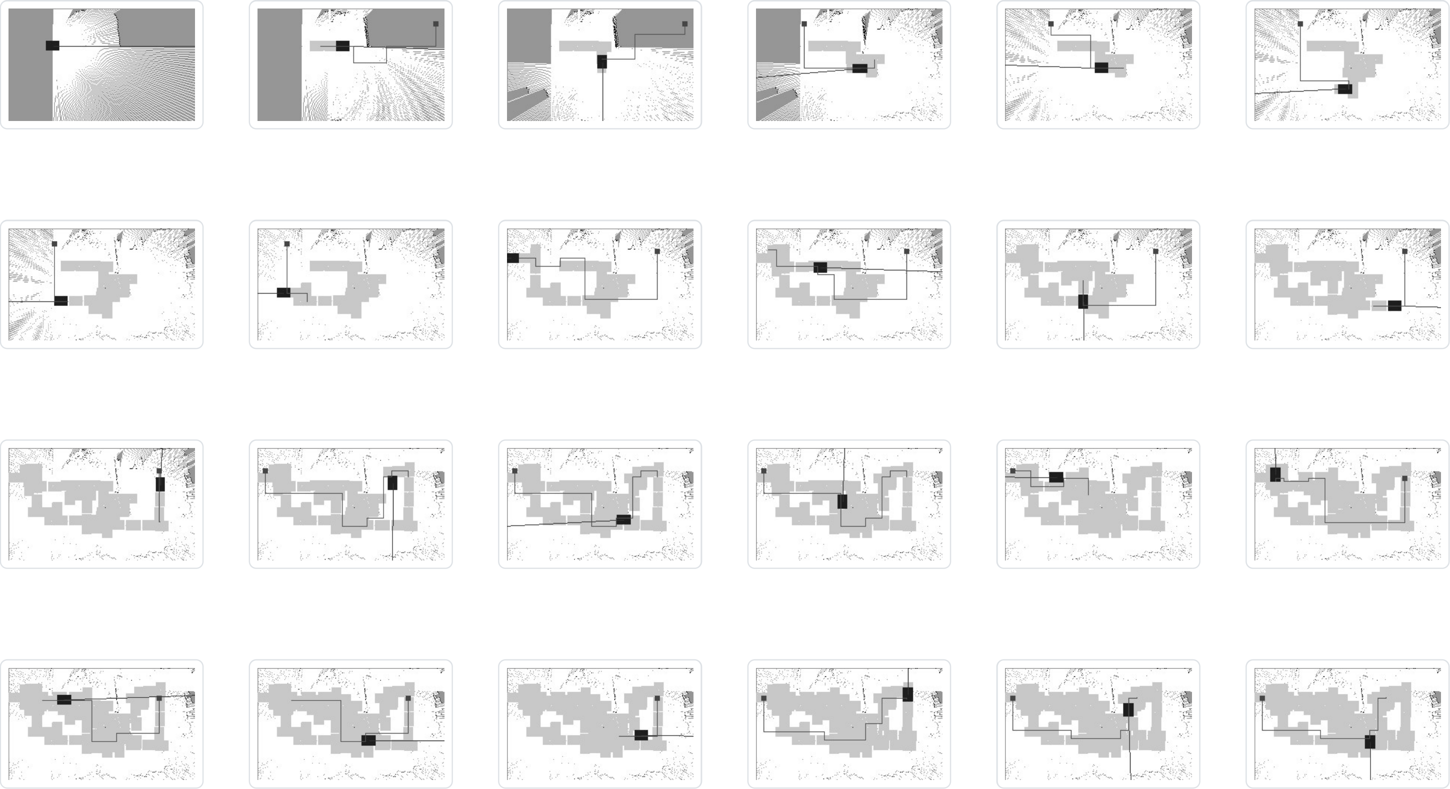
ID	Time	visited	not visited	obstacle	percentage
1	00:00:00	0	37477	23	0
10	00:00:34	3122	34355	23	8
20	00:01:09	6384	31093	23	17
30	00:01:42	11418	26059	23	30
40	00:02:18	16796	20681	23	44
50	00:02:52	21863	15614	23	58
60	00:03:28	26231	11246	23	69
70	00:04:02	31444	6033	23	83
80	00:04:36	34128	3349	23	91
82	00:04:43	34630	2847	23	92

Gridbase - map with obstacle - mower

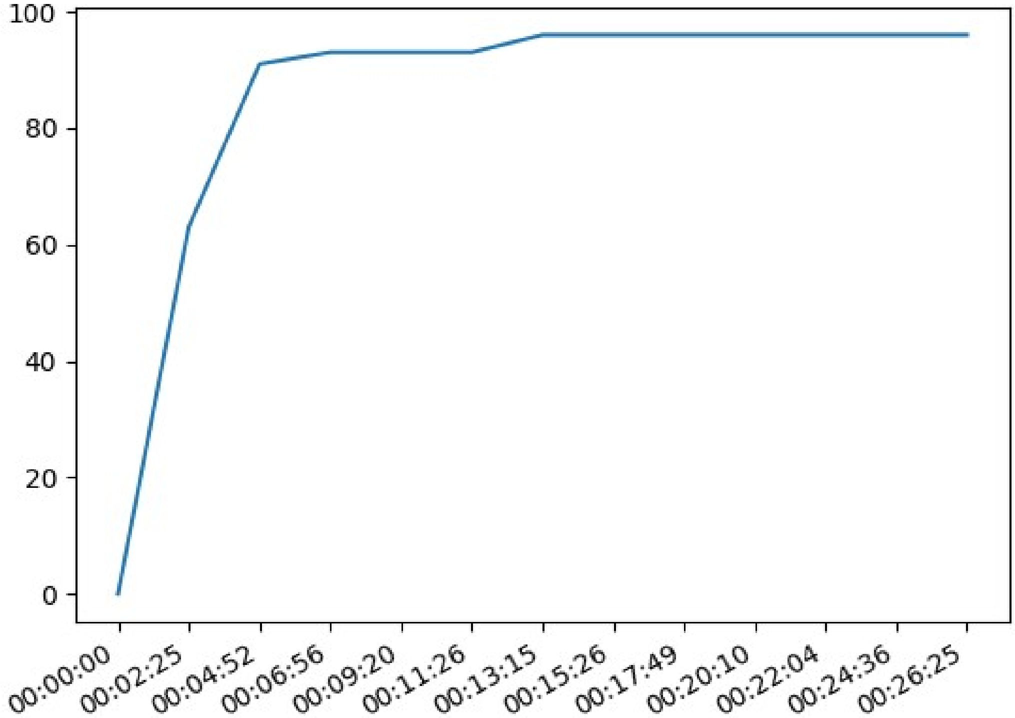
Messages overview

Duration:	00:26:29	Messages sent	260	Message received	1121
Report	205	Ack	232	Nak	25
Count turn	78	Left	32	Right	46
Count forward	102	meters:	31	Count alive	647
Count backwards	21	meters:	1	Count Stop	8
Count scan	19	Angular correction	29	Report Error	0

Maps overview

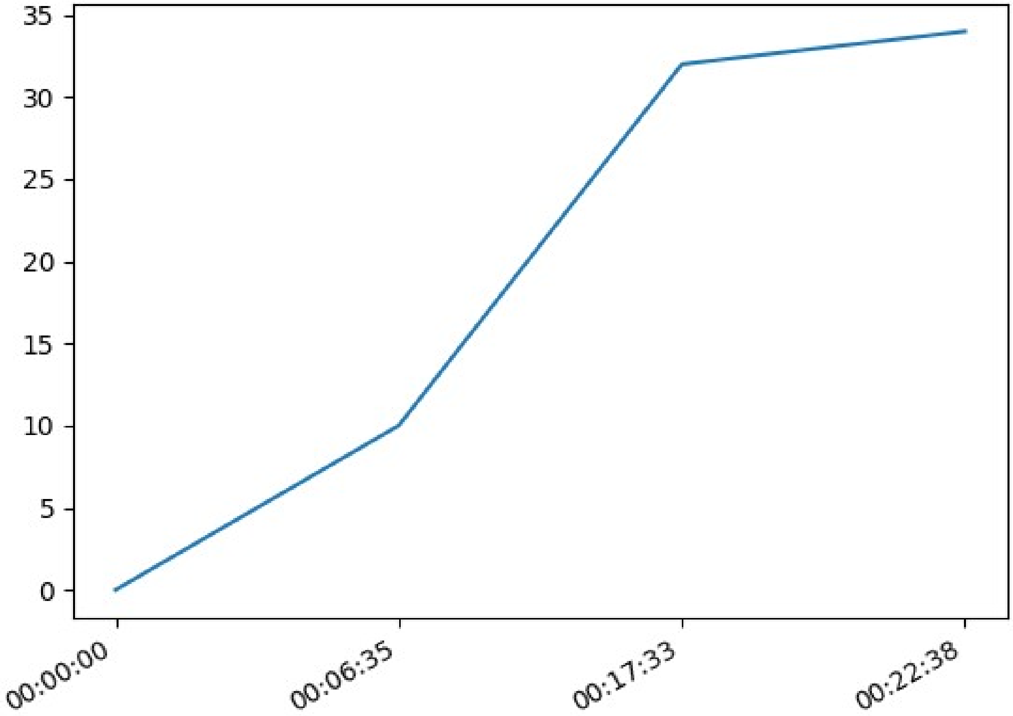


Explore map percentage



id	time	unknown	explored	obstacle	not set	percentage
0	00:00:00	37500	0	0	1	36
4	00:00:38	19104	18299	94	3	48
8	00:01:20	16310	21020	131	39	56
13	00:02:05	13632	23687	141	40	63
17	00:02:36	9861	27454	138	47	73
26	00:03:47	5788	31323	224	165	83
33	00:04:36	2997	34133	149	221	91
214	00:26:23	1163	36025	88	224	96

Visited map percentage



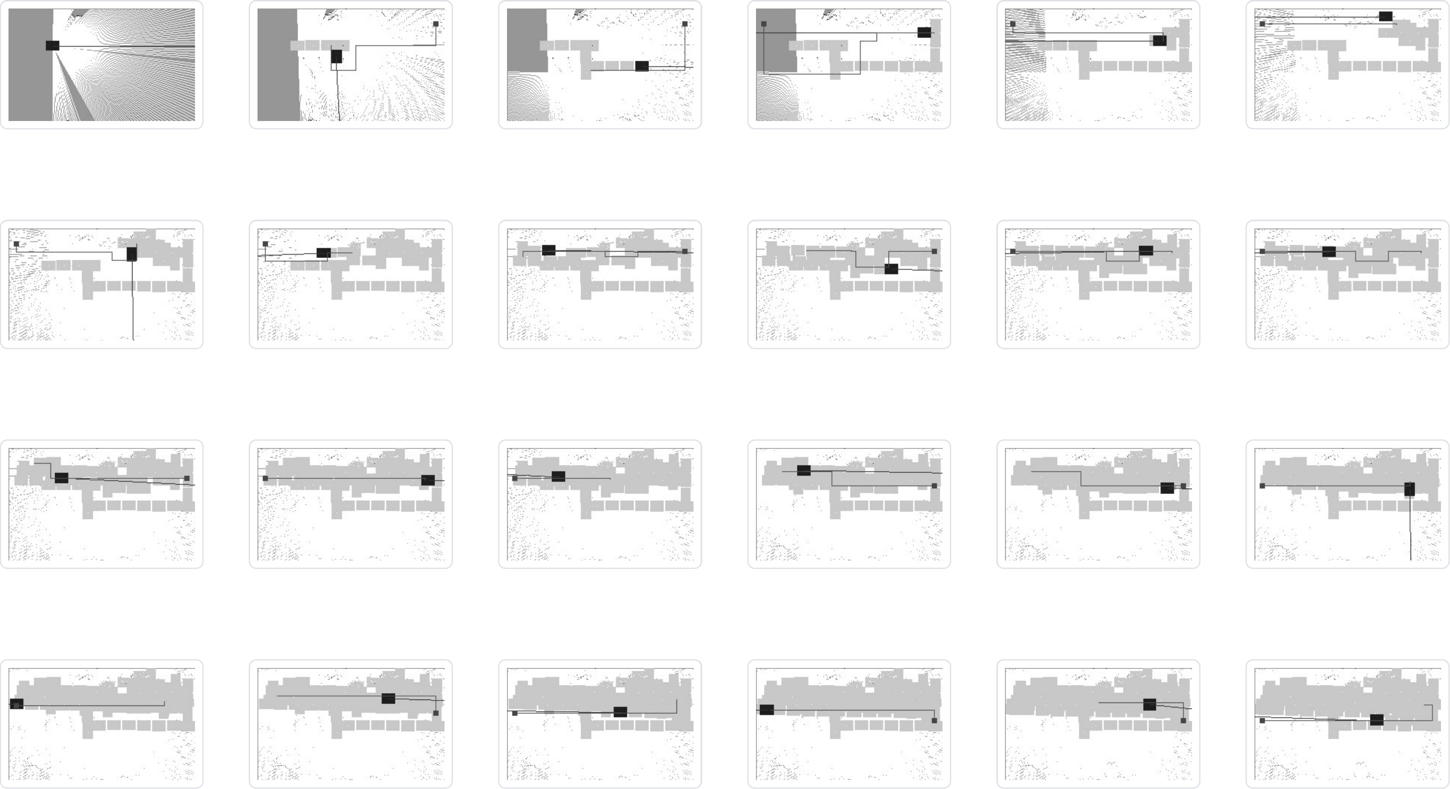
ID	Time	visited	not visited	obstacle	percentage
1	00:00:00	0	37412	88	0
10	00:03:54	2280	35132	88	6
20	00:08:03	5489	31923	88	14
30	00:12:16	8820	28592	88	23
40	00:21:32	12803	24609	88	34
41	00:22:38	12895	24517	88	34

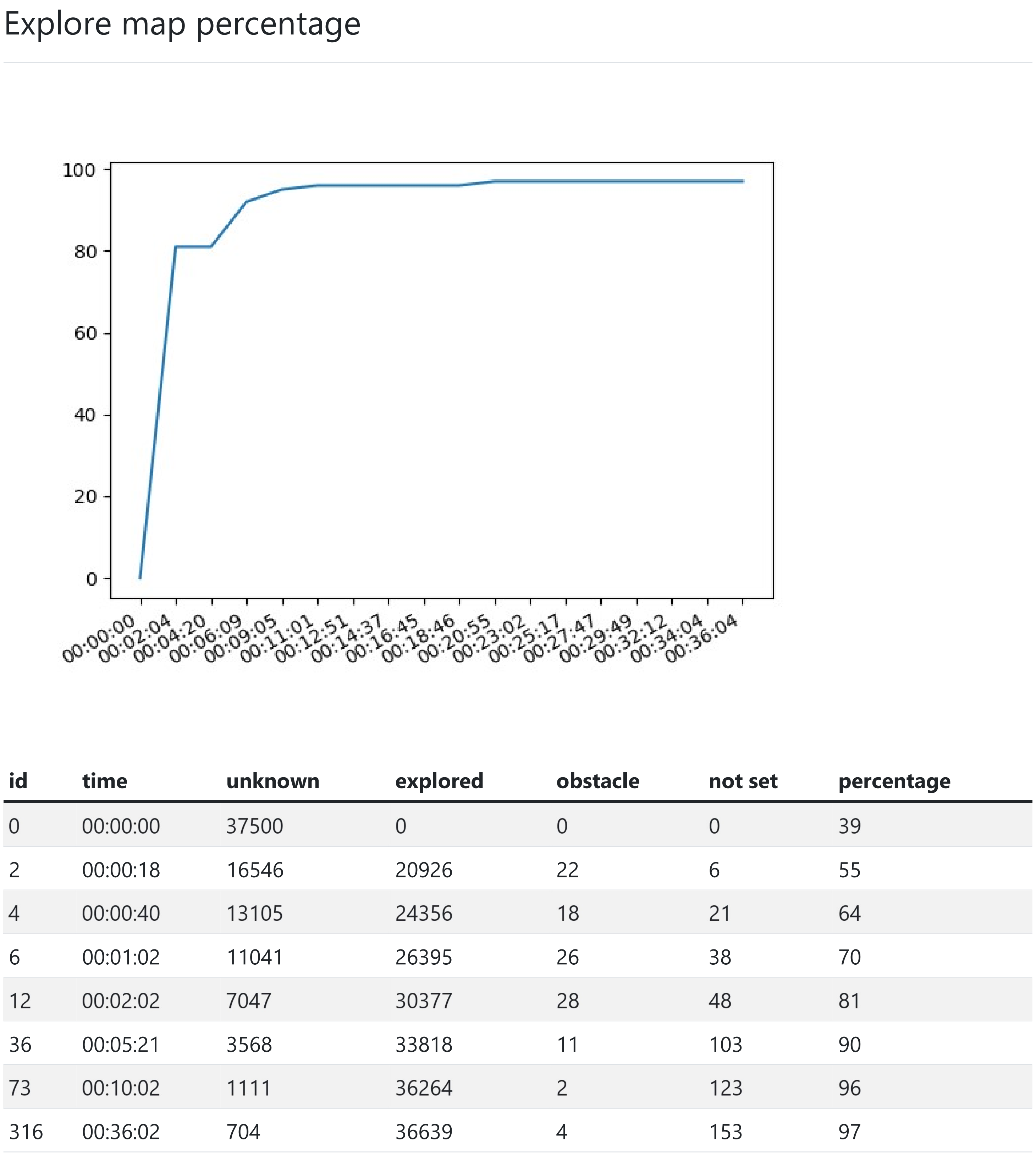
Gridbase - empty map - mower

Messages overview

Duration:	00:36:12	Messages sent	379	Message received	1308
Report	304	Ack	329	Nak	49
Count turn	87	Left	49	Right	38
Count forward	178	meters:	63	Count alive	611
Count backwards	38	meters:	4	Count Stop	3
Count scan	19	Angular correction	51	Report Error	0

Maps overview

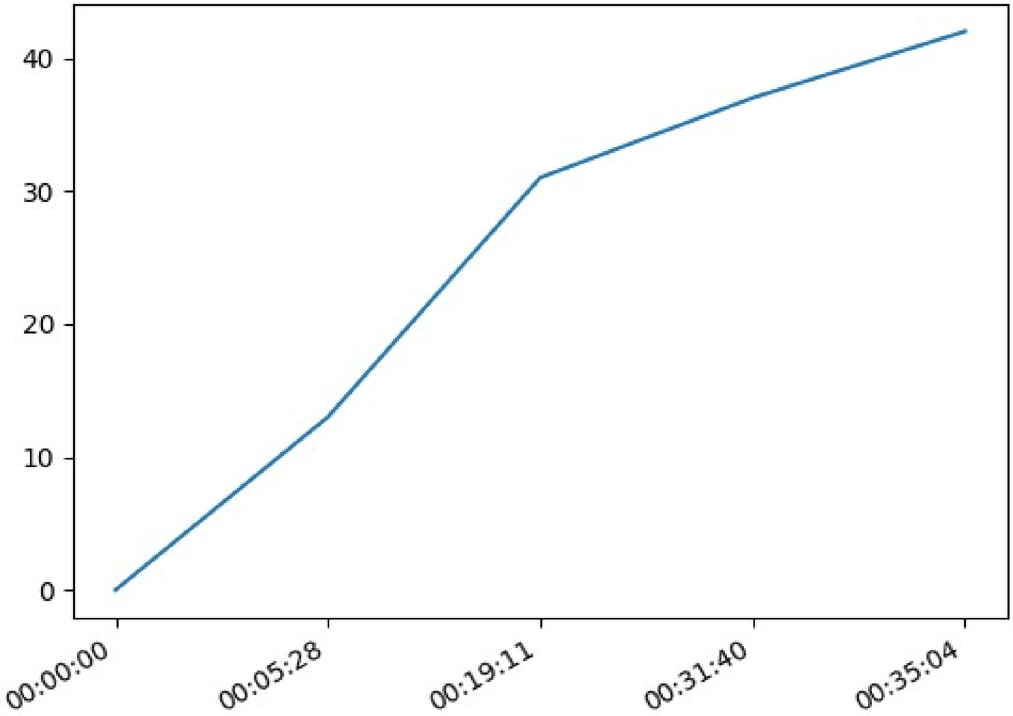




Explore map percentage

id	time	unknown	explored	obstacle	not set	percentage
0	00:00:00	37500	0	0	0	39
2	00:00:18	16546	20926	22	6	55
4	00:00:40	13105	24356	18	21	64
6	00:01:02	11041	26395	26	38	70
12	00:02:02	7047	30377	28	48	81
36	00:05:21	3568	33818	11	103	90
73	00:10:02	1111	36264	2	123	96
316	00:36:02	704	36639	4	153	97

Visited map percentage



ID	Time	visited	not visited	obstacle	percentage
1	00:00:00	0	37496	4	0
10	00:03:46	3610	33886	4	9
20	00:06:30	5588	31908	4	14
30	00:13:33	10971	26525	4	29
40	00:23:28	12514	24982	4	33
50	00:31:40	14160	23336	4	37
56	00:35:04	16075	21421	4	42

Bibliography

[1] Ercan U Acar and Howie Choset. Sensor-based coverage of unknown environments: Incremental construction of morse decompositions. *The International Journal of Robotics Research*, 21(4):345–366, 2002.

[2] Ercan U Acar, Howie Choset, Alfred A Rizzi, Prasad N Atkar, and Douglas Hull. Morse decompositions for coverage tasks. *The international journal of robotics research*, 21(4):331–344, 2002.

[3] Richard Bellman. On a routing problem. *Quarterly of applied mathematics*, 16(1):87–90, 1958.

[4] Thomas Bräunl. *Robot adventures in Python and C.* Springer, 2020.

[5] Howie Choset. Coverage for robotics–a survey of recent results. *Annals of mathematics and artificial intelligence*, 31(1):113–126, 2001.

[6] Andrew Cochrum, Joseph Corteo, Jason Oppel, and Matthew Seth. An autonomous lawnmower 'the manscaper'.

[7] Edsger W Dijkstra et al. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[8] Lester E Dubins. On curves of minimal length with a constraint on average curvature, and with prescribed initial and terminal positions and tangents. *American Journal of mathematics*, 79(3):497–516, 1957.

[9] Robert W Floyd. Algorithm 97: shortest path. *Communications of the ACM*, 5(6):345, 1962.

[10] Enric Galceran and Marc Carreras. A survey on coverage path planning for robotics. *Robotics and Autonomous systems*, 61(12):1258–1276, 2013.

[11] Nilsson Hart. Raphael; a formal basis for the heuristic determination of minimum cost path. *IEEE Transactions on Systems Science and Cybernetics SSC4*, 4(2), 1968.

[12] John Hunt. Gang of four design patterns. In *Scala design patterns*, pages 135–136. Springer, 2013.

[13] Dave Hershberger itan Marder-Eppstein, David V. Lu. Cost map 2d. `http://wiki.ros.org/`, 2018.

[14] Vladimir J Lumelsky, Snehasis Mukhopadhyay, and Kang Sun. Dynamic path planning in sensor-based terrain acquisition. *IEEE Transactions on Robotics and Automation*, 6(4):462–472, 1990.

[15] Hans Moravec and Alberto Elfes. High resolution maps from wide angle sonar. In *Proceedings. 1985 IEEE international conference on robotics and automation*, volume 2, pages 116–121. IEEE, 1985.

[16] Robin R Murphy. *Introduction to AI robotics*. MIT press, 2019.

[17] Dennis M Ritchie, Brian W Kernighan, and Michael E Lesk. *The C programming language*. Prentice Hall Englewood Cliffs, 1988.

[18] Mobile robotics C++ librariesa. Probabilistic motion models. `https://www.mrpt.org/tutorials/programming/odometry-and-motion-models/probabilistic_motion_models`, 2021. [Online; accessed 7-November-2021].

[19] Andrei M Shkel and Vladimir Lumelsky. Classification of the dubins set. *Robotics and Autonomous Systems*, 34(4):179–202, 2001.

[20] Alexander Shvets. data class. `https://refactoring.guru/smells/data-class`.

[21] Junnan Song and Shalabh Gupta. epsilon-star: An online coverage path planning algorithm. *IEEE Transactions on Robotics*, 34(2):526–533, 2018.

[22] Bjarne Stroustrup. An overview of the c++ programming language. *Handbook of object technology*, 1999.

[23] Columbia University. Icc kinematics - lectures notes. `http://www.cs.columbia.edu/allen/F17/NOTES/icckinematics.pdf`. [Online; accessed 21-August-2021].

[24] Andrew Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE transactions on Information Theory*, 13(2):260–269, 1967.

[25] Andrew Walker. Dubins-curves: an open implementation of shortest paths for the forward only car. `https://github.com/AndrewWalker/Dubins-Curves`, 2008–.

[26] Wikipedia. Single board computer. `https://en.wikipedia.org/wiki/Single-board_computer`.

[27] Wikipedia. Bresenham's line algorithm. `https://en.wikipedia.org/wiki/Bresenham%27s_line_algorithm`, 2021. [Online; accessed 27-September-2021].

[28] Wikipedia. Lidar. `https://en.wikipedia.org/wiki/Lidar`, 2021. [Online; accessed 27-September-2021].

[29] Wikipedia. Observer pattern. `https://en.wikipedia.org/wiki/Observer_pattern`, 2021. [Online; accessed 27-September-2021].

[30] Wikipedia. Python. `https://en.wikipedia.org/wiki/Python_(programming_language)`, 2021. [Online; accessed 27-September-2021].