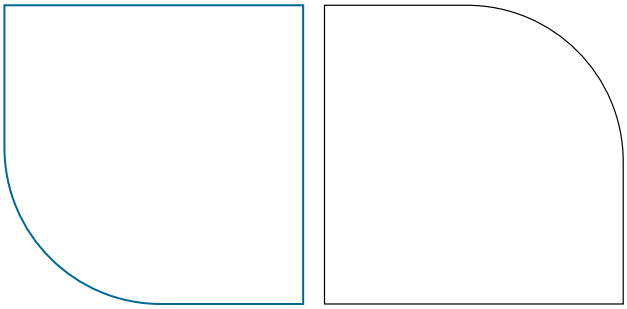
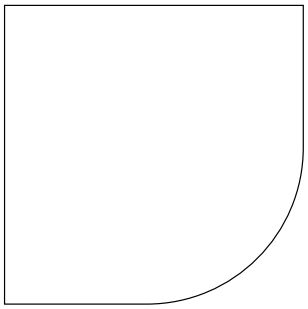


Ensta Bretagne
2, rue François Verny
29806 Brest cedex
France
Tel +33 (0)2 98 34 88 00
www.ensta-bretagne.fr



Report

FISE 2024

May-August 2023

Assistant Engineer Internship

Carl Von Ossietzky University, Oldenburg, Germany

Hofmann Hugo
hugo.hofmann@ensta-bretagne.org

Résumé

Dans le cadre de ma formation à l'ENSTA Bretagne j'ai été amené à réaliser un stage à l'étranger en deuxième année en assistant ingénieur. J'ai eu la chance d'être accepté dans l'université d'Oldenburg en Allemagne, où j'ai pu travailler en autonomie sur un robot, le Turtlebot3 Waffle Pi. Mon objectif durant ce stage était de mettre en place le robot, et d'établir une stratégie permettant de le contrôler efficacement et avec précision, tout en le rendant en partie autonome. Je m'attarderai dans ce rapport sur les différents aspects abordés lors de ce stage, qu'il s'agisse du contrôle, de l'estimation de l'état du système, de la planification de trajectoires ainsi que de l'architecture des programmes développés. Je présenterai d'abord le robot et ses capteurs, ce qui m'amènera à formuler une mission à réaliser qui me servira d'objectif pour ce stage. Je détaillerai ensuite le travail fait sur le contrôle du robot, dans le but de le rendre capable de suivre une trajectoire en un temps donné. Je parlerai ensuite d'observation et d'estimation d'état en présentant ma démarche visant à améliorer la connaissance de l'état du robot et notamment de sa vitesse. Je pourrai ensuite aborder la question de la planification au travers des programmes de recherche de trajectoires et aux différents algorithmes utilisés, en passant par la détection d'objets et l'exploration autonome de l'environnement.

Abstract

As part of my training at ENSTA Bretagne, I was asked to do an internship abroad in my second year as an assistant engineer. I was lucky enough to be accepted at the University of Oldenburg in Germany, where I was able to work autonomously on a robot, the Turtlebot3 Waffle Pi. My aim during this internship was to set up the robot, and establish a strategy for controlling it efficiently and accurately, while making it partially autonomous. In this report, I will focus on the various aspects of the internship, including control, system state estimation, trajectory planning and program architecture. I will first present the robot and its sensors, which will lead me to formulate a mission to be carried out, which will serve as my objective for this internship. I'll then detail the work done on controlling the robot, with the aim of making it capable of following a given trajectory in a given time. I'll then talk about observation and state estimation, presenting my approach to improving knowledge of the robot's state, and in particular its speed. I'll then move on to the question of planning, through trajectory search programs and the various algorithms used, to object detection and autonomous exploration of the environment.

Contents

Note: The online documentation is also available here:

https://waffle-pi-project.readthedocs.io/en/latest/

Chapter 1

Introduction

1.1 The *Turtlebot3 Waffle Pi*

The provided robot is a *Turtlebot3 Waffle Pi* robot, a robotics platform with open-source hardware/software developed by the Korean company *Robotis*.

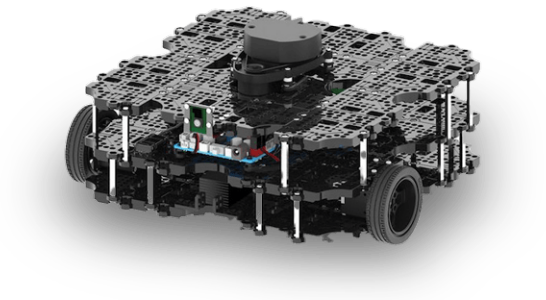
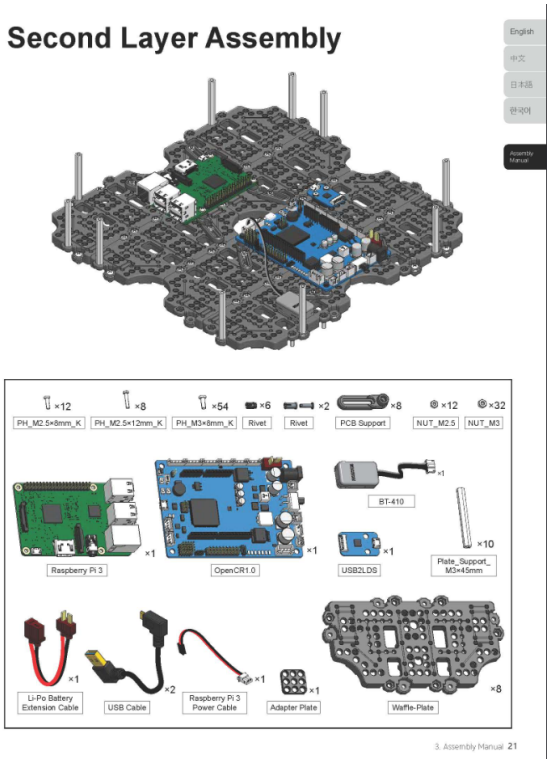


Figure 1.1 – *The Turtlebot3 Waffle Pi*

The robot comes with all parts ready to be assembled and a full documentation explaining how to build it



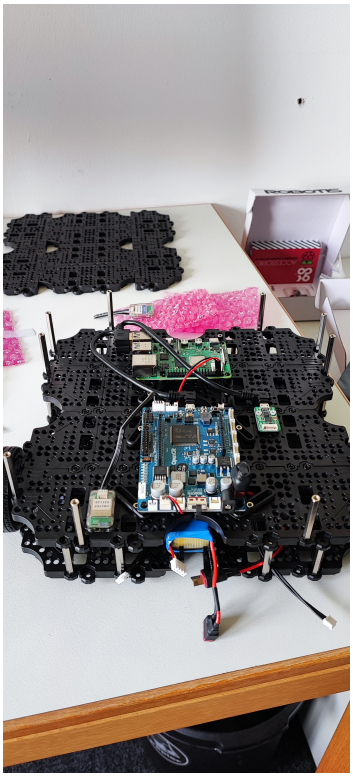


Figure 1.2 – *Building the robot*

and set it up[1]. My first task was therefore to follow these instructions as seen in Figure 1.2 to get ready to move onto the actual work.

1.1.1 Available Hardware

The robot is mainly composed of :

- 1x *Raspberry Pi 4 2GB*
- 1x *Raspberry Pi* camera module
- 1x *OpenCR* controller board
- 1x *LDS-02* LIDAR Scanner [Fig.1.4]
- 2x *Dynamixel XM430-W210* servomotors

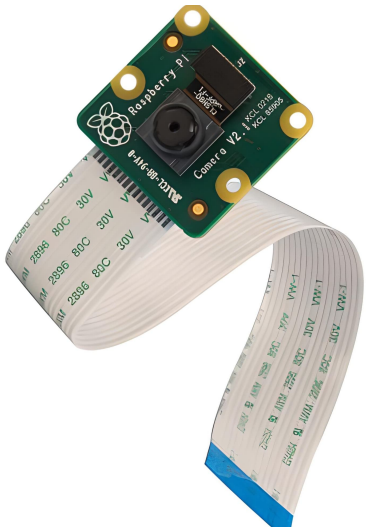


Figure 1.3 – Camera Module

Taking into account all of the sensors internal to some of the components, we get the following list of available sensors to be used :

- 1x *LDS-02* LIDAR Scanner [Fig.1.4]
- 2x Wheel Encoders (in the servomotors)
- 1x *Raspberry Pi* Camera Module
- 1x 9-axis IMU (within the *OpenCR*)

Figure 1.4 – LDS-02 scanner

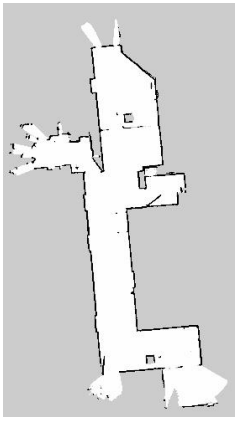
1.1.2 Available Software

The *Turtlebot3 Waffle Pi* comes with various tools to start using the robot almost as soon as it has been assembled. First, it is compatible with ROS and ROS2. In my case, I chose the ROS2 *Foxy* version to work on. The provided *Ubuntu* image for the SD card contains all the ROS2 packages needed to launch the robot, including the drivers that allow the user to access the sensors and control the servos (apart from the camera module that was not detected [insert explaination]) through dedicated ROS2 topics (ex: `\cmd_vel` to send the commands). One interesting thing about the drivers is that they already take care of processing incoming sensor data. For example, the odometry is constantly taken into account and integrated, so that we already have access to an $\begin{bmatrix} x \\ y \end{bmatrix}$ estimation of the position of the robot at any time.

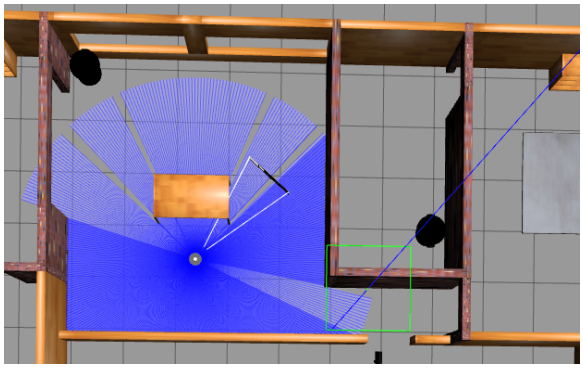
The provided software goes even further and brings some more very useful features :

- *Gazebo* Simulation
- Cartographer (or SLAM) Node
- Navigation Node
- Keyboard Teleoperation Node

These tools constitute a great base to work on, just like the usual ROS2 tools I will also be using, such as *RVIZ2* or *RQT*.



(a) Map made with the Cartographer Node



(b) *Gazebo* Simulation

Figure 1.5 – A few examples of provided software

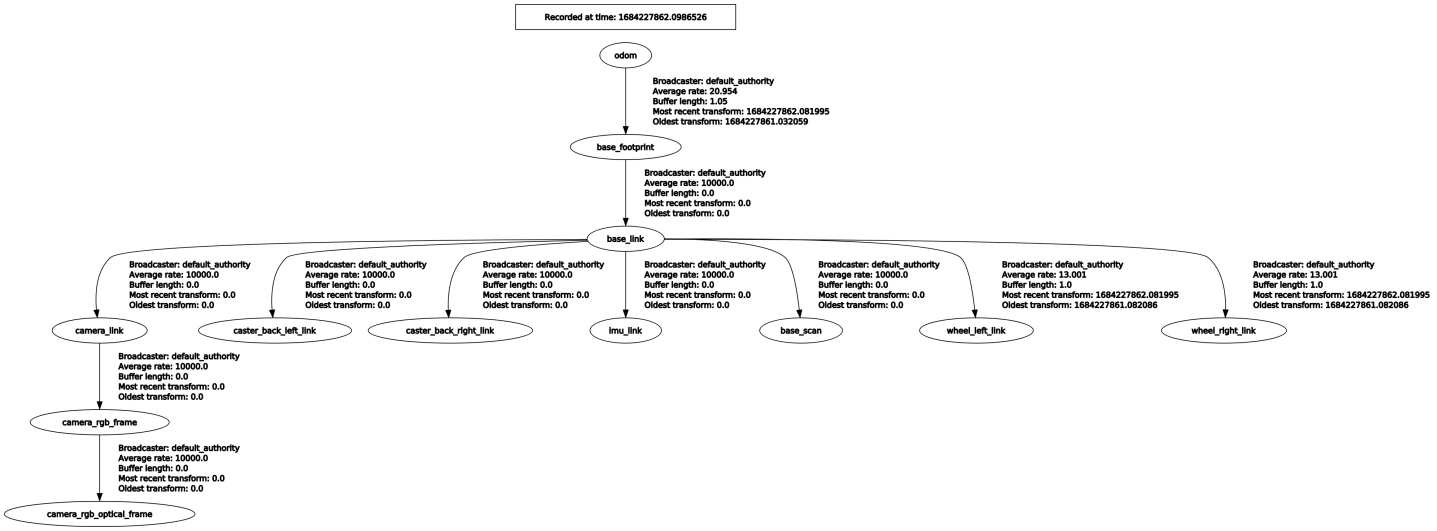


Figure 1.6 – ROS2 Frames (TF) of the robot, updated in real time

To use the robot and program it, I will need a first estimation of its state. This is actually already taken care of by the drivers. For instance, Figure 1.6 shows the "TF Tree": how the drivers automatically publish the *TF* transforms that I can then use to simply deduce the position of the robot. This is done by subscribing to the TF transforms topic and extracting the components (`translation` and `rotation`) of the transform between our base or world frame (`odom`) and the frame that represents our robot (`base_link`, the frame located in the center of the wheel axis). This allowed me to start working immediately to obtain some first results.

When it comes to the programs I will be developing during this internship, ROS2 allows me to choose between *C++* and *Python*. I chose the latter, since it is significantly easier to write and debug, although slower. Thus translating the code to *C++* could prove to be a challenge in itself but would also eventually bear its fruits when it comes to tackling some speed issues, but *Python* simply seems like the best option to start with.

1.2 Objective

For this internship I had the chance of being quite free when it comes to the exact nature of my work with the *Turtlebot3 Waffle Pi* . Indeed, while the general idea was for me to set up the robot and work on its control, I got to choose what would be my specific objective. I immediately thought of starting with implementing a simple controller and an observer with the methods that I had learnt so far, since this robot was a great opportunity for me to test these in practice. Then the various sensors, especially the LIDAR (and the cartographer node that comes with it) and the camera convinced me to try to use them for more ambitious, interesting and even fun missions, rather than simply working on an intermediate-level control. Progressively as I worked on the first few scripts to learn how to use the robot, my goal became to be able to give the robot several behaviours and even to make it at least partially autonomous in its path planning as well. With this in mind and the sensors available I eventually came up with this objective that I thought would make for an interesting internship: developing a high-level control for the *Turtlebot3 Waffle Pi* in the form of a user interface and a mission planning program allowing the robot to carry out several missions. One of these, probably the most interesting one, being:

"In order to become a true German citizen, one must overcome the challenge of returning water bottles to get the cherished Pfand (deposit) back". With this goal in mind, to what extent can one turn the *Turtlebot3 Waffle Pi* into an autonomous bottle finder ?

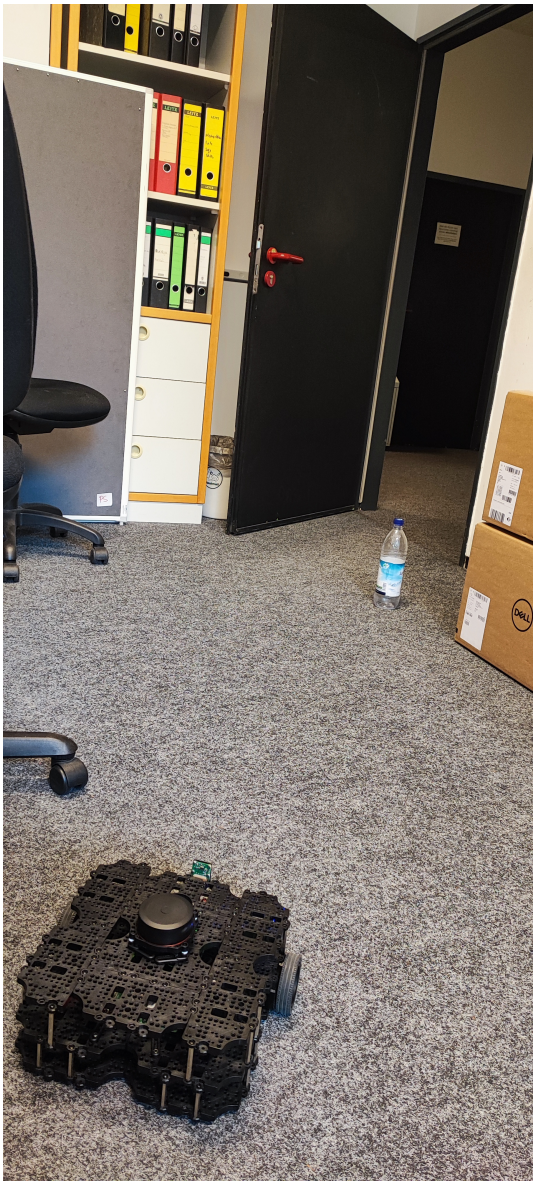


Figure 1.7 – The robot getting ready to explore

Chapter 2

Control

When trying to make the robot more autonomous, the first thing that comes to mind is working on giving the robot the right commands so that it follows a certain trajectory. As explained later further in the report, this will become the foundation of the other developed programs, as most of the desired features involve following a certain path. One thing we can take into account is that the robot's sensors and drivers are already good enough to work on controlling the robot without worrying about accurately estimating the position and heading of the robot, and as explained earlier, we have an easy access to the position and heading thanks to *TF* transforms. This led me to work on a controller, starting with an appropriate cinematic model for the *Turtlebot3 Waffle Pi* .

2.1 Cinematic Model

In order to control the robot, we first need to analyze its behaviour and what type of commands are to be sent. The robot in itself is quite obviously rather simple in its way of moving. It has two servomotors on the rear end that turn and make it move. At first instinct we may be tempted to find the relationship between the angular speeds ω_l and ω_r of the wheels and the general linear v and angular speed θ. Using Varignon's theorem, we would easily get :

$$\begin{cases} v = R\frac{\omega_l+\omega_r}{2} \\ \dot{\theta} = R\frac{\omega_r-\omega_l}{2L} \end{cases} \quad \text{Where } R \text{ is the radius of the wheels, } L \text{ half the robot's width} \tag{2.1}$$

Although this is interesting, the provided software takes care of the low-level control of the wheels and does not expect ω_l or ω_r. Instead, on the ROS2 topic called `cmd_vel`, the robot expects a *Twist* message,

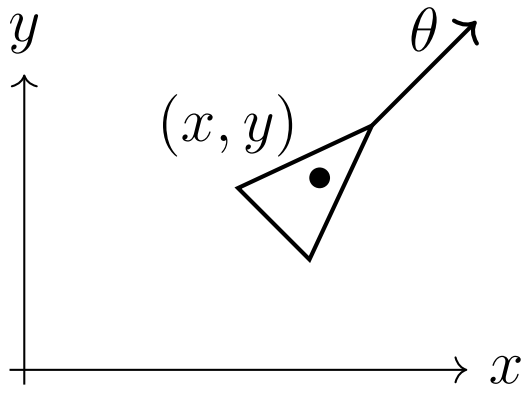


Figure 2.1 – Cinematic Model

that is, the linear and angular speeds $\begin{bmatrix} v \\ \dot{\theta} \end{bmatrix}$. Therefore, these become the commands that our controller needs to compute, as we will not have to deal with the individual wheels but instead will rather reason in terms of acceleration and heading change. This corresponds to the classic cinematic model of the *Dubins Car*, hence the corresponding cinematic equations :

$$\begin{cases} \dot{x} = u_1 \cos(\theta) \\ \dot{y} = u_1 \sin(\theta) \\ \dot{\theta} = u_2 \end{cases} \tag{2.2}$$

where x and y are the robot's coordinates in the horizontal plane and θ its heading. That way we successfully obtained our cinematic model with a state vector X :

$$\dot{X} = f(X, u) \quad \text{where} \quad \begin{cases} X = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix} \\ u = \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} \end{cases} \tag{2.3}$$

However, my idea was to use a specific method called *Feedback Linearization* to control the robot, a method I will explain in details further into this chapter. Although the previous state equations found in 2.2 are

a good start, we need to slightly change them for the method to work properly. Instead, we will consider the new system :

$$\begin{cases} \dot{x} = z\cos(\theta) \\ \dot{y} = z\sin(\theta) \\ \dot{z} = c_1 \\ \dot{\theta} = u_2 \end{cases} \tag{2.4}$$

The is simply a different version of the same equations, where we introduced z, a new state variable that represents the robots's linear speed and replaces u_1. The new command introduced here is c_1, the derivative of z, meaning we now have a system where we rather control the acceleration rather than the speed in itself, whereas controlling the heading remains as simple as before. It goes without saying that this new set of equations is "virtual", meaning the new command vector $u = \begin{bmatrix} c_1 \\ u_2 \end{bmatrix}$ is not the set of commands the robot will actually use, but rather $\begin{bmatrix} z \\ u_2 \end{bmatrix}$.

2.2 Trajectory definition

Now that we have a suitable cinematic model, let's define the trajectory we would like to achieve. A simple answer is to follow a trajectory of the form :

$$w_d(t) = \begin{bmatrix} x_d(t) \\ y_d(t) \end{bmatrix} \tag{2.5}$$

that represents the path defined by a function in the horizontal plane, as illustrated in Figure 2.2 :

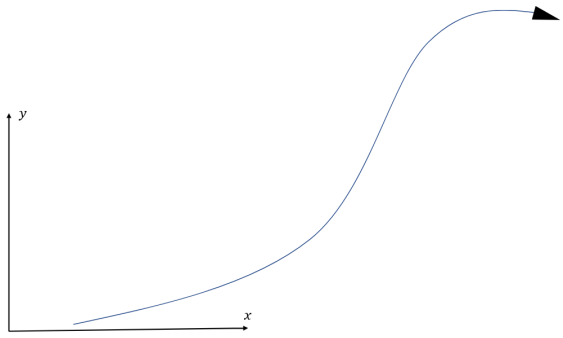


Figure 2.2 – Example of a trajectory

In order to first test the controller, I chose Bézier curves as the $w_d(t)$ functions. By using $n+1$ control points $P_0, P_1, P_2, ...P_n$, we're able to easily obtain some interesting trajectories defined by the formula :

$$B(t) = \sum_{i=0}^{n} \binom{n}{i} (1-t)^{n-i} t^i P_i \quad \text{with} \quad 0 \leq t \leq 1 \tag{2.6}$$

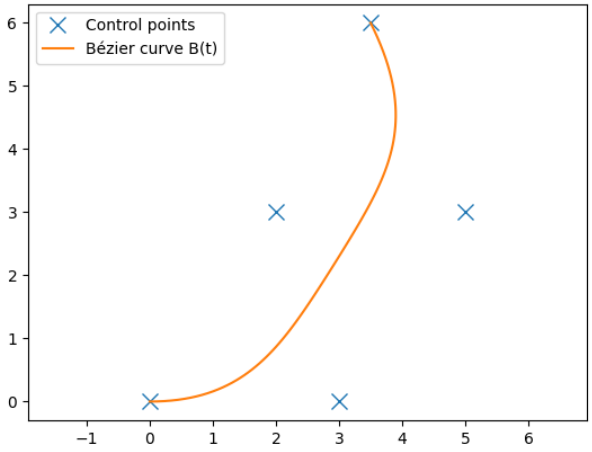


Figure 2.3 – Example of a Bézier curve

Later in the report, I will use Bézier curves again as part of a *B-splines*, but for now these Bézier curves will suffice, as long as they do not rely on too many control points, since the degree of the polynomial $B(t)$ is equal to the number of the points used. The fact that it is a polynomial function also has the advantage of allowing one to easily derive it as much as needed, which will prove useful for the controller, since not only will we need the position w_d but also the speed $\dot{w}_d$ and acceleration $\ddot{w}_d$.

As visible in the definition of the function in Equation 2.6, the timescale of the Bézier function is normalized, that is, the trajectory is defined to last 1 second only (if the time unit used is the second). However in our case, we might want to choose a duration T. This is easily done by a variable change in the function, which gives us :

$$\begin{cases} w_d(t) = B(\frac{t}{T}) \\ \dot{w}_d(t) = \dot{B}(\frac{t}{T}) \cdot \frac{1}{T} \\ \ddot{w}_d(t) = \ddot{B}(\frac{t}{T}) \cdot \frac{1}{T^2} \end{cases} \tag{2.7}$$

In practice, this is easily added in Python. I implemented a new object class called `BezierCurve` that is initialized with all the control points. Its `.eval(t)` method returns all $w_d, \dot{w}_d, \ddot{w}_d$ evaluated at time t.

2.3 Controller - Feedback Linearization

Now that the objective has been defined and that we have a model that describes how the robot moves, we can now focus on building a controller. As quickly mentioned before, I we will use a method called *Feedback Linearization*[2] to compute the vector $\begin{bmatrix} c_1 \\ u_2 \end{bmatrix}$. Once this is done we will only need to integrate z with c_1 in order to send the commands.

To start off, we want to control the robot's position :

$$\begin{bmatrix} x \\ y \end{bmatrix}$$

Let us derive this position according to our state equations 2.4 until the commands appear :

$$\begin{bmatrix} \dot{x} \\ y \end{bmatrix} = \begin{bmatrix} \dot{x} \\ \dot{y} \end{bmatrix} = \begin{bmatrix} z\cos(\theta) \\ z\sin(\theta) \end{bmatrix} \tag{2.8}$$

$$\begin{bmatrix} \ddot{x} \\ y \end{bmatrix} = \begin{bmatrix} \ddot{x} \\ \ddot{y} \end{bmatrix} = \begin{bmatrix} \dot{z}\cos(\theta) - z\dot{\theta}\sin(\theta) \\ \dot{z}\sin(\theta) + z\dot{\theta}\cos(\theta) \end{bmatrix} = \begin{bmatrix} c_1\cos(\theta) - zu_2\sin(\theta) \\ c_1\sin(\theta) + zu_2\cos(\theta) \end{bmatrix} \tag{2.9}$$

We can also write Equation 2.9 as :

$$v = \begin{bmatrix} \ddot{x} \\ y \end{bmatrix} = \underbrace{\begin{bmatrix} \cos(\theta) & z\sin(\theta) \\ \sin(\theta) & z\cos(\theta) \end{bmatrix}}_{A(X)} \cdot \begin{bmatrix} c_1 \\ u_2 \end{bmatrix} \tag{2.10}$$

where $A(X)$ is matrix that depends on state vector $X = \begin{bmatrix} x \\ y \\ z \\ \theta \end{bmatrix}$. $A(X)$ is invertible as long as $z \neq 0$, that is, when the robot's speed is not null. In that case, we have :

$$A^{-1} = \begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\frac{\sin(\theta)}{z} & \frac{\cos(\theta)}{z} \end{bmatrix}$$

Now we want to make sure that the error $e = w_d - v$ converges to 0 by following a second order linear differential equation of the form:

$$k_1 e + k_2 \dot{e} + \ddot{e} = 0 \tag{2.11}$$

The coefficients k_1 and k_2 can be chosen to change the behaviour of system, since they define the poles of the differential equation. For instance, if we want the poles p_1 and p_2 to both take the value -1 (since the poles's real component must be strictly negative for the system to be stable):

$$p_1 = p_2 = -1 \Rightarrow \begin{cases} k_1 = 1 \\ k_2 = 2 \end{cases} \tag{2.12}$$

Deducing v from Equation 2.11 and using Equation 2.10, we obtain the following controller:

$$\begin{cases} \begin{bmatrix} c_1 \\ u_2 \end{bmatrix} = \begin{bmatrix} \cos\theta & \sin\theta \\ -\frac{\sin\theta}{z} & \frac{\cos\theta}{z} \end{bmatrix} v \\ v = \underbrace{k_1 \left(w_d(t) - \begin{bmatrix} x \\ y \end{bmatrix} \right)}_{\text{Proportional error}} + \underbrace{k_2 \left(\dot{w}_d(t) - \begin{bmatrix} \dot{x} \\ \dot{y} \end{bmatrix} \right)}_{\text{Derivative error}} + \ddot{w}_d(t) \end{cases} \tag{2.13}$$

Finally, by integrating z from the acceleration command c_1, we get the following controller:

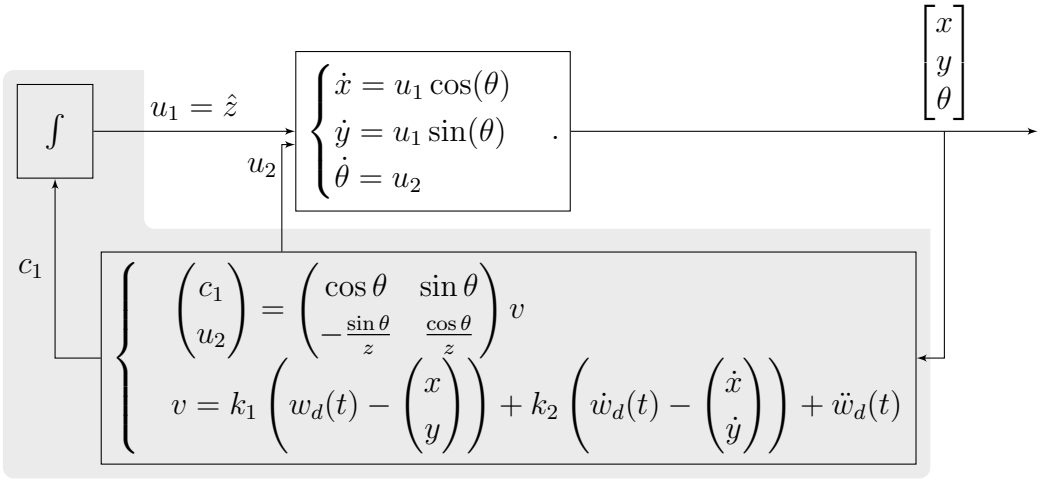


Figure 2.4 – Diagram of the full controller

2.4 Implementation

After designing the controller, the next step is to implement it. We will use the previous controller equations and the implemented Bézier curve class to create a ROS2 node that will serve as the controller of our system. Inside the node class we call a timer associated with a *timer callback function*. At each call of this function, we follow the same pattern. We retrieve the latest information about the robot's state. We call the Bézier curve function to obtain $w_d, \dot{w}_d, \ddot{w}_d$. We then call the `controller()` function which uses the previous equations to get c_1 and u_2. After integrating z with c_1, all that is left to do is to make sure the commands z and u_2 stay within a defined interval $[-value_{max}, +value_{max}]$ so that we respect the physical limitations of the servomotors[1]. We end the loop iteration by sending the commands to the drivers by publishing a *Twist* message of the form

$$\begin{cases} \vec{V} = z \cdot \vec{x} \\ \vec{\Omega} = u_2 \cdot \vec{z} \end{cases}$$

on the `\cmd_vel` topic.

```
function timer_callback():
    x, y, θ ← observe()
    t ← time()
    w_d, ẇ_d, ẅ_d ← bezier.eval(t)
    X, Ẋ ← [x; y], [z cos(θ); z sin(θ)]
    c_1, u_2 ← controller(X, Ẋ)
    z ← z + Δt . c_1
    z ← max(−z_max, min(z_max, z))
    u_2 ← max(−u_max, min(u_max, u_2))
    send_command(z, u_2)
```

Figure 2.5 – Controller pseudo-algorithm

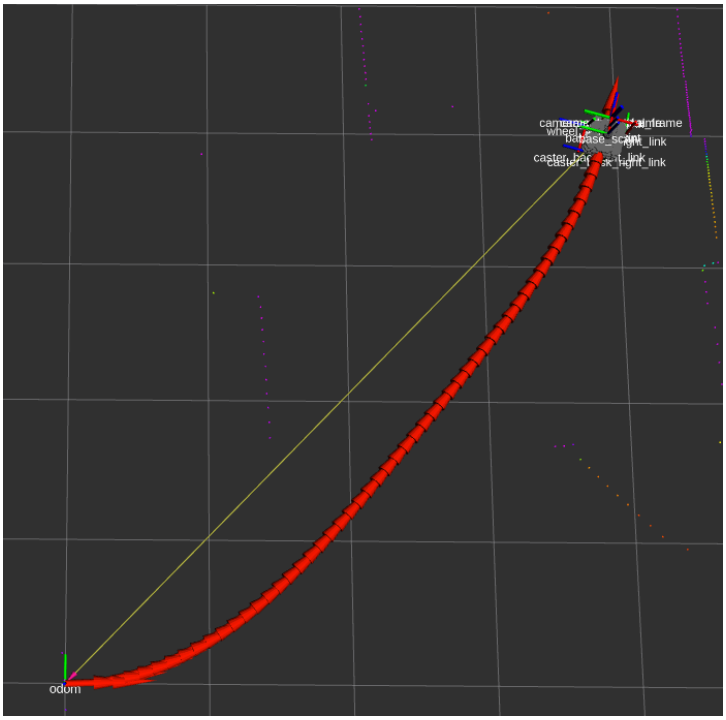
Since we do not retrieve any information about z, we need to manually initialize it. For the reasons explained earlier when designing the controller, it is better not to set it to 0 but rather to a small value if we want the robot to start from a standstill, unless we take care of that problematic case; for example, by making the robot move forward when $z = 0$.

Note that the function `observe()` for now does nothing else other than retrieving the information directly given by the drivers, but as we move on, it can be modified so as to add an observer to better estimate not only the position and heading but also the speed z (instead of assuming the robot will actually go as fast as the command orders it to).

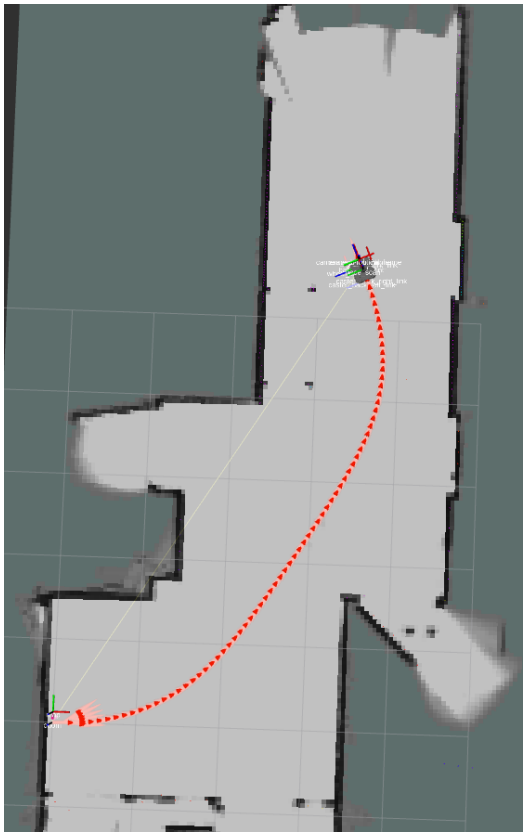
2.5 First results

Now that the controller is implemented, I can test it on the robot by making it follow a chosen Bézier curve. This is as simple as adding the control points into the program and feeding it to the `BezierCurve` object. Here in Figure 2.6 are the first results:

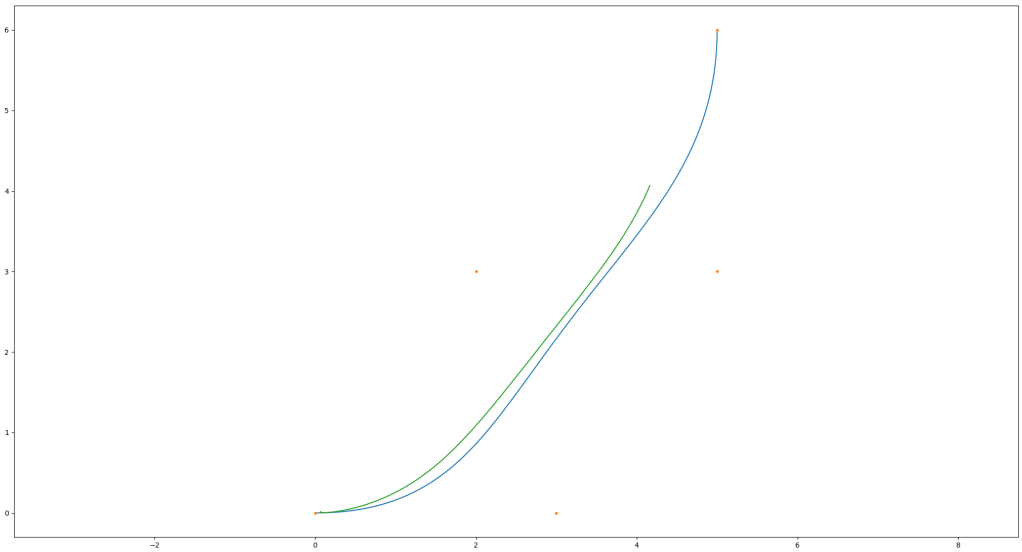
[1] Suggested values for z_{max} and u_{max} are respectively 0.26m/s and 1.82rad/s [1].



(a) RVIZ2 view of the real robot in real time



(b) Path of the robot in the hallway



(c) Comparison with the goal trajectory (blue=goal, green=reality)

Figure 2.6 – First results of the controller

The results are rather encouraging: the robot clearly followed the path rather accurately although it sometimes seemed that when the rotational and linear speeds were both too strong at the same time, it was impossible for the robot to accomplish both. This comes from the fact that on the low-level side of the system, the robot has to find the right speed for each wheel. But this can be difficult if not impossible to achieve if the linear speed z (which more or less defines the mean of the wheels' angular speeds, see Equation 2.1) is so high that the difference of speed defined this time by the angular speed u_2 makes one of the two speeds w_l or w_r out of reach for the servomotors.

In any case, this issue can at least partially be solved by simply tweaking the parameters of the controllers, namely k_1, k_2 (or p_1, p_2 if we consider the poles). In fact, with $k_1 = 9$ and $k_2 = 6$, we get with the same goal trajectory as previously the result in Figure 2.7:

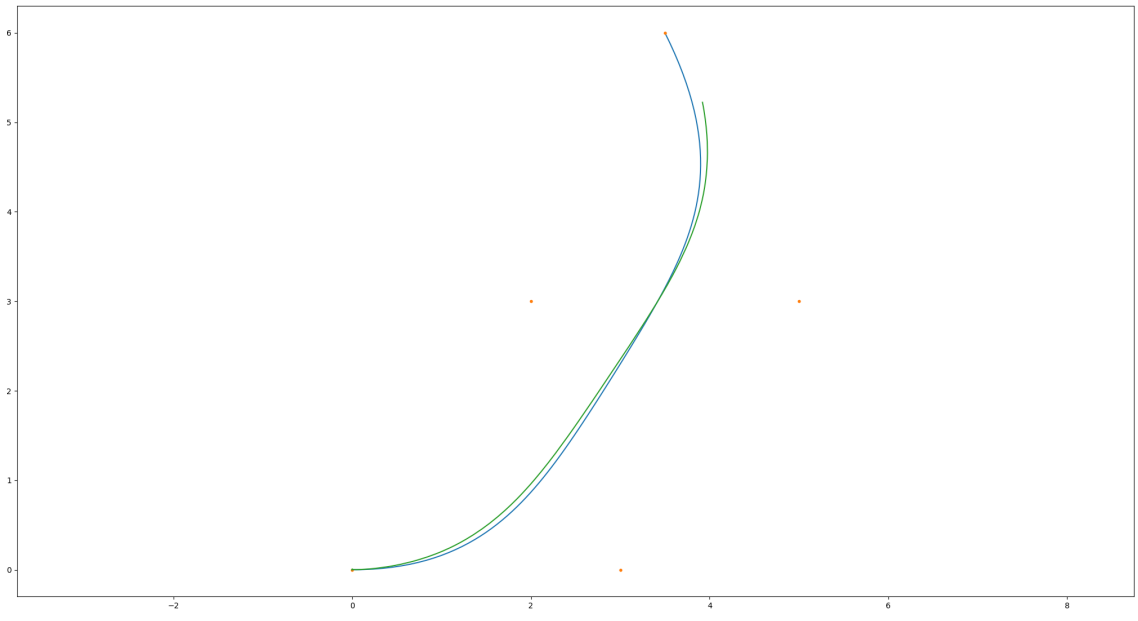


Figure 2.7 – Same Bézier curve with k_1=9 and k_2=6 (blue=goal, green=reality)

These values for the parameters already give use significantly better results. Nonetheless, one assumption remains that may explain the difficulty the robot has to move quickly enough to follow the trajectory. This is the fact that the speed z is not measured or deduced in any way and is simply considered to be equal to the command sent. To tackle this issue, I will now introduce my work on an observer.

Chapter 3

State Estimation

The issue previously mentioned about the lack of knowledge of z is one of the good reasons to start working on an observer. Although the provided *TurtleBot3* packages are open-source, it is rather difficult to determine what the drivers do specifically, but it seems that they merely use the sensor data to determine the position of the robot (relative to its starting point, and even maybe relative to the map frame in the SLAM node, using previous knowledge of the map to recognize landmarks). In that context it could be interesting to add a new observer that will also take into account the cinematic model developed earlier in Section 2.1. This will add a temporal dimension to our state estimation and therefore improve it. Note that in the actual implemented observer, we may decide to ignore the results of the position and rather focus on the speed z, since this is the main unknown variable that we are the most interested in, and the other variables already go through some kind of filtering.

When it comes to the kind of observer used, I immmediately thought of a simple *Extended Kalman Filter* to do the job. This should allow for a good enough estimation despite the fact that the system is not linear (in that case, a normal *Kalman filter* would have been optimal[3]).

3.1 Extended Kalman Filter (EKF)

To start designing the Extended Kalman Filter [4], we need to write our system like so:

$$\begin{cases} X_k = f(X_{k-1}, uk) + \alpha_k \\ Y_k = h(X_k) + \beta_k \end{cases} \tag{3.1}$$

where k is the index corresponding to the instant t_k (time is discretized), $X_k = \begin{pmatrix} x \\ y \\ \theta \\ z \end{pmatrix}$ is the corresponding state, and Y_k is the measurement vector, given by drivers as $Y_k = \begin{bmatrix} \hat{x}_k \\ \hat{y}_k \\ \hat{\theta}_k \end{bmatrix}$. α_k and β_k represent respectively the dynamic uncertainty and measurement uncertainty. Let's also call the uncertainty matrix Γ_k which represents the variance values of uncertainties for the state X_k. We initialize it according to what we think the uncertainty is in the beginning. For X_k, we need to determine a recursive relationship. To do this, we simply use our cinematic model (which gives us $\dot{X}_k$) along with Euler's approximation to obtain:

$$X_k = f(X_{k-1}, u_k) \approx X_{k-1} + \delta_t \dot{X}_{k-1} \tag{3.2}$$

where $\delta_t = t_k - t_{k-1}$ is the constant time gap between two iterations of the filter. This gives us the function f

$$f(X, u) = \begin{bmatrix} x + \delta_t z \cos(\theta) \\ y + \delta_t z \sin(\theta) \\ \theta + \delta_t u_2 \\ z + \delta_t c_1 \end{bmatrix}$$

and its derivative $F(X)$:

$$F(X) = \frac{\partial f}{\partial X} = \begin{bmatrix} 1 & 0 & -\delta_t z \sin(\theta) & \delta_t \cos(\theta) \\ 0 & 1 & \delta_t z \cos(\theta) & \delta_t \sin(\theta) \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.3}$$

As for the measurements Y_k, we immediately obtain the function $h(X)$

$$h(X) = \underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}}_{H(X)=H} \cdot X \tag{3.4}$$

along with its derivative $H(X) = \frac{dh}{dX}$. We now have everything needed to use the Extended Kalman Filter by following its set of equations to iteratively compute our estimation of the robot's state:

$$
\begin{aligned}
A_k &= \frac{\partial f(\hat{X}_{k|k}, u_k)}{\partial X} = F(\hat{X}_{k|k}) \\
C_k &= \frac{\mathrm{d}h(\hat{X}_{k|k-1})}{\mathrm{d}X} = H \\
S_k &= C_k \Gamma_{k|k-1} C_k^T + \Gamma_{\beta_k} \\
K_k &= \Gamma_{k|k-1} C_k^T S_k^{-1} \\
\\
\hat{Y}_k &= Y_k - h(\hat{X}_{k|k-1}) \\
\hat{X}_{k|k} &= \hat{X}_{k|k-1} + K_k \cdot \hat{Y}_k && \text{(Correction)} \\
\Gamma_{k|k} &= (I - K_k C_k) \Gamma_{k|k-1} \\
\\
\hat{X}_{k+1|k} &= f(\hat{X}_{k|k}, u_k) \\
\Gamma_{k+1|k} &= A_k \cdot \Gamma_{k|k} \cdot {A_k}^T + \Gamma_{\alpha_k} && \text{(Evolution)}
\end{aligned}
$$

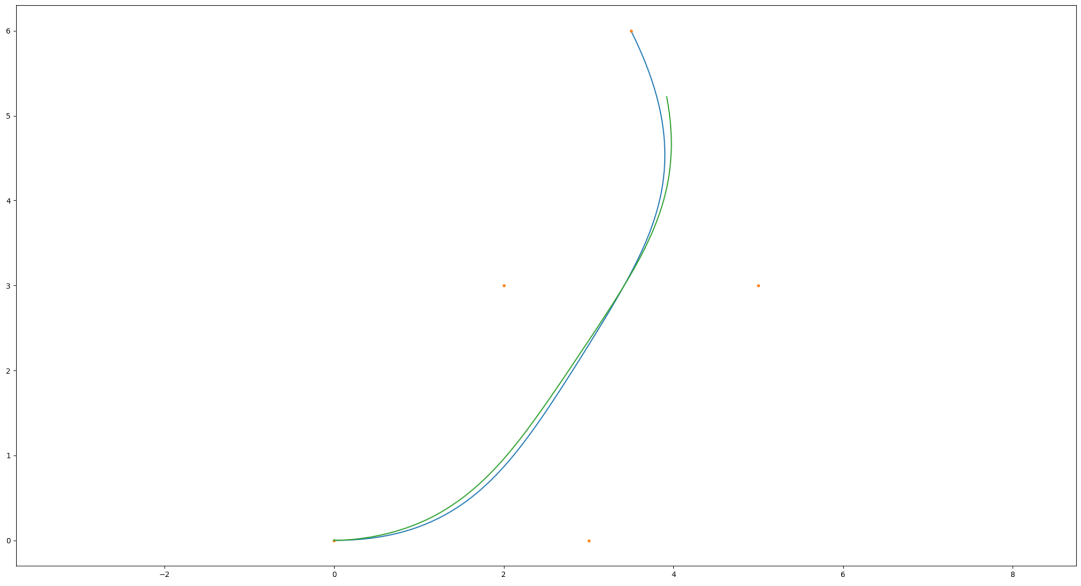
Figure 3.1 – Extended Kalman Filter equations

3.2 Implementation

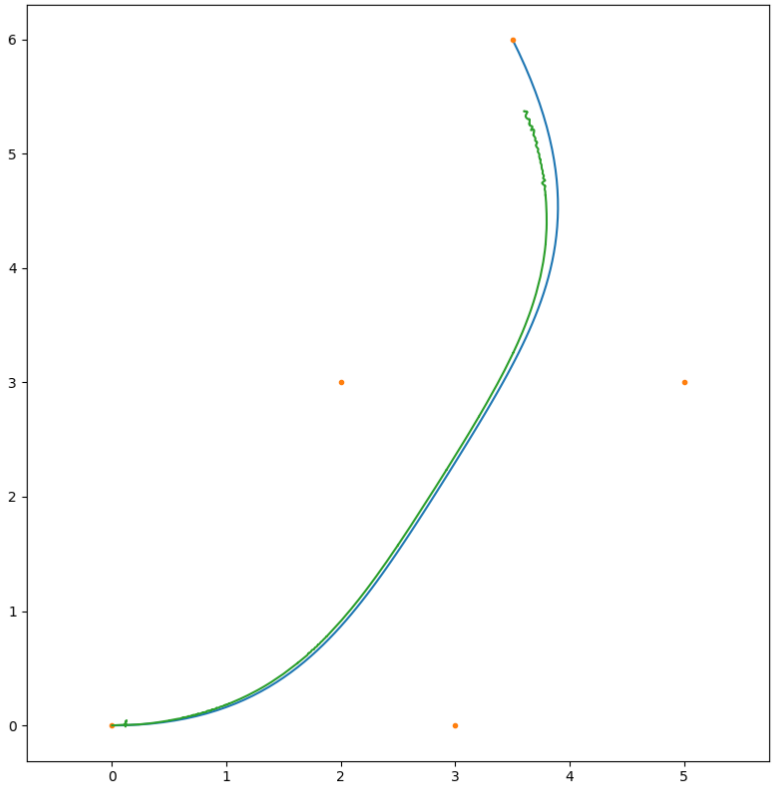
The implementation is particularly easy in *Python* once the filter has been designed. The `observer()` function previously seen in the controller callback (see Figure 2.5) can be modified to call an `ExtendedKalmanFilter()` function that computes the new state X_k and uncertainty Γ_k at each iteration, using the equations in Figure 3.1.

3.3 Results

To test this observer, I started testing the robot with the same trajectory as before to compare the two methods, with and without the filter. Here are the results:



(a) Without EKF (blue=goal, green=reality)



(b) With EKF (blue=goal, green=reality)

The filter seems to have improved the way the robot follows the trajectory. This can be interpreted as the sign that the robot (or its controller/observer node) now has a better understanding of the state of the robot and particularly of its linear speed z, which is what allows the controller to be accurate.

Zooming on the result with the EKF, one can notice some oscillations occuring especially near the end of the trajectory. This leaves the question of the cause of this phenomenon, which may be caused by some instability left behind by the filter in itself, though this does not seem to happen every time. To try to analyze and understand the reasons for this issue, I decided to do some more tests, this time, with some other Bézier curve function with tighter turns, as seen in Figure 3.3.

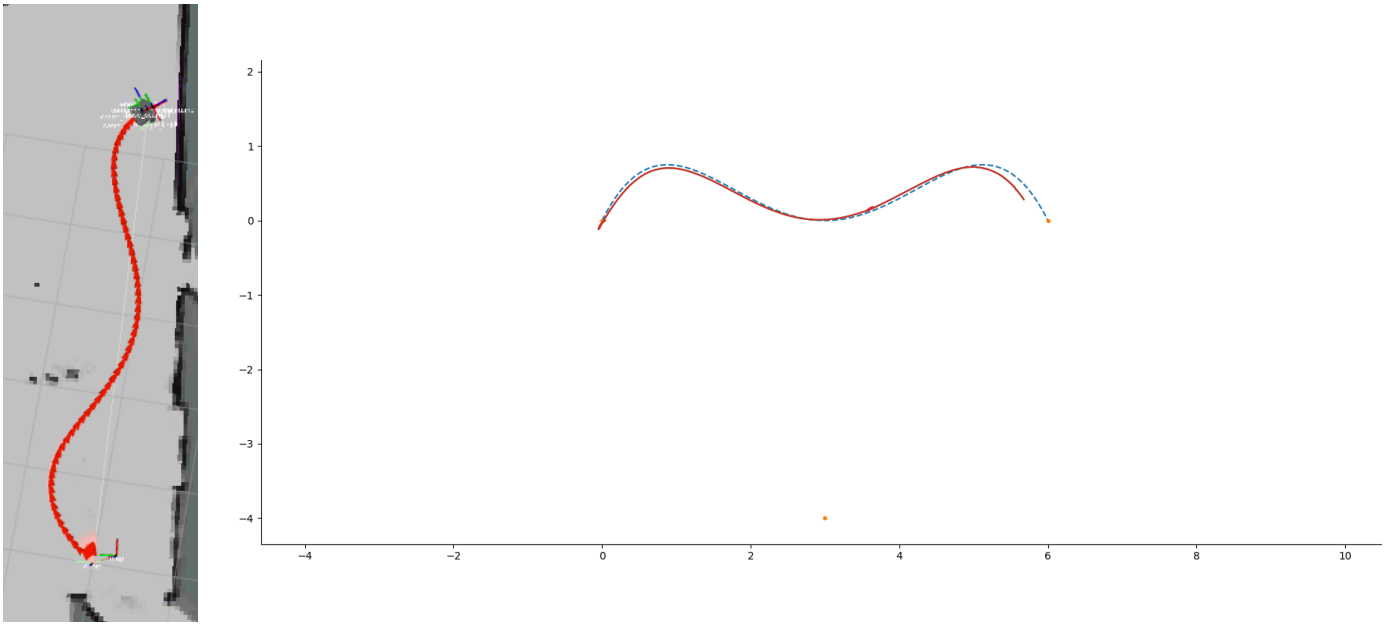


Figure 3.3 – More tests with the EKF
(dotted blue=goal, red=real)

Although the robot seems to be doing a decent job at following the trajectory here, we can notice that it looks like the robot sometimes does not manage to reach the line as if either:

- The proportional error is not computed correctly
- The derivative error somehow accounts for too much in the control (the robot seems to stay tangent to the curve with its heading)

Chapter 4

Path Planning

Now that we are able to control the robot accurately enough with the methods previously described, the next logical step is to work on an higher level, by making it easier for the user to give the robot specific missions, which should be the fun part! Indeed, to hopefully make the robot able to explore and search its environment on its own (or really, any other such mission), we need to work on a program that sends the right instructions to the controller/observer node.

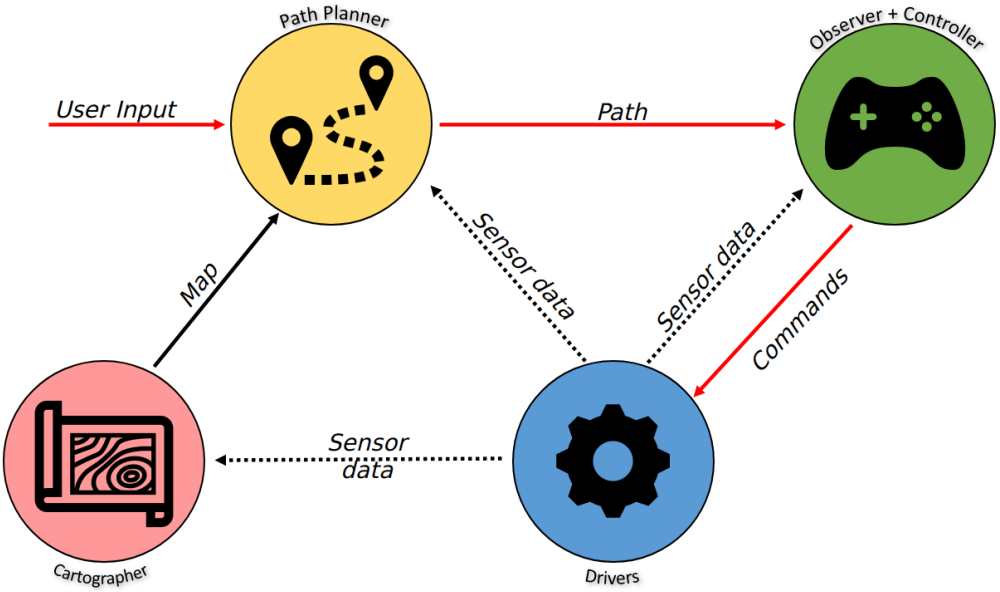


Figure 4.1 – Proposed ROS2 architecture system

This is how I ended up deciding on making a second node that serves as an interface to the user and as a mission planner to send the right instructions (ie, paths) to the controller node, see Figure 4.1.

4.1 User Interface

My goal working on this interface was to develop a workframe both for me and for other students who might work on the same robot later, so that the user could easily pick up from where I stopped and add new features. The program should therefore be versatile enough to be modified and improved, hence a clear architecture that allows for the user to easily change settings, debug, etc. I decided to start developing a new ROS2 node that, apart from the obvious publishers and subscribers it would need to communicate with the rest of the system, would also incorporate a *Finite State Machine* (FSM). This state machine would then gather the user's input and based on that apply specific behaviours and algorithms to accomplish various tasks and perform specific actions. This would then be paired with a display system, so that the user can see the map updated in real-time with the current position of the robot, its objectives, and other useful information. The general architecture of this program is illustrated in Figure 4.2.

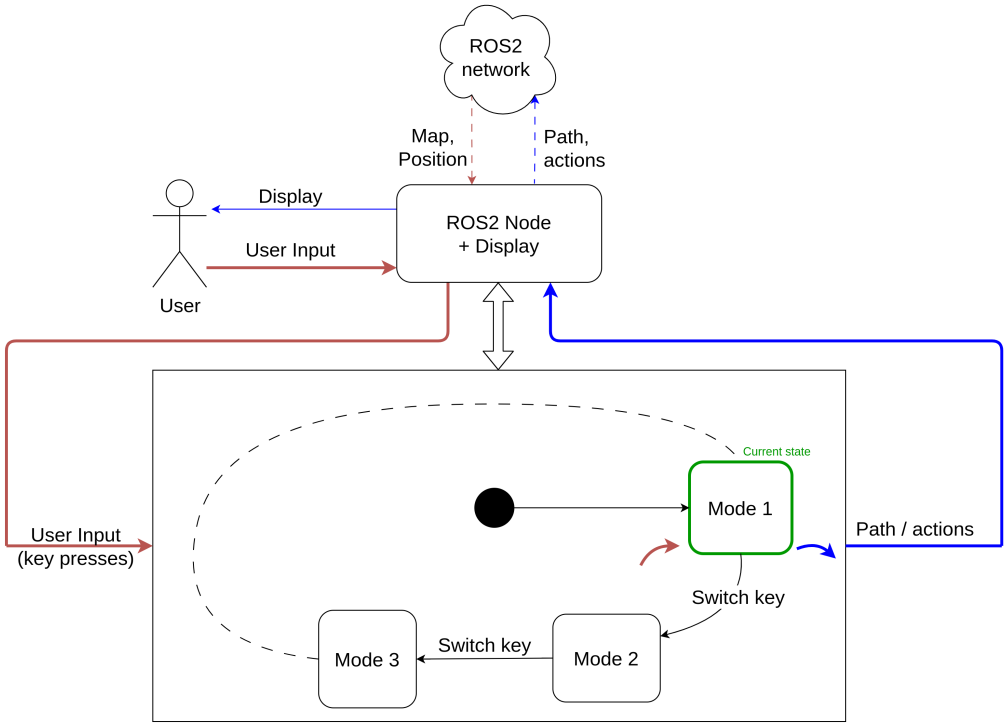


Figure 4.2 – Architecture of the mission planning program

4.1.1 Finite State Machine (FSM)

The system should allow complex missions to be carried out by the robot through a system of states. Despite the numerous actions or computations some specific missions can involve, we can generally speaking expect a mission to revolve around three main steps, as seen in Figure 4.3.

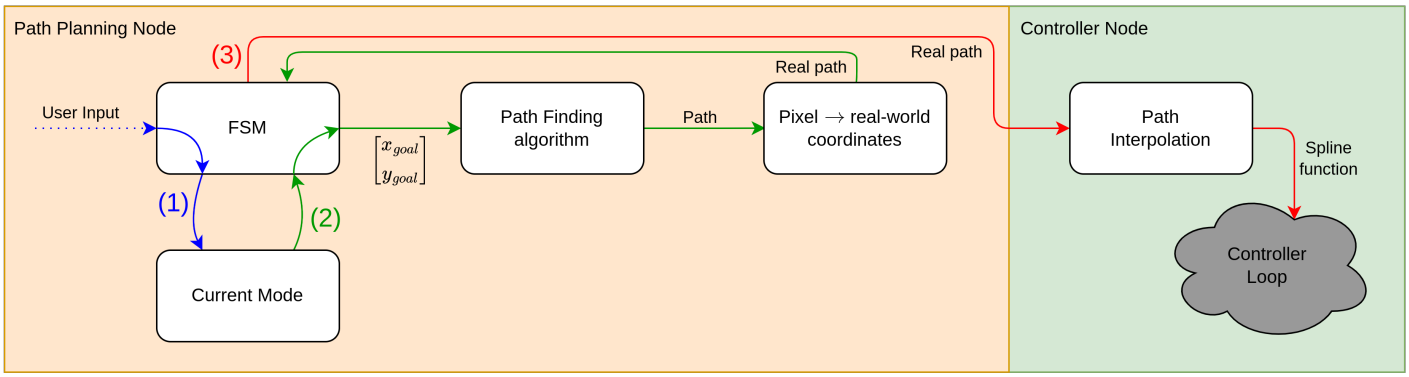


Figure 4.3 – General workflow for path planning

- (1) : The user interacts with the interface (by clicking on it or pressing some key), which triggers the mode that is currently selected in the state machine. Eventually the state machine outputs a goal destination $\begin{bmatrix} x & y \end{bmatrix}^T$ for the robot to go to; for example, if the user clicks on the pixel of coordinate (x, y) so that the robot tries to navigate its way to the corresponding real-life location. Note that in this case, the goal is already in "map" coordinates, but in other modes, we might deal with real-world coordinates, meaning the goal position should be converted into map coordinates before the next step.
- (2) : The $\begin{bmatrix} x & y \end{bmatrix}^T$ is then sent to the path finding algorithm (which may be switched via the state machine as well if we have several) which takes into account the map provided by the cartographer node and the two positions *Start* (typically, the current position of the robot) and *End* (the goal destination). The algorithms returns a path (ie, an array of *2D* points) which avoids known obstacles and that is converted to the real-world coordinates according to the map's information. The path is at this point saved in the state machine and ready to be displayed (after converting it back to "map" coordinates[1]).
- (3) : Eventually (right after computing or after the user confirms the path for example), the state machine sends the saved path(s) to the controller/observer node. The received set of points is

[1] Although the two conversions can sound redundant, they are necessary because the map's size and origin can change over time (as the robot explores more and more); saving the paths in real-world coordinates overcomes this problem as the coordinates can remain constant no matter what happens to the map.

interpreted and interpolated to obtain a continuous and differentiable function that the controller will be able to use to compute the input $\begin{bmatrix} X_d & \dot{X}_d & \ddot{X}_d \end{bmatrix}^T$, as explained later in Section 4.3.

This of course only illustrates the general behaviour and some complex or specific tasks may for instance require signals sent from the controller back to the mission planner (example: to warn that the trajectory was successfully followed and to ask for the next one).

4.1.2 Map processing and display

To use the information sent by the cartographer node, we need to interpret the data sent and transform it into a usable and displayable image. Throughout the project I used *OpenCV*[5] and its Python API to process images and this is also what I used for the user interface, along with the *Numpy*[6] module. The conversion from the *OccupancyGrid* ROS2 message format to a *Numpy/OpenCV* array is rather easy to do. We obtain a matrix of values in the interval $[0, 100]$ (or -1 for unexplored places). Each value corresponds to the probability (in percentage) of the presence of an obstacle at the corresponding location. To apply path finding algorithms on the map, we need a clearer matrix, so that we each pixel is either assigned a 0 if it is considered "empty" or 1 if it contains an obstacle (for display purposes and some advanced features, we might want to take into account the -1 values that represent the unexplored pixels; note that in any case this is done in parallel and does not directly intervene in this step of processing the map for path finding, but we can easily display these areas in a different color). Obtaining that matrix is simply done by binarizing the matrix with a certain threshold (ex: the value 50).

However, the current matrix is still not suitable for our application. Indeed, although it is practical to represent the robot as a point on the map, the program still needs to somehow take into account its width and length so that it does not get too close to a wall and get stuck. Fortunately, there is a simple solution to this issue. Now that the map image is binary, we can apply a *morphological dilation*[7], essentially making all of the obstacles bigger, so that we end up with a "safety" margin : the program will consider these augmented obstacles as the reference for path finding, ensuring that the robot will never hit a wall by driving too close to it.

Applying these two steps is enough to obtain a map usable by the path finding algorithm, but we are obviously free to copy the image into a color one for display purposes.

With the robot launched and ready to go, we obtain something like Figure 4.4, where one can see the wall in black, the margin in orange, the robot in green and the explored areas in white.

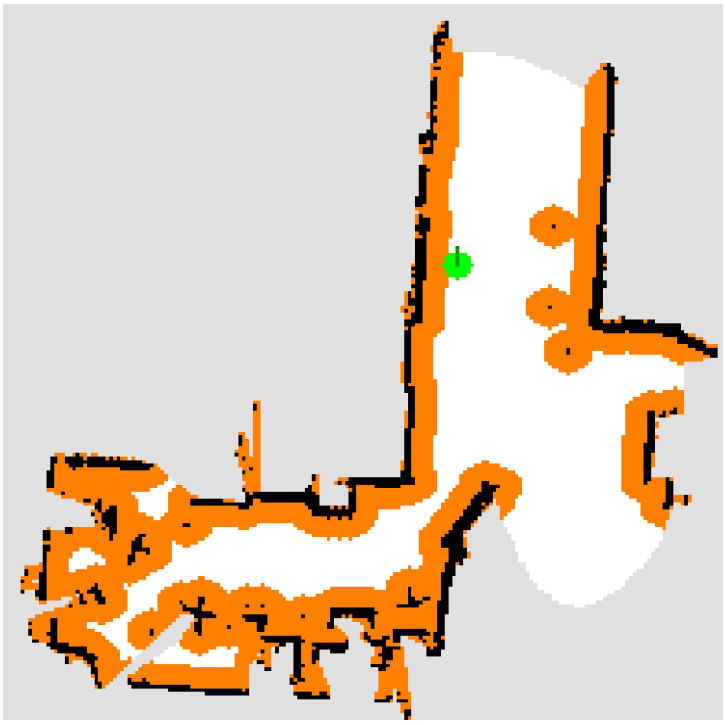


Figure 4.4 – Interface view in real-time

4.2 Path Finding

With an appropriate map, we can now start computing paths that avoid obstacles. In general, we are interested in short paths (although we are fine with not getting the absolute most optimized path). Path finding is a very common problem and there are a lot of different methods and algorithms to tackle it. Here are some of the methods I have come across throughout my research:

- *Dijkstra's algorithm*[8], one of the most famous path searching algorithms,
- The *A* algorithm*, a variant of Dijkstra's algorithm,
- Various methods involving potential fields,
- *ACS (Ant Colony System)* algorithms inspired by ant colonies[9]

For this project I chose two different methods among these. The first one is the *A* algorithm*. This choice is justified by the how common this algorithm is in the fields of robotics and video games, where many times the obstacles found create a similar environment as the one the *Turtlebot3 Waffle Pi* will have to deal with: an environment with most of the space being open, and where the short path only rarely has

to go in a direction generally opposite of the goal (as opposed to complex mazes, where the solution may require the robot to go in completely different ways). This is exactly the kind of situations where we can expect the *A* algorithm* to work well and in particular, faster than *Dijkstra's algorithm*.

The second chosen method is one based on potential fields. This one was first suggested by my internship tutor, as its general principle should make it rather easy to implement. In the case of this algorithm and as opposed to the first one, I started imagining and developping it from scratch.

4.2.1 The A* Algorithm

While *Dijkstra's algorithm* progressively searches for the shortest path in all directions, the *A* algorithm* instead introduces a bias in its search. This bias corresponds to the distance separating each node from to desired goal. This corresponds to a rather intuitive way of thinking about the problem of path finding in the case of a not so complex environment as described earlier: to find a short path to our destination, a good way to start is to look at the nodes that will get us immediately closer to the goal. That way in most cases we end up finding a short path faster than without this bias.

The implementation is not too complicated since it is well documented[10]. The only difficulty is adapting it to the matrix we have.

In addition to the general method of this algorithm, I found some papers describing specific improvements that could be made to the algorithm and that corresponded to the situation the robot is in. The one I decided to try and implement was the *JPS (Jump Point Search)* method[11]. This allows the algorithm to "skip" nodes to essentially jump from node to node, reaching the goal faster. This is done by *neighbour pruning*, that is, determining the "neighbouring" cells for every node according to various rules.

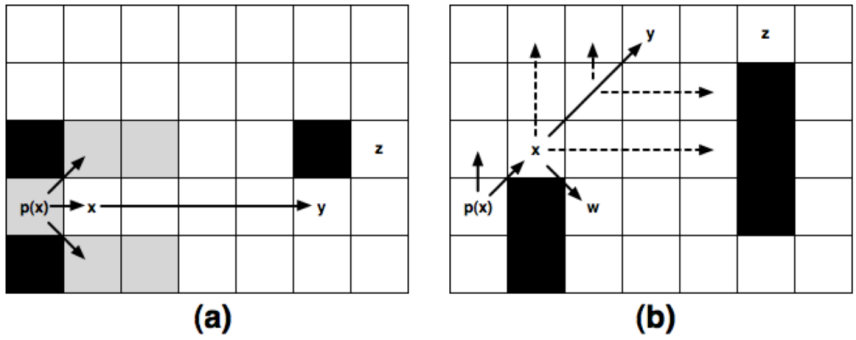
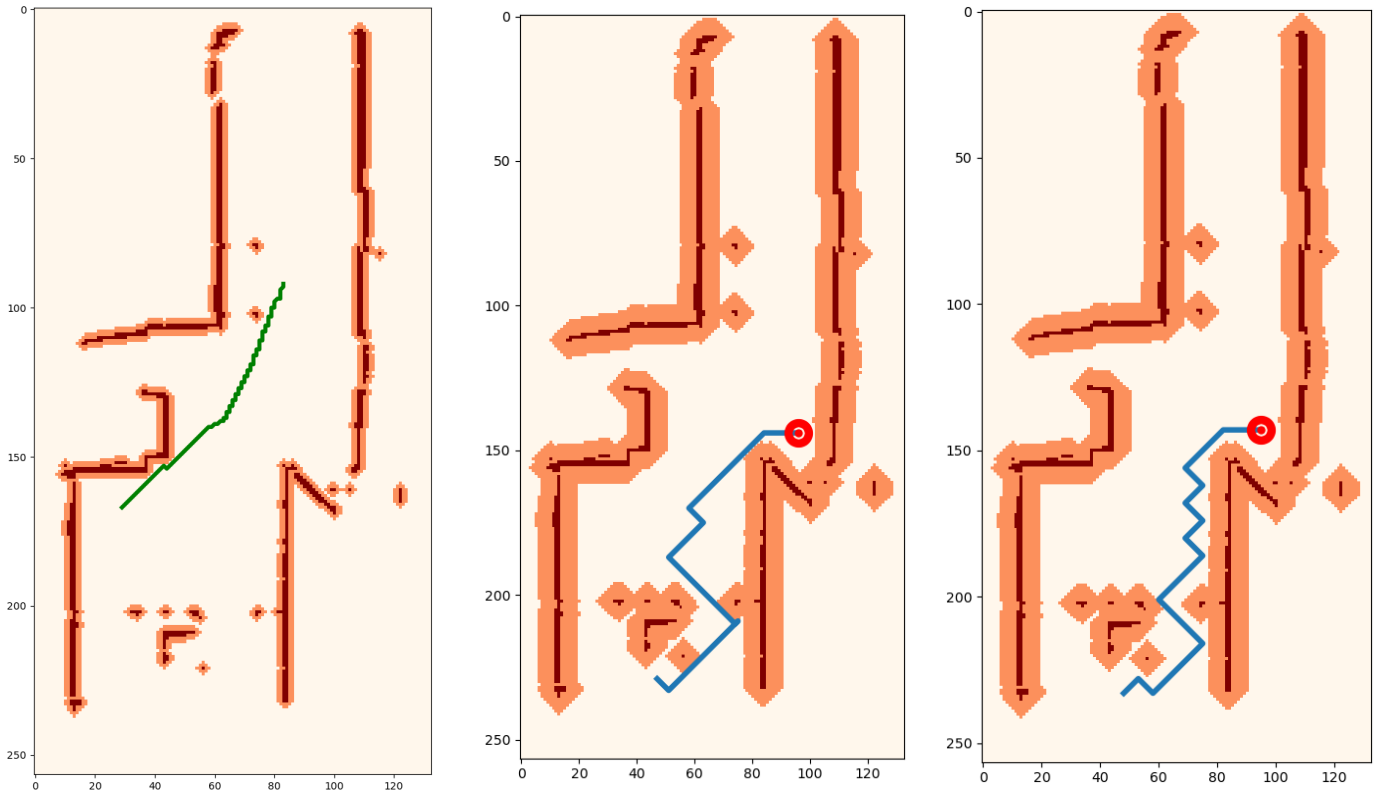


Figure 4.5 – Straight (a) and diagonal (b) jump points of the JPS method (taken from [11])

After writing all the necessary functions, I started testing the algorithm on some examples (saved maps from previous attempts of controlling the robot):



(a) First test without JPS (b) JPS with bias coefficient $k = 1$ (c) JPS with bias coefficient $k < 1$

Figure 4.6 – Tests of the *A* algorithm*

The algorithm clearly seems to be working although there is room for improvement. First of all we notice the JPS method yields more "simple" paths but technically longer (although this could be to the various bugs found throughout development). The first test without the JPS method seems to work rather well, although it was a bit slow to compute sometimes (especially at first and with longer distances). Overall both versions could be used and when porting the algorithm to the real world, the bias coefficient k can be tweaked to find a good compromise.

An important thing to keep in mind is that as long as we are stuck within this 8-connectivity (where the robot can only go in straight lines and diagonals), the resulting "short" path will not necessarily look as good as the one a human would imagine (meaning, "straightforward"). To tackle this issue, one could work on *Any-angle path planning*[12] methods, which the *A* algorithm* can be adapted to, and that should give results more "natural" than the ones shown here.

4.2.2 Path Finding with Potential Fields

Because it might be interesting to see how other methods compare with the *A* algorithm* and also to design a method without simply following pre-established pseudocodes (although I am far from having invented anything here either), I decided to try working on another algorithm, this time based on potential fields.

The idea here is rather simple. The map we obtained that describes the obstacles is to be turned into a potential field (which we can represent as surface in a 3D space) where the closer a point is to an obstacle, the higher the corresponding potential is. From there, a simple algorithm similar to a gradient descent follows the field from the starting point down the slope and (hopefully) towards the destination. Similarly to some physical phenomena (like magnetic fields), we can create a potential field by simply adding up components (attracting, repulsive) independently.

The key to this algorithm is in the process of computing the field. It is the part that takes the most time by far as well. In practice in Python I tried to use vectorized *Numpy* functions as much as possible in an effort to speed up the process.

A good start is thinking how to make the gradient descent finish at the destination. This is simply done by creating a "slope" around the goal position, so that the robot gets attracted. Let us call $G = \begin{bmatrix} x_g & y_g \end{bmatrix}^T$ the coordinate vector of the goal. We can then define a function A_G to calculate the attracting field:

$$A_G(X) = K \cdot ||G - X|| \quad \forall X \in S_{pixels} \tag{4.1}$$

where S_{pixels} is the set of all coordinate vectors corresponding to each pixel (x,y) of the map. This function creates a field where the "altitude" is proportional to the distance to the goal G. The norm $||\cdot||$ can be the euclidean norm but it could also make sense to compute only the Manhattan distance (for computational efficiency). The coefficient K can be tweaked to change the strength of the attracting component.

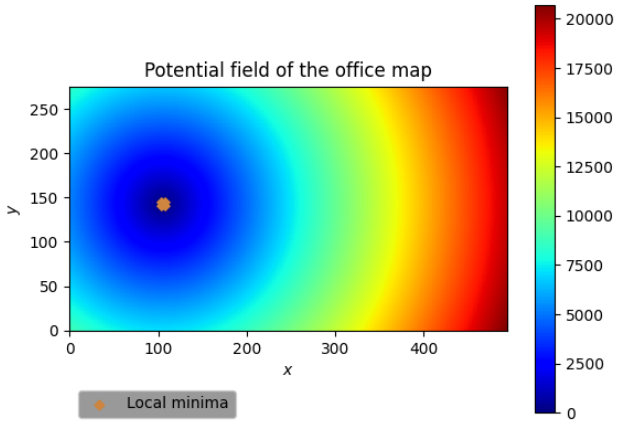
Without surprise this is not sufficient to compute a viable path, since the walls are still not taken into account. We must then define another function generating high values for pixels close to or on obstacles. It must be applied for every pixel W representing an obstacle (*Wall*). This can be achieved with a function R_W of the form:

$$R_W(X) = \frac{P_{max}}{\delta + K \cdot ||W - X||} \quad \forall X \in S_{pixels} \tag{4.2}$$

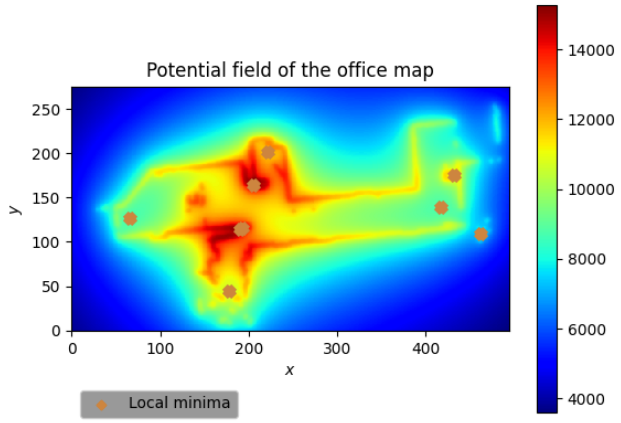
Parameters P_{max}, δ, K can again be tweaked to change the maximum values and the slope of the repulsive component of the field.

Finally, an optional component to add would be one describing the distance to a line (defined by a linear function between two points). This component can be useful to enclose the area around the map in order to prevent the robot from staying stuck on the edges for example. Let us consider two points of the horizontal plane $a = (x_a, y_a)$ and $b = (x_b, y_b)$. We can define the function $E_{a,b}$ with:

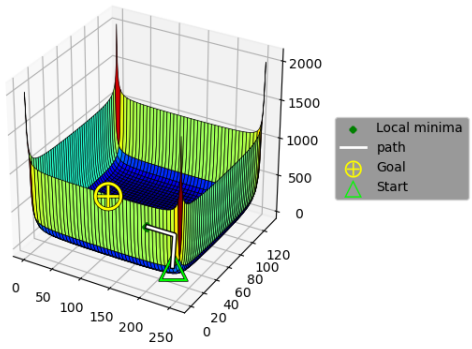
$$\forall X \in S_{pixels}, \quad E_{a,b}(X) = \begin{cases} K \cdot |x - x_a| & \text{if } x_b = x_a \\ K \cdot |y - ((y_a - \alpha \cdot x_a) + \alpha \cdot x)| & \text{else, with } \alpha = \frac{y_b - y_a}{x_b - x_a} \end{cases} \tag{4.3}$$



(a) Attraction component A_G



(b) Repulsive component R_G



(c) Edge repulsive component $E_{a,b}$ applied for 4 edges

Figure 4.7 – Demonstration of the different components of the potential field

The last step of the process is to simply add up all of the components. Let us call P the field function. Assuming we add all four edges of the map as obstacles (as seen in Figure 4.7c), we obtain a potential field of the form

$$\forall X \in S_{pixels}, \quad P(X) = A_G(X) + \sum_{w \in S_W} R_w(X) + \sum_{e \in S_E} E_{e_a,e_b}(X) \tag{4.4}$$

with S_W being the set of all pixels containing obstacles and S_E the set of all edges (pairs of points) taken into account.

All that is left to do is to compute P for the entire image of the map, and apply a simple gradient descent to find the path. We make sure the algorithm does not create a path that crosses itself (that would mean the path is wrong, there's no reason to create loops), and we also limit the maximum length of the path in case something goes wrong and the robot is not able to reach its destination. Simulated tests give the following results:

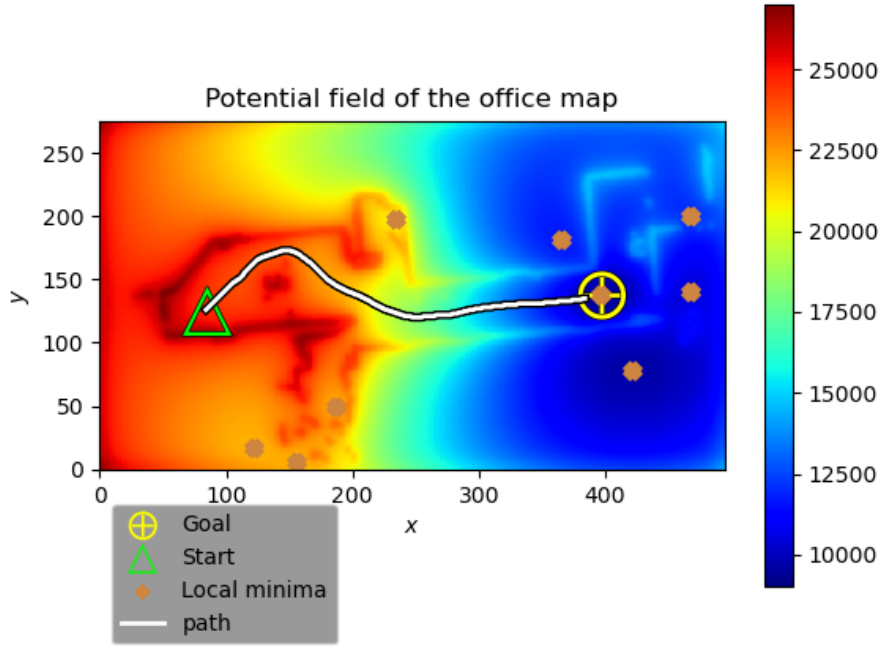


Figure 4.8 – First simulated results of path planning with potential fields

The algorithm in many cases yields great results. Not only does it find a viable path, the latter is also

usually much smoother than the ones found by the previous *A* algorithm* and it usually stays in the middle when there are obstacles on both sides. That being said, there are still some issues. The first one is making sure the path reaches the goal. Indeed sometimes the program gets stuck in local minimum like as in Figure 4.9.

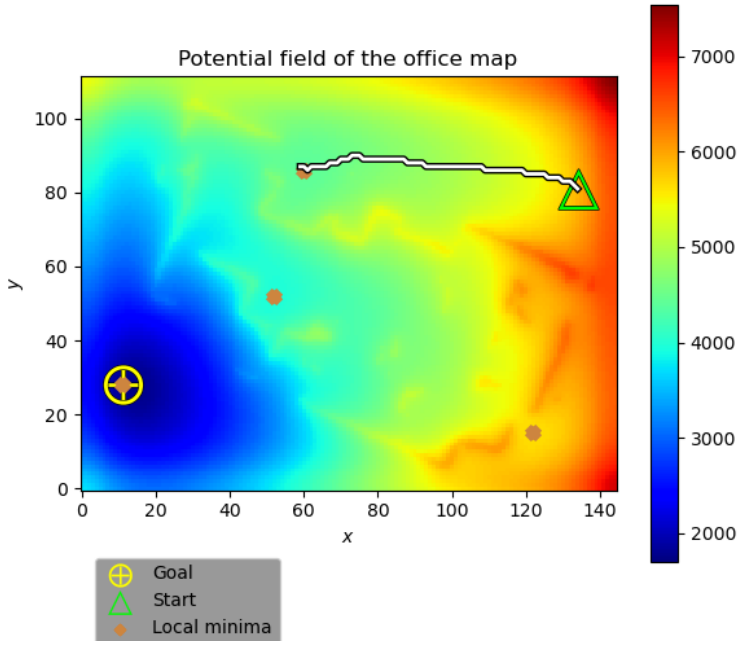


Figure 4.9 – Case where the algorithm gets stuck on a local minimum

A first attempt at fixing this issue was to add some small repulsive components on local minima until there were none (or very few). Although this did work in a few cases, the downside is the added computation even though the path is still not guaranteed to get computed correctly. This brings me to the second issue, being the time needed to compute the result. In the case of a small map computation is rather quick, but this quickly changes as the map gets bigger and bigger. This is due to very high number of computations when adding up the repulsive components for each wall. Indeed, we can expect the number of pixels with obstacles to be generally speaking proportional to the total number of pixels of the image. This gives us a complexity of $\mathcal{O}(n^2)$ where n is the number of pixels of the image (since for each obstacle we compute the resulting repulsive field for each and every pixel).

4.3 Path Interpolation

4.3.1 Cubic Bézier Splines

Now that the program is able to find short paths between two points, we need to work on processing this path for it to be usable by the controller. Indeed, the controller needs to be able to compute $X_d, \dot{X}_d, \ddot{X}_d$ at any time t when the robot is following the trajectory. There needs to be some kind of conversion to go from a discrete set of points to a continuous and differentiable function. In other words, the program must be able to interpolate discrete paths.

To achieve this, various methods could be used, although many of the ones that can come to mind at first seem not to be the right fit. For instance, a simple polynomial interpolation is risky: since the number of points constituting the path can vary a lot (we can definitely obtain paths of length $n > 100$), we would end up with a polynomial of degree n, which can be particularly costly for the program to compute. Instead, I decided to look into splines and specifically *Cubic Bézier Splines*[13], which are a perfect follow-up to the Bézier curves I had already implemented back in Section 2.2 when testing the controller.

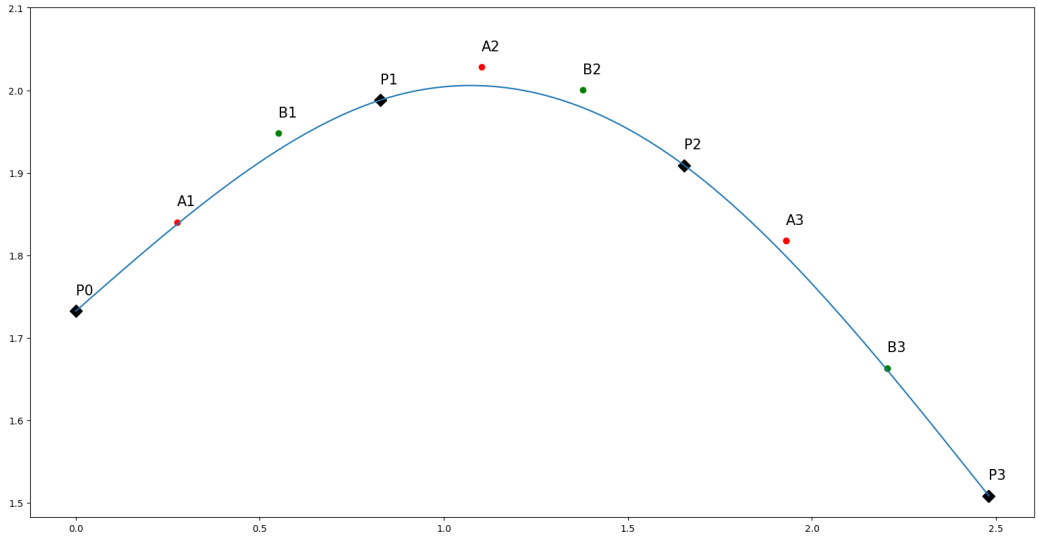


Figure 4.10 – Example of a cubic Bézier spline

Let us consider $P_0, P_1, ..., P_n$, the $n+1$ points constituting the path. A cubic Bézier spline is a curve obtained by the concatenation of n cubic Bézier curves called segments between each pair $(P_{i-1}, P_i), \forall i \in [\![0, n]\!]$.

Each segment i thus consists of two control points A_i and B_i. While $P_0, ..., P_n$ is already given by our path, the control points (A_i) and (B_i) are to be computed in order to obtain a path whose first and second derivative are continuous. Writing these constraints gives us equations we can use to recursively calculate the control points (see [13]).

In practice, all there is to do is to create a new Spline object that can compute the control points as explained, and then create the corresponding Bézier curves using the previously implemented class. Computing the result $S(t)$ at any time t is then just a matter of dividing the desired time interval (typically $[0, 1]$) into n segments, and converting t into a t_{local} corresponding to the time scale of the appropriate segment. Figure 4.11 shows the program has no problem successfully interpolating rather complex trajectories.

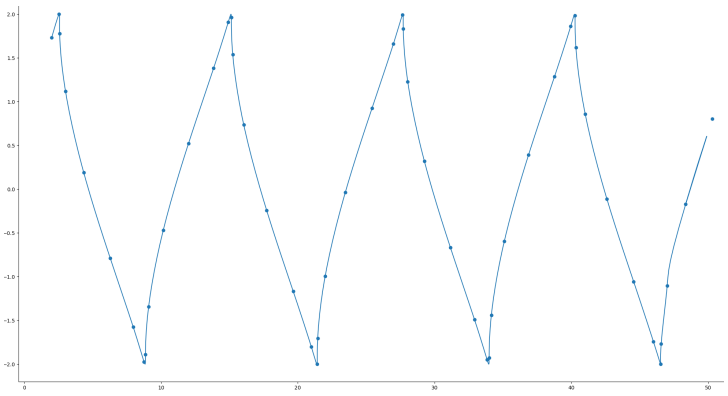


Figure 4.11 – Random complex trajectory interpolated with success

4.3.2 Results

We can now test the interpolation by implementing it inside the controller node to give the robot a specific trajectory to follow. We can also use this opportunity to test the previously implemented path finding algorithms by selecting a goal position on the map.

The program, after (a lot of) debugging, yields very good results. The robot is able to compute a viable path and reach its destination without too much trouble. This path searching/interpolation component is extremely useful to the rest of the project consisting of coding specific behaviours to the user interface.

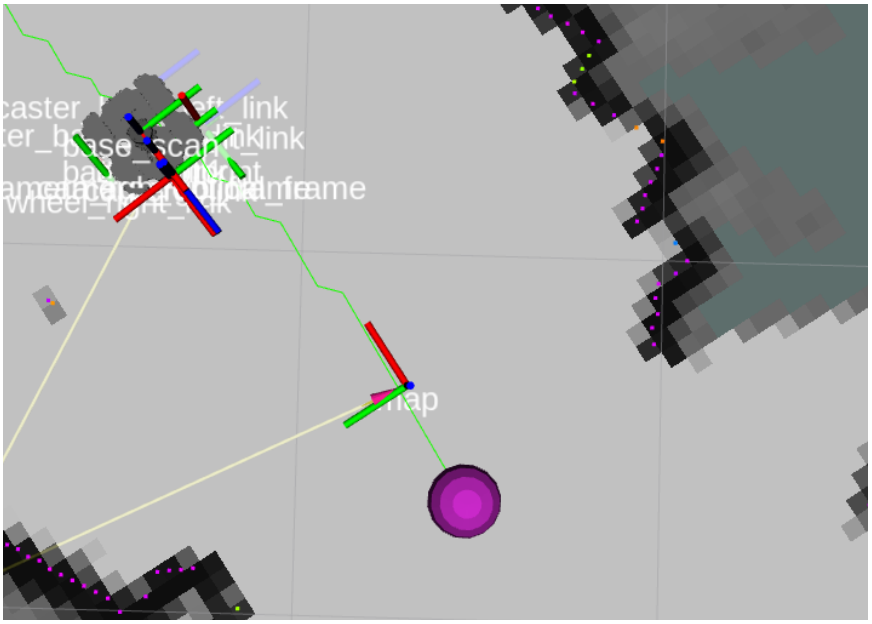


Figure 4.12 – Real-time view (RVIZ2) of the robot following a path

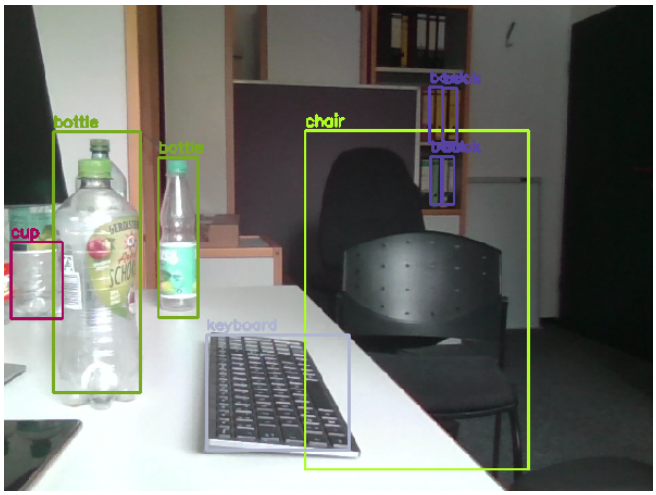
4.4 Object Recognition

Now that the robot can navigate through its environment, an interesting thing to do would be to utilize the camera module, which had not been used yet. With the idea of adding features to my user interface still in mind, my goal was to make a program able to use the video feed to detect objects such as a bottles lying around the office (of which there were quite a few, waiting to be returned in exchange for their deposite). The program should then be able to use the bottle's position in the image and the data sent by the LIDAR to deduce an estimation of the location. This location should then be displayed on the user interface. This would essentially make the robot able to explore the environment and detect lost objects on its own.

Developing such a program starts in practice by coding a ROS2 node running on the *Turtlebot3 Waffle Pi* and that will read the camera feed and send it to the rest of the program running on my computer. This comes with a first challenge, which is managing to send all this data without too much latency, since sending images (of good enough quality) takes times over Wifi, especially from a *Raspberry Pi*. This issue is at least partially solved by compressing the image before sending it. Despite having to decompress it again when receiving it on my computer, this method greatly improves the latency.

4.4.1 YOLO CNN Model

Although this could have been an interesting (yet time consuming) challenge, the image processing was not started from scratch. Instead, I used an already trained CNN Model called *YOLO V3*[14]. This is a model trained to detect various objects from everyday life. I simply used the model inside of the ROS2 node running on my computer to receive the video feed, and analyze each frame with the model in order to detect bottles. When a bottle is detected, its general position in the image is fairly easy to obtain thanks to the bounding boxes the network yields.



(a) Examples of the various objects that can be detected

(b) Bottle detection in a typical scenario

(c) Real-time interface and camera feed

Figure 4.13 – Demonstrations of the detection of objects

Right out of the box the model yields great results and manages to detect the bottles rather easily. Although this works, the remaining problem is clearly the speed at which the programs processes the image; the latency is still manageable and is enough for the use I want to make of it, but a machine with more power

would have certainly made things even better.

4.4.2 Localizing an object

Now that the program can detect where the bottle is within the image, my goal is to be able to compute an estimation of its position in the real-world coordinates (ie, according to the world frame established by the robot). To achieve this, I want to use the position in the image, which at the end of the day is an angular value, to deduce which angle to look at withing the measurements taken by the LIDAR. The problem we have with the *Turtlebot3 Waffle Pi* is that the angle from the camera is not the same as the angle from the LIDAR, since the two sensors are not aligned:

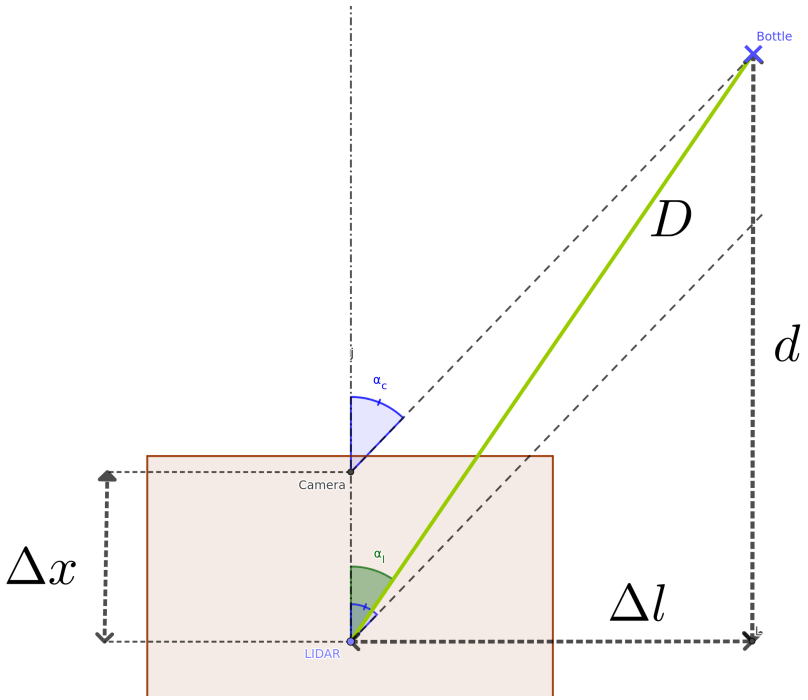


Figure 4.14 – Geometry of the bottle relative to the robot and its sensors

The angle we have access to is α_c, the angle relative to the camera module. Unfortunately the distance information we have is D, the distance from the LIDAR to the object. This means that in order to deduce d and Δl (respectively the x and y coordinate of the object in the robot's frame), we need α_l, the angle of the bottle as seen from the LIDAR. Yet in general, $\alpha_l \neq \alpha_c$ since $\Delta x \neq 0$ (in particular, we have $|\alpha_c| \geq |\alpha_l|$). However, approximating α_l by $\alpha_l \approx \alpha_c$ is an acceptable approximation as long as $\Delta x \ll d$, which is clearly the case in our situation. Indeed, $\Delta x \approx 10cm$ while we can expect that the object will be

at least $1m$ or more away from the robot.

However, this approximation is not accurate enough to simply take the LIDAR measurement at angle α_c and compute the result like so. Instead, we consider that the right measurement to take into account is found in an interval centered around $[\alpha_c - \delta, \alpha_c + \delta]$.

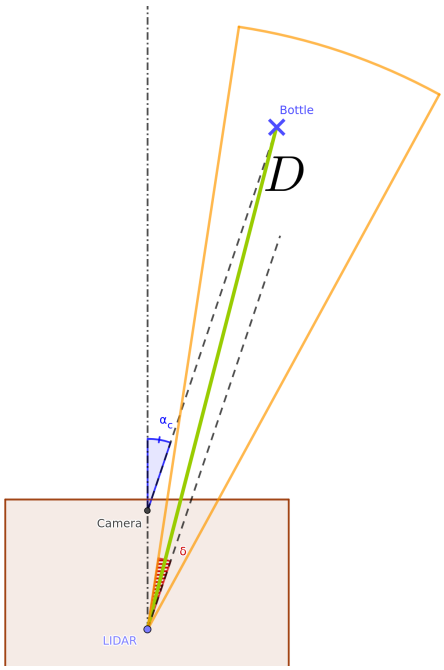


Figure 4.15 – Geometry of the bottle relative to the robot and its sensors

The programs thus finds distance D by finding the minimum value within the angle interval $[\alpha_c - \delta, \alpha_c + \delta]$. That way we also obtain a better estimation of α_l.

Finally, we can then compute the absolute position of the bottle. Let us define:

- $\vec{P} = \begin{bmatrix} x & y \end{bmatrix}^T$ the current position of the robot,
- $\vec{X} = \begin{bmatrix} \cos(\theta) & \sin(\theta) \end{bmatrix}^T$ the unit vector describing the robot's current heading,
- $\vec{Y} = \begin{bmatrix} -\sin(\theta) & \cos(\theta) \end{bmatrix}^T$ the unit vector perpendicular to $\vec{X}$

We obtain for the position $\vec{B}$ of the bottle the formula

$$\vec{B} = \vec{P} + D\cos(\alpha_l) \cdot \vec{X} + D\sin(\alpha_l) \cdot \vec{Y} \tag{4.5}$$

4.4.3 Results

Figure 4.16 – The program successfully detects a bottle and computes its location on the map (purple)

Although the code could be improved to be both faster and more reliable, the goal is reached. The program successfully manages to find where the bottle (or really any object) and the results are rather accurate, as seen in Figure 4.16, where the bottle appears on the map right where it should (the purple point sits on top of a small obstacle as higlighted by the orange pixels around) (the map appears mirrored).

4.5 Missions

To finalize the project and now that all the components needed have been implemented, I can work on coding "modes" to the user interface, that is, behaviours to make the robot accomplish missions.

4.5.1 Single Goal Mode

This first mode is the most simple of them all. It simply consists of the user choosing a destination on the map that the robot needs to go to while avoiding obstacles.

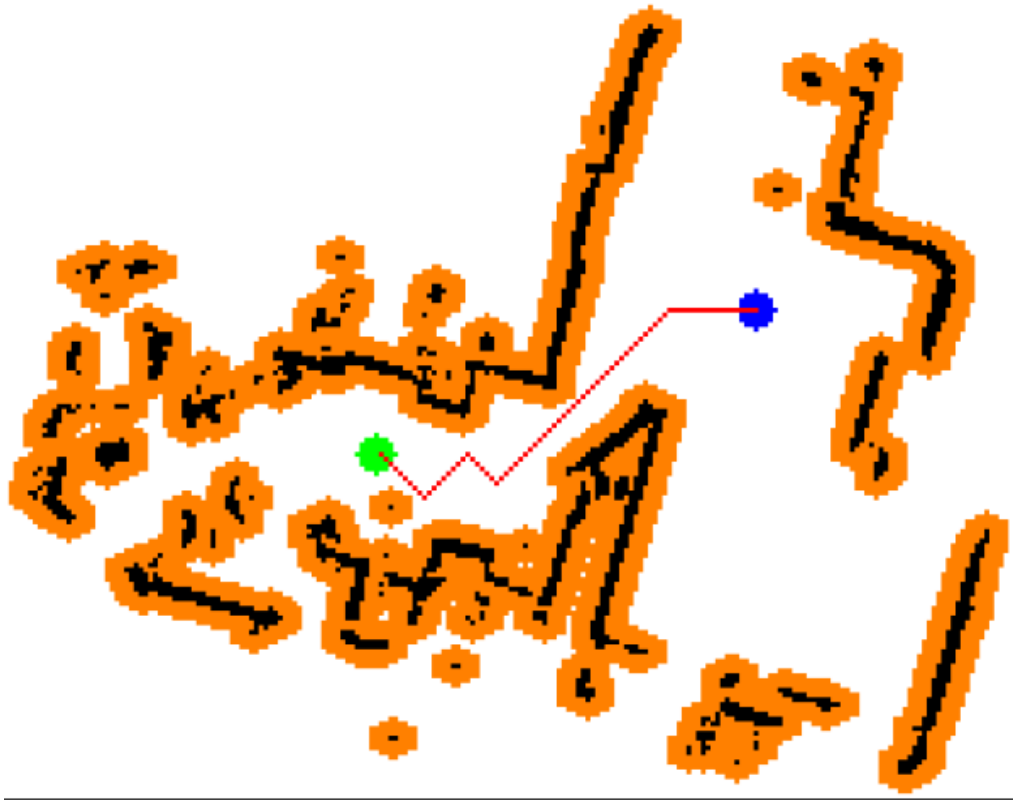


Figure 4.17 – The robot (green) following its path to its destination (blue)

Again, both the path finding and interpolation algorithms seem to work properly and make the robot able to reach its destination. While still working on this simple mode, I thought about adding a new useful feature. We know that for now the robot takes into account the whole map given by the cartographer node, which is powerful as it makes the robot "remember" the environment it already explored. But what if that environment changes ? Let us say that the robot has visited at some point a specific corridor but is now elsewhere. If the user or the current mission suddenly wants the robot to go back to that location (or to follow a path nearby), we want the robot to be ready to avoid any obstacle, even if that obstacle used not to be there. That is currently not possible as the robot only computes a viable path at the very beginning, right before starting to follow it.

This calls for the implementation of a *Rerouting* system, an important feature which will also prove useful for other kinds of missions. I thus implemented functions to detect in real-time (as the robot is moving along its path) if the path currently followed happens to cross an obstacle (or worse, lead directly to an obstacle). If there is an obstacle right where the robot was supposed to go, the algorithm recomputes a new path with the new map, and the robot can continue its way towards it destination.

(a) The robot plans a path based on its knowledge on the environment

(b) An unexpected obstacle appears along its way, the robot computes a new path avoiding it

Figure 4.18 – Demonstration of the rerouting feature

4.5.2 Checkpoint Mode and TSP

Now that the robot can reach a single goal it is very easy to create a mode where the user can select several checkpoints that the robot has to go through. If we simply take the checkpoints in order, and concatenate the paths into one, we obtain the results seen in Figure 4.19.

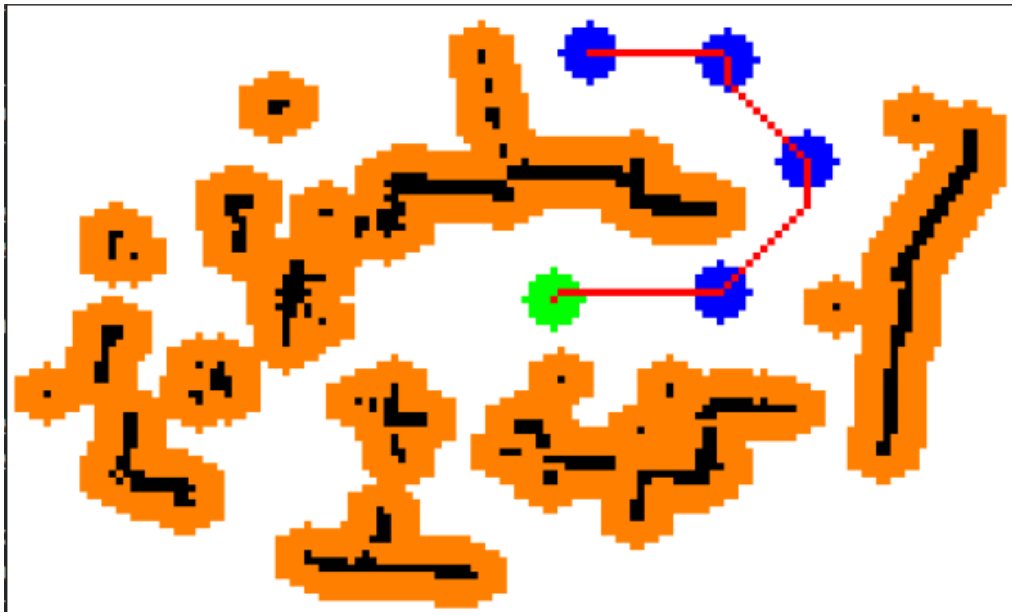
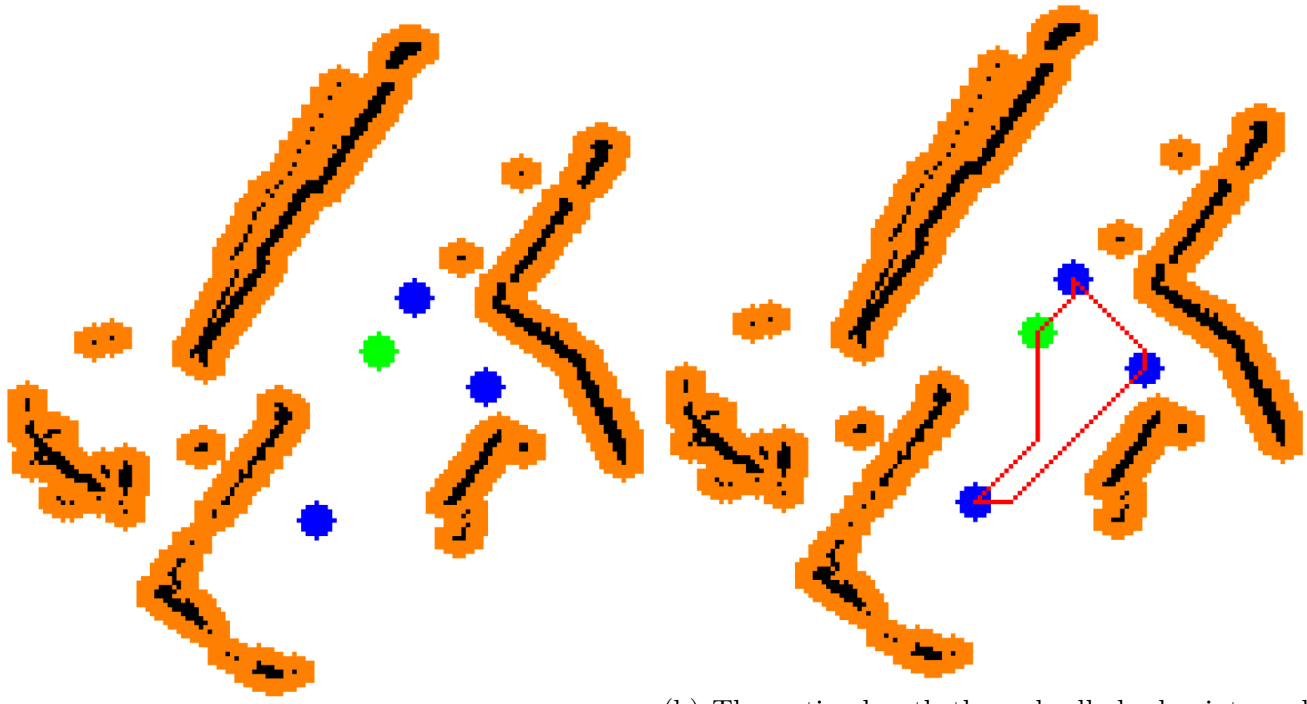


Figure 4.19 – A complex path defined by 4 checkpoints (in blue)

Now, what if the user wanted the robot to go to several locations, as quickly as possible ? This is a feature quite useful in robotics, when we want to pick up objects or perform specific tasks in various places before finally going back to the starting point. This problem is the rather famous *Travelling Salesman Problem* (*TSP*) and happens to have been particularly well researched, although it is *NP-hard*. With some more time, implementing one of the many methods developed for this problem would have been rather easy (especially the *Nearest Neighbor (NN) algorithm*[15], but this is a greedy algorithm). But since tackling this problem was not a key part of my project, I chose to implement the simplest (yet exact!) algorithm there is: brute-force search. It goes without saying that trying all permutations, which gives a complexity of $\mathcal{O}(n!)$ (n being the number of points), is completely unreasonable even for large numbers of checkpoints, and this is even worse considering that in my case calculating the lengths of the edges (ie, the paths between each pair of checkpoints) takes a long time on its own (because for it to make sense the length corresponds to the number of points constituting the path computing via one of the path searching algorithms described earlier, which in itself is already quite slow).

Thus after implementing it I tested it on a few points (more than 5 points was a bit too much, as it took a long time to compute), as illustrated in Figure 4.20:



(a) The user selects 3 checkpoints on the map

(b) The optimal path through all checkpoints and back to the start is computed

Figure 4.20 – Demonstration of the TSP Mode

4.5.3 Exploration Mode

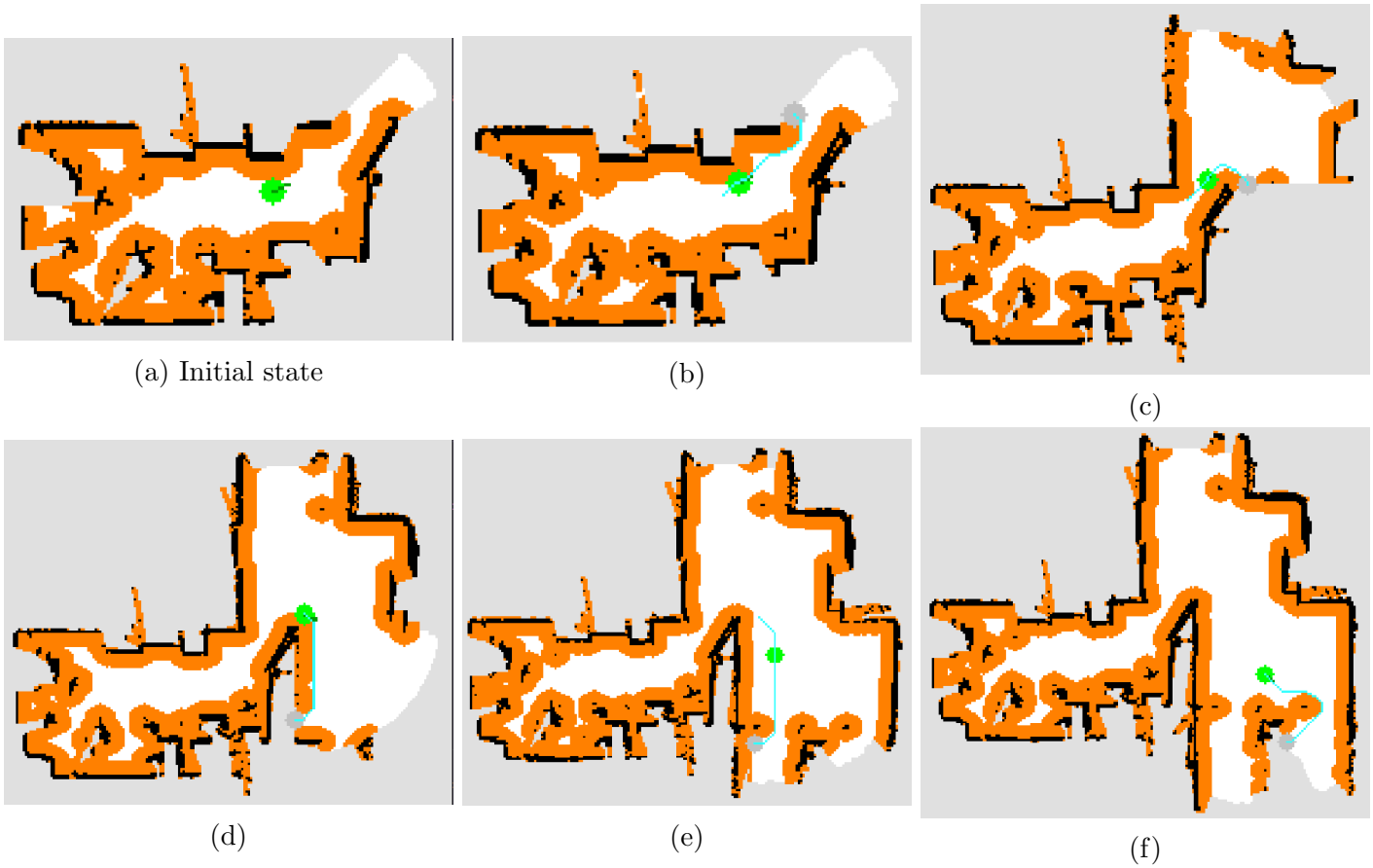
The last mode developed during this internship was the *Exploration Mode*, a mode that is a first step towards making the robot autonomous while also reusing all of the features that were implemented until then. The robot's goal in this mode is simple:

1. Exploring its environment autonomously to map it
2. Scan the environment to find bottles (or any object of interest) and add their location on the map

Exploring the map is done by using the map divided in three different areas:

- Unexplored
- Obstacle
- Explored (no obstacle)

We then simply detect the borders between two regions *Unexplored* and *Explored*, and find the point belonging to one of these borders that is the closest to the robot current's position. The program then computes a new path to the said point. Once the point is reached (or if the point happens to be an obstacle), a new point is recomputed using the same technique. This is how the robot can progressively explore and map its environment without the user's intervention.



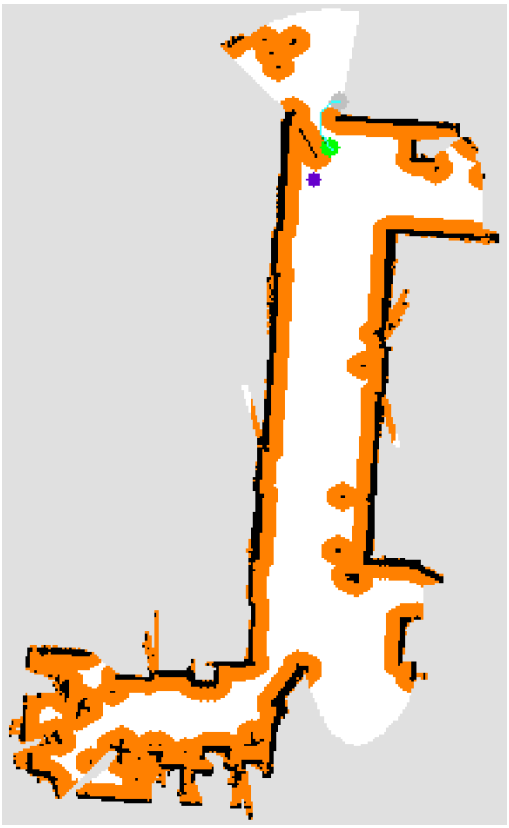
(a) Initial state (b) (c)

(d) (e) (f)

Figure 4.21 – A wild *Turtlebot3 Waffle Pi* in *Exploration Mode*

Finally, by using the methods described in Section 4.4, the robot becomes able to look for bottles as it is driving around (see Figure 4.22), becoming a *Pfand* gatherer!

(a) The robot starts exploring



(b) A bottle has been found along the way (in purple)

Figure 4.22 – Demonstration of the full *Exploration Mode* with the camera

Conclusion

In conclusion, I believe the internship has been a success! Despite the challenge this was, managing to set my own objectives when it comes to how many features I wanted to implement was very interesting; seeing all of them come to life on the robot was particularly rewarding and satisfying, and so was working my way up from the intermediate-level control to the high-level control. The robot is now able to be controlled in several ways, while being monitored by the user through the interface. The last mode implemented, the *Exploration Mode*, to me is the most important one, as it combines all previous features and allows the robot to become somewhat autonomous. With more time my focus would have shifted to making existing features more reliable and working on new modes in an attempt to make the robot accomplish missions more and more ambitious and complex without the user's intervention. Although no revolutionary method was developed during this internship, I did learn interesting concepts such as the implementation of specific path finding and path interpolation algorithms; moreover, I believe the biggest challenge for me was managing to get the whole system working despite the many programs and methods involved. The *Turtlebot3 Waffle Pi* does have the advantage of having very accurate actuators and sensors, which quite obviously proved to be very useful, yet debugging the whole system sometimes was difficult. Maybe more intensive, rigorous and specific tests on every algorithm or feature would have helped me figure out the sources of the issues faster and would have essentially sped up my progress. Finally, one of the goals I had set was to make programs that other people could later use as well without too much trouble. I believe this objective is somewhat met thanks to the documentation I started writing and the general architecture of the programs. I hope for instance that another student will be able to use the *Mission Planner* node (or Path Planning node) easily and add their own new path searching algorithms or missions.

Acknowledgments

I would like to express my gratitude to all of the people helping me throughout this internship, starting with the Department of Computer Science of the University of Oldenburg and Andreas Rauh, who allowed me to have this internship and work on this robot. His help along with Oussama Benzinane's guidance were important assets to this project and I appreciated the freedom I had when it comes to the exact nature of my work with the *Turtlebot3 Waffle Pi* . I thank them for giving me a glimpse at the world of research and for the opportunity to apply many methods I had yet to experiment with in real life, while also learning new ones. I would also like to thank my teachers back in France including Luc Jaulin whose teachings were undoubtedly very useful during this internship. Finally I would like to point out my general satisfaction as regards these 16 weeks spent in Germany, as studying and working abroad is an interesting and rewarding experience that in my case was a clear success overall.

Bibliography

[1] TurtleBot3. https://emanual.robotis.com/docs/en/platform/turtlebot3/overview/.

[2] Luc Jaulin. Mobile robotics: Guidance. *Mobile robotics*. https://www.ensta-bretagne.fr/robmooc/.

[3] Kalman filter. *Wikipedia*, June 2023. https://en.wikipedia.org/w/index.php?title=Kalman_filter&oldid=1160108771.

[4] Luc Jaulin. Mobile robotics: Kalman Filter. *Mobile robotics*. https://www.ensta-bretagne.fr/kalmooc/.

[5] OpenCV. https://opencv.org/.

[6] NumPy. https://numpy.org/.

[7] Morphological Image Processing. https://www.cs.auckland.ac.nz/courses/compsci773s1c/lectures/ImageProcessing-html/topic4.htm.

[8] Dijkstra's algorithm. *Wikipedia*, July 2023. https://en.wikipedia.org/w/index.php?title=Dijkstra%27s_algorithm&oldid=1167009358.

[9] M. Dorigo and L.M. Gambardella. Ant colony system: A cooperative learning approach to the traveling salesman problem. *IEEE Transactions on Evolutionary Computation*, 1(1):53–66, April 1997. http://ieeexplore.ieee.org/document/585892/.

[10] A* search algorithm. *Wikipedia*, August 2023. https://en.wikipedia.org/w/index.php?title=A*_search_algorithm&oldid=1169167028.

[11] Daniel Harabor and Alban Grastien. Online Graph Pruning for Pathfinding On Grid Maps. *Proceedings of the AAAI Conference on Artificial Intelligence*, 25(1):1114–1119, August 2011. https://ojs.aaai.org/index.php/AAAI/article/view/7994.

[12] Any-angle path planning. *Wikipedia*, July 2023. https://en.wikipedia.org/w/index.php?title=Any-angle_path_planning&oldid=1166809000.

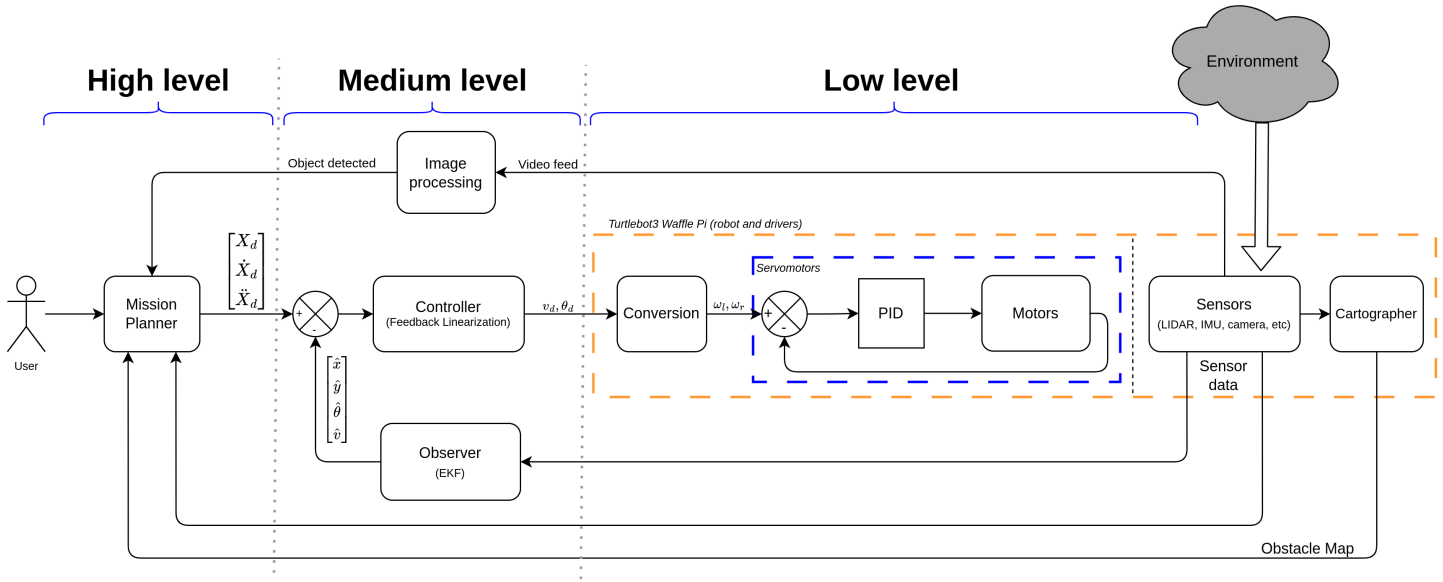
[13] Michael Joost. Cubic Bezier Splines.

[14] YOLO - object detection — OpenCV tutorial 2019 documentation. https://opencv-tutorial.readthedocs.io/en/latest/yolo/yolo.html.

[15] Nearest neighbour algorithm. *Wikipedia*, April 2023. https://en.wikipedia.org/w/index.php?title=Nearest_neighbour_algorithm&oldid=1147996543.

System Block Diagram - Turtlebot3 Waffle Pi

The system can be divided into three levels of control, the low level being already taken care of by the hardware and software provided with the robot. Some data processing including cartography is also included, making easier to obtain a good estimation of the state of the system. The project aimed to develop the higher levels of control, allowing the user or algorithms to dictate the robot its behavior.



Notes

$X_d = \begin{bmatrix} x_d \\ y_d \end{bmatrix}$ corresponds to the desired trajectory to follow

TurtleBot3 Waffle Pi Documentation

Release 1.0

Hugo HOFMANN

Aug 10, 2023

CONTENTS

CHAPTER

ONE

DESCRIPTION

This is the documentation regarding the Python programs developed during the summer of 2023 at the University of Oldenburg, regarding the control, observation and path planning of the TurtleBot3 Waffle Pi. The code is available here[1].

Important: All provided codes are purely experimental and would still require more work, more features and debugging. This projects intends to offer some ground work to experiment with the robot, allowing one to control it.

[1] https://gitlab.ensta-bretagne.fr/hofmanhu/turtlebot-waffle-claw-stage-2a

CHAPTER

TWO

SUMMARY

2.1 Installation

2.1.1 Setup

The first step is to follow ROBOTIS's user guide[2] to build the robot and prepare the software with the provided packages. All the code developed throughout this project was run on **Ubuntu 20.04** and **ROS2 Foxy**. Be sure to follow the right tutorial to install the necessary packages. Follow the steps to setup the Raspberry Pi to connect to your WIFI network and if possible try to setup its IP address static. This will facilitate connecting to the robot via SSH.

Tip: The tutorial provides an image of *Ubuntu 20.04* with all the necessary *ROS2* packages for the Raspberry Pi of the robot, which makes the whole setup easier.

2.1.2 Camera debugging (if necessary)

Important: The currently provided image of Ubuntu 20.04 is a 64-bit version, which may cause problems with the camera drivers. After installing everything, follow the *bringup* steps to test the robot for the first time. Then in a terminal on your computer, run:

```
ros2 topic list
```

Check for any topic named "*camera*" or a similar name. If no such topic is present in the list, try the following steps explained in this Github issue[3]

To use the camera (although most of the code developed in this project can be run independently of it), a specific ROS2 node is provided as part of this project, that should be running on the robot's Raspberry Pi just like all the drivers. To facilitate launching the robot, the repository contains a set of *bash* scripts as explained in the next section that take care of launching all these programs on the robot.

[2] https://emanual.robotis.com/docs/en/platform/turtlebot3/quick-start/#pc-setup
[3] https://github.com/ROBOTIS-GIT/turtlebot3/issues/863#issuecomment-1243283552

2.1.3 Project code

Clone the git repository[4] of the project on your computer with the command:

```
git clone https://gitlab.ensta-bretagne.fr/hofmanhu/turtlebot-waffle-claw-stage-2a
```

To copy the necessary files from your computer to the robot, run:

```
cd turtlebot-waffle-claw-stage-2a
scp basic_launch.sh Scripts/node_camera.py ubuntu@IP_ADDRESS:~
```

Enter the password of the Raspberry (by default, `turtlebot`) to copy the files.

Now that this is done you should be ready to use the robot and the programs. *Quick Start* explains basic usage of the developed programs.

2.2 Quick Start

2.2.1 Robot bringup

Everytime you turn on the robot, follow these few steps:

1. Wait for it to connect to the WIFI network
2. Launch the bringup script on your computer to connect to the robot and launch the necessary nodes:

   ```
   ./start_waffle IP_ADDRESS
   ```

   Replace "IP_ADDRESS" with the IP address of the robot on the local network. This script will connect via SSH and launch the ROS2 nodes to access the sensor data and control the robot. The script will also launch the cartographer (SLAM) node and an RVIZ2 window to start monitoring the robot and mapping its environment.

   Tip: To avoid having to remember the IP address everytime you launch the robot, you can change the default IP address used by the script by changing the "WAFFLE_ADDRESS" variable at the top of the script. Just call

   ```
   ./start_waffle
   ```

   to launch the script with the default address.

   ---

3. Test that the robot is working by trying to control it manually with the keyboard:

   ```
   export TURTLEBOT3_MODEL=waffle_pi
   ros2 run turtlebot3_teleop teleop_keyboard
   ```

   You should be able to make the robot move around.

[4] https://gitlab.ensta-bretagne.fr/hofmanhu/turtlebot-waffle-claw-stage-2a

2.2.2 Controller / Observer Node

The first node developed is the controller/observer node that allows the robot to follow trajectories. At the root of the cloned repository, run:

```
cd Scripts
python3 ControllerObserver.py
```

This will initalize the node, making the robot ready to receive commands (ie, a set of points) to convert into a trajectory to follow. By default the program tries to make the robot reach the goal in a fixed amount of time (ex: 90 seconds).

2.2.3 Path planning

To launch the user interface of the path planner node, run:

```
python3 PathPlannerV2.py
```

A new window should appear:

You should see the map sent by the cartographer node updated in real time.

Note:

- The black pixels represent the obstacles (walls) detected by the robot with its LIDAR.
- The surrounding orange pixels represent the extension of the walls that the robot takes into account in its computations to account for its width and for a "safety" margin.
- The green circle represents the current position of the robot. It should move accordingly when the robot is moving.

Tip: The node displays important and useful information about what is happening in the terminal. A good way to use it would be to have both the map window and the terminal appearing on the screen at the same time.

Basic commands

From there, you can start using the path planner with the following non-exhaustive list of keys:

- press `m` to switch between the different path planning modes
- press `a` to change the algorithm used to compute paths (ex: A* algorithm)
- **double left click** to add a new marker (usually a goal destination depending on the current mode) to the map (represented by a blue circle)
- **double right click** to cancel the last marker or reset the mode
- press `r` to enable/disable *rerouting* mode. If enabled, the robot will try to recompute a new trajectory everytime an unexpected obstacle is detected across the path

In any case each mode has its own set of commands used and you may add some more commands if needed. Refer to the terminal when running the program to see the different keys.

Featured Modes

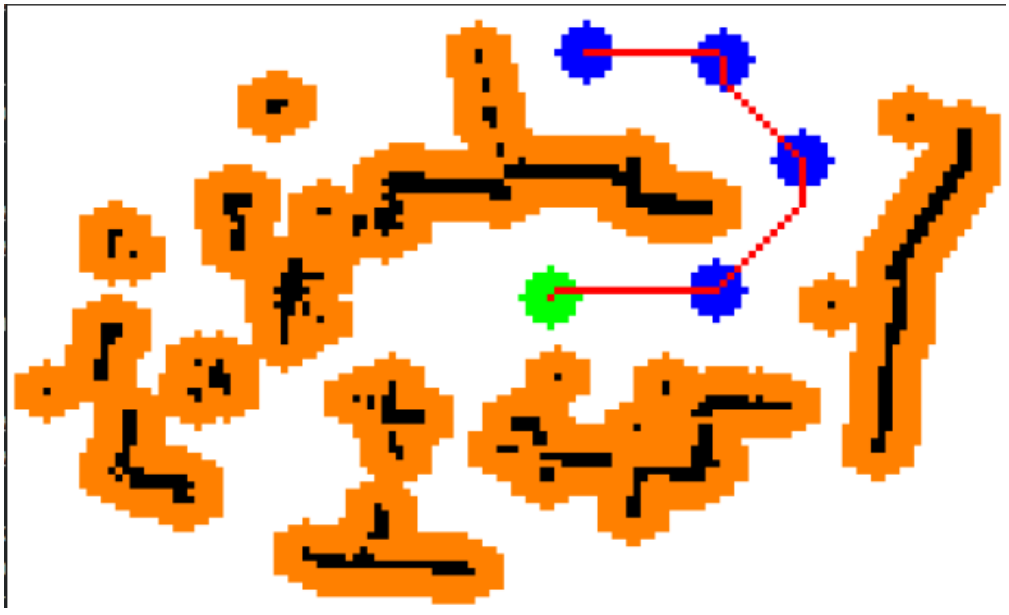
Single Goal Mode

Add a new marker (**double left click**) so the program computes a path from the robot's current position (green circle) to the selected goal. The result should appear as a red path.

> **Warning:** The corresponding set of points is by default immediately sent as a command to the controller node. If the latter is running, make sure you're ready! The controller node will convert the points into a spline function and the robot should then start moving towards its destination.

Checkpoint Mode

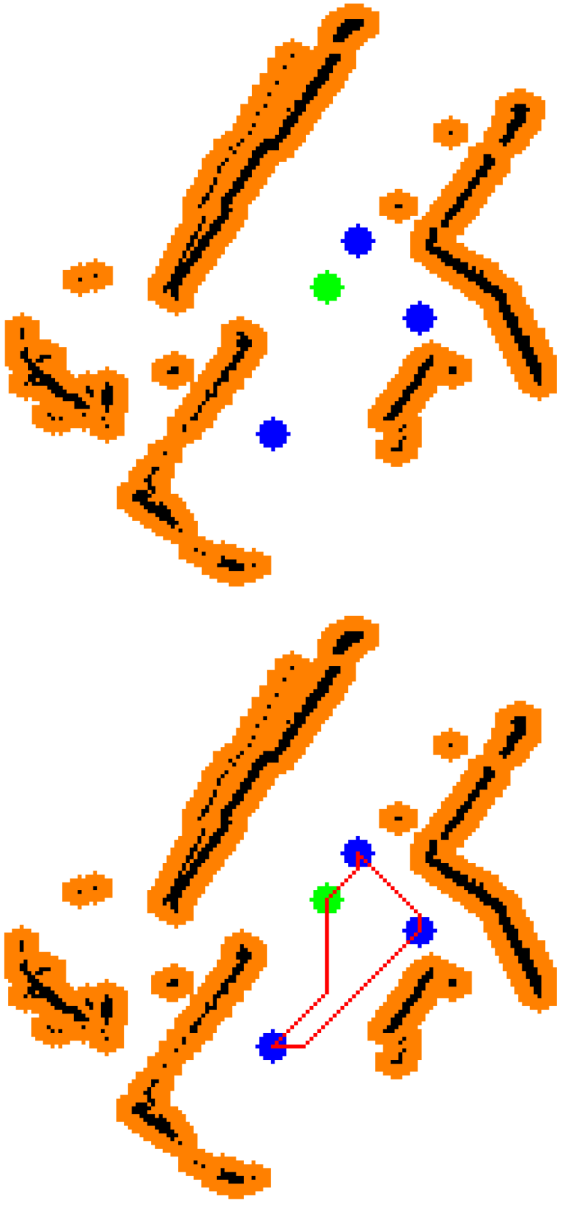
This mode allows the user to compute complex trajectories by adding several checkpoints for the robot to go through (in the same order that they were added). Simply add markers, and you should progressively see the path being constructed.



Press `Enter` to confirm the path and launch the robot.

Travelling Salesman Problem mode

In this mode, the user can add checkpoints just like the previous mode, except this time the order is not necessarily the one the markers were added. Instead, the algorithm attempts to find the optimal path (that is, the shortest) going from the robot's initial position through each checkpoint and back to the start.



Warning: In the current version of the code, as a first test, the method used is brute-force. That means all possible combinations are tested! This makes it completely impossible to compute with too many points (calculations with 4 added markers already take some time), especially since calculating the path for each edge (to get the distance) already takes some time on its own no matter the path searching algorithm used.

Press `Enter` to compute the optimal path.

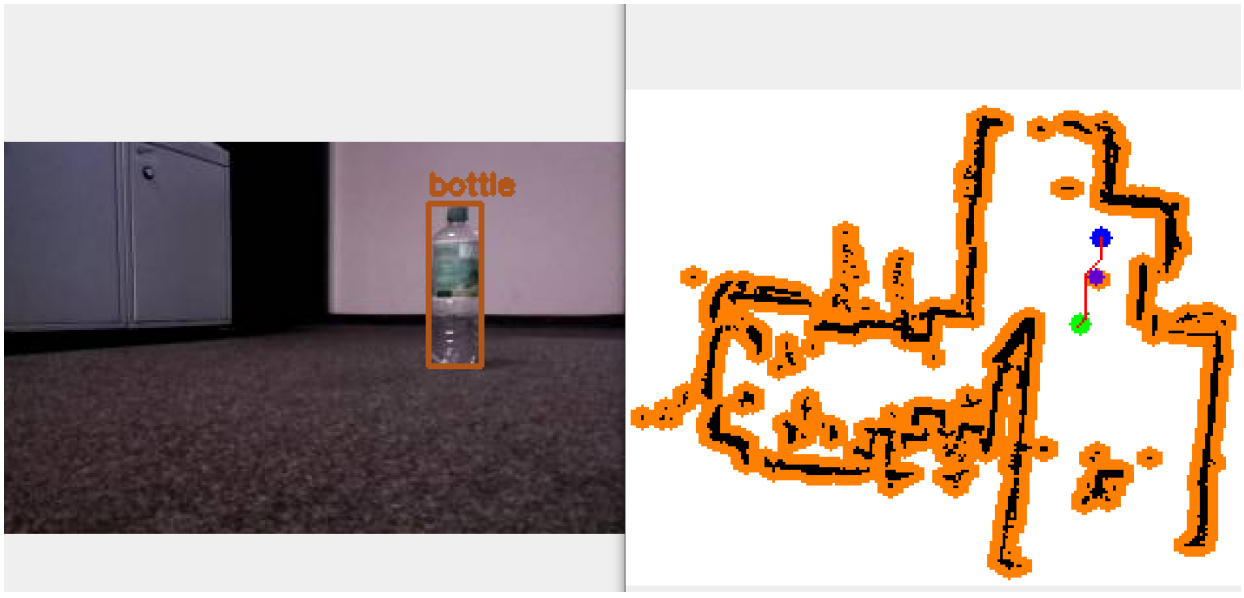
2.2.4 Bottle Recognition

While running the previous Path planning node as explained above, you may run the *CAMERA* module in another terminal with:

```
python3 CAMERA.py
```

This will initalize the subscriber node that receives the camera feed sent by the robot. The image is sent compressed and is uncompressed upon being received to be processed and displayed live (though sometimes with 1 or 2 seconds of latency):

The camera node takes care of processing the image and feeding it as an input to a pre-trained YOLO[5] model which is a popular model capable of detecting various objects. The node detects a bottle and sends its position in the image to the *PathPlannerV2* node which can then compute an estimation its the relative and absolute coordinates with the help of the received information and the LIDAR. The result should appear on the user interface, with a purple dot appearing where the bottle is:



This can be done while the robot is moving around (for example, when following a given trajectory on its own, as in the above picture). In that case the robot should briefly stop to overcome the latency issues and compute its estimation.

[5] https://opencv-tutorial.readthedocs.io/en/latest/yolo/yolo.html

2.3 Adding more features

2.3.1 How to add modes

The Path Planning node was developed with the intent to make it easier to add new features to the robot with a system of modes that the user can easily switch through and play around with to test and debug. We can thus imagine introducing new behaviours that could for example make the robot more autonomous (mapping its environment by choosing its path appropriately to cover as much ground as possible), or simply some features that give more control and possibilities to the user through specific logic functions and algorithms.

In this section is explained how one could use the current API to add new custom features rather easily to the user interface of the path planner node. Adding a new mode should be done by adding a new class inheriting from the class *OpenCV_Interface.Mode* defined in the *OpenCV_Interface.py module*.

Tip: The best way to understand the way the code is supposed to be implemented is to look at the already implemented modes such as *OpenCV_Interface.SingleGoalMode*, in particular for the *OpenCV_Interface.SingleGoalMode.mouse_event()* and *OpenCV_Interface.SingleGoalMode.key_pressed()* method examples which show how to call the path searching algorithms and other useful technical details.

A nice template to start building a new mode would be:

```
class NewMode(Mode):
    def __init__(self, algorithm=find_path_in_potential_field, node=None):
        super().__init__(algorithm, node)
        ## Add Mode Specific Variables

    def description(self):
        ## Print the mode's description (called automatically)
        pass

    def key_pressed(self, key):
        ## Defines the actions defined by the keys ...
        if key == ord("r"): ## Exemple with the "r" key
            print("Action linked to the 'r' key!")

    def mouse_event(self, event, x: int, y: int):
        ## Method being called when the user clicks on the (x,y) pixel on the map␣
↪interface
        if event == cv2.EVENT_LBUTTONDBCLK:
            ## double left click
            pass
        elif event == cv2.EVENT_RBUTTONDBCLK:
            ## double right click
            pass
        elif event ==  cv2.EVENT_LBUTTONDOWN:
            ## left click
            pass
        elif event == cv2.EVENT_RBUTTONDOWN:
            ## right click
            pass

    def handle_reroute(self):
```

(continues on next page)

(continued from previous page)

```
        ## Define what the behaviour should be when the reroute mode is enabled and at␣
↪least one path must be rerouted
        ## For example,
        ## You can iterate through the self.reroute_to_do list to get access to all the␣
↪paths that need to be recomputed
        pass
```

To add the new mode to the set of modes of the interface, look for the `OpenCV_Interface.default_mode_list` variable and add your own mode as a "name"/"type" entry:

```
default_mode_list = {"mode1": Mode1,
                     "mode2": Mode2
                     }
```

You should now be able to run the programs and enable your mode by switching through the mode list by pressing `m`.

2.3.2 How to add path searching algorithms

The project comes with two different algorithms to compute short paths between two points on a map:

- The A* algorithm —> *A_star_utils.plan_path()*
- An algorithm based on Potential Fields and Gradient Descent –> *PotentialFieldsUtils.find_path_in_potential_field()*

The user is free to implement either improved versions of the latter or completely new algorithms. To facilitate integrating the new function into the system, follow the general template:

```
def path_searching_algorithm(img: np.ndarray, start: "tuple[int, int]", goal: "tuple[int,
↪ int]", show: bool=False)->np.ndarray:
    ## Computations...
    ##
    return path
```

where `img` corresponds to the processed binary image of the map with its obstacles (accessed via *OpenCV_Interface.Mode.get_map()*), `start` and `goal` are the (x,y) pixel-coordinates of the two points to find a path between, and `show` is a boolean toggling on the verbose mode for debugging or more (obviously not necessary).

The returned `path` should be a numpy array of the shape $(2, N)$ where N is the number of points constituting the path.

In a similar way as adding new modes, integrating new path searching algorithms into the interface is done by adding said function into a list, here `StateMachine.algorithms`:

```
self.algorithms = [("name alg 1", function1),
                   ("name alg 2", function2)]
```

This should make you able to switch to your new path searching function by pressing the `r` key.

Tip: Switching algorithms automatically changes the `algorithm` variable that you can use in your mode when you want to call the current algorithm's function to compute a path. This is why following the function template makes it easier to add new functions.

2.4 List of modules

2.4.1 A_star_test module

A_star_test.**draw_polyline**(*image*, *path*, *color*)

A_star_test.**mouse_callback**(*event*, *x*, *y*, *flags*, *param*)

2.4.2 A_star_utils module

class A_star_utils.**FilePrioritaire**(*depart:* Noeud)

Bases: list

Sorted List object to easily obtain the minimum element of the list at any time

defiler()

inserer(*n1:* Noeud)

class A_star_utils.**Noeud**(*x=0*, *y=0*, *cout=0*, *heuristique=0*)

Bases: object

Defines a Node, ie a Pixel object within a map (empty of obstacles)

to_array()

A_star_utils.**cheminPlusCourt**(*graphe: list*, *objectif: tuple*, *depart: tuple*, *show=False*)

A_star_utils.**convert_img2maze**(*img*)

A_star_utils.**detect_forced_neighbour**(*x*, *n*, *direction*, *graphe*)

A_star_utils.**diagonal_first_path**(*graphe: ndarray*, *chemin*)

A_star_utils.**direction**(*x*, *n*)

A_star_utils.**distance**(*n1:* Noeud, *goal:* Noeud, *mode_voisin=True*)

A_star_utils.**fill_path**(*path*)

A_star_utils.**from_diagonal_get_straights**(*diag*)

A_star_utils.**identify_successors**(*graphe*, *noeud*, *start*, *goal*)

A_star_utils.**is_in_graphe**(*coord*, *graphe*)

A_star_utils.**jump**(*x*, *direction*, *start*, *goal*, *graphe*)

A_star_utils.**maze2nodes**(*maze: ndarray*, *depart: tuple[int, int]*)

Creates all the Nodes object within the map where there is no obstacle

Args:
maze (np.ndarray): Map image depart (tuple[int, int]): Starting Node position

Returns:
type: _description_

A_star_utils.**norme_vec**(*vec*)

A_star_utils.**plan_path**(*img: ndarray*, *depart: tuple[int, int]*, *arrivee: tuple[int, int]*, *show: bool = False*) → ndarray

Computes a path using the A* algorithm between two points

Args:
img (np.ndarray): Map array depart (tuple[int, int]): Starting node arrivee (tuple[int, int]): Ending node, goal show (bool, optional): Debug mode. Defaults to False.

Returns:
np.ndarray: Array of the points constituting the path

A_star_utils.**prune**(*graphe: ndarray*, *noeud*)

Finds the optimal neighbours based on the JPS variant of the A* algorithm

Args:
graphe (_type_): map noeud (_type_): current node

A_star_utils.**retrouver_chemin**(*n:* Noeud) → list[*Noeud*]

Unpacks the path ending at node n by recursively finding the path backwards

Args:
n (Noeud): Ending node

A_star_utils.**sign**(*x*)

No need to explain this one, right ?

A_star_utils.**unpack_path**(*result*)

2.4.3 CAMERA module

class CAMERA.**ImageSubscriber**(**args: Any*, ***kwargs: Any*)

Bases: Node

detected_bottle(*measure*, *height*, *mode_lidar=False*)

image_callback(*msg*)

send_move_forward(*distance=-0.3*)

CAMERA.**calculate_delta_x_bottle**(*mesures*, *height*)

CAMERA.**calculate_distance**(*mesures*)

CAMERA.**main**(*args=None*)

2.4.4 ControllerObserverNode module

class ControllerObserverNode.**BezierCurve**(*pis: ndarray*)

Bases: object

Defines a Bézier curve linking the first and last points of the provided set

check()

eval(*t: float*)

class ControllerObserverNode.**CubicSplines**(*points: ndarray*, *T: float = 1*)

Bases: object

Defines a Cubic Bézier Spline defined by a series of points in the real world coordinates

eval(*t: float*)

get_control_points(*index: int*)

get_trajectory(*nb_points=200*)

init()

class ControllerObserverNode.**PointGoal**(*x: float*, *y: float*, *margin: float = 0.02*)

Bases: object

Generates a pseudo-trajectory defined by a constant point

eval(*t*)

is_arrived(*robot_pos*)

class ControllerObserverNode.**RobotTest**

Bases: Node

ExtendedKalmanFilter(*X: ndarray*, *uk: ndarray*, *Gamma: ndarray*, *Y*)

control_robot()

detect_obstacle()

estimate_time(*curve*, *x*, *y*, *duration*)

Calcule l'instant t le plus cohérent vis-à-vis de la position du robot par rapport à la spline

Args:

curve (_type_): Spline x (_type_): Position x du robot y (_type_): Position y du robot duration (_type_): Durée T théorique du suivi de la spline curve

Returns:

type: Temps t

euler_from_quaternion(*quat*)

Convert quaternion (w in last place) to euler roll, pitch, yaw. quat = [x, y, z, w]

follow_trajectory()

Computes the right commands to follow the current trajectory using the Feedback Linearization controller

get_turn_90_degrees(*direction*, *current_heading*)

get_turn_angle_relative(*angle*, *current_heading*)

move_forward_callback(*msg*)

observe()

odom_callback(*msg*)

path_callback(*path: nav_msgs.msg.Path*)

sawtooth(*x*)

scan_callback(*msg*)

set_goal_point(*position: tuple*)

set_turning_mode(*heading*)

speed_change_callback(*msg*)

standstill

** Initialisation des publishers et subscribers ROS2

stop(*stop_program=True*, *reset_path=True*)

Arrête le robot et éventuellement le programme

stop_callback(*signal*)

turn_angle(*heading*, *current_heading*)

update_callback()

ControllerObserverNode.**inv_mat_A**(*th*, *z*)

Retourne la matrice A telle que u = A @ v où u est la commande et v l'erreur

ControllerObserverNode.**main**(*args=None*)

ControllerObserverNode.**trajectory_PID_v**(*w*, *wd*, *wdd*, *X*, *Xd*, *k1=1*, *k2=2*)

Retourne v, l'erreur à minimiser dans le PID de suivi d'une trajectoire

Args:

w (array): état désiré (X désiré) à l'instant actuel wd (array): dérivée première de w wdd (array): deuxième dérivée de w X (array): état actuel du système Xd (array): première dérivée de X k1 (int, optional): Coefficient terme proportionnel du PID. Defaults to 1. k2 (int, optional): Coefficient terme dérivé du PID. Defaults to 2.

Returns:

array: erreur v

2.4.5 OpenCV_Interface module

class OpenCV_Interface.**CheckPointMode**(*algorithm=<function find_path_in_potential_field>*, *node=None*)

Bases: *Mode*

Mode where the user can add and remove checkpoints to pass through to create a unique path to send to the robot

Args:

Mode (Mode): Parent class

description()

handle_reroute()

Defines the way reroutes are handled, ie how to recompute a wrong path (varies according to specific mode)

key_pressed(*key*)

mouse_event(*event*, *x: int*, *y: int*)

Mouse event callback to be called when the interface window is clicked on (varies depending on the specific mode)

class OpenCV_Interface.**ExplorationMode**(*algorithm=<function find_path_in_potential_field>*, *node=None*)

Bases: *Mode*

Mode where the robot autonomously attempts to explore its environment by progressively setting goal destinations on the border between the explored regions and the unexplored regions

Args:
Mode (Mode): Parent class

description()

end_of_trajectory()

Defines the behaviour to follow when a trajectory has been fully followed

find_unexplored_region_to_go_to()

handle_reroute()

Defines the way reroutes are handled, ie how to recompute a wrong path (varies according to specific mode)

key_pressed(*key*)

class OpenCV_Interface.**Mode**(*algorithm=<function find_path_in_potential_field>*, *node=None*)

Bases: object

Mode Parent class to describe behaviour of the map interface

add_marker(*x*, *y*, *color=(255, 0, 0)*, *size=4*)

add_path(*path*, *color=(0, 0, 255)*)

add_reroute(*path_to_reroute: Trajectory*)

Adds an unvalid path to the list of paths that need to be rerouted

Args:
path_to_reroute (Trajectory): path to be rerouted

convert_array_map2real(*points: ndarray*)

convert_array_real2map(*points: ndarray*)

convert_map2real(*x: int*, *y: int*, *publish: bool = False*)

convert_real2map(*point: tuple*)

description()

draw_circle(*image: ndarray*, *x: float*, *y: float*, *color: tuple*, *size=4*)

draw_marker(*image*, *marker: SinglePoint*, *check_obstacle=False*)

Draws a circle on the provided image from a Marker

Args:
image (array): Image to draw onto marker (SinglePoint): Point to display on the image check_obstacle (bool, optional): Enables the check_obstacle mode, where the Marker is checked to make sure it does not intersect a wall. Defaults to False.

Returns:
array: Resulting image

draw_path(*image: ndarray*, *path:* Trajectory, *check_obstacle=False*)

Draws a trajectory on the image from a Path (Trajectory) object

Args:
image (np.ndarray): Image to draw onto path (Trajectory): Path to draw check_obstacle (bool, optional): Obstacle Mode. Defaults to False.

Returns:
type: _description_

draw_polyline(*image: ndarray*, *path:* Trajectory, *color: tuple*)

draw_result(*image: ndarray*)

Draws the current markers and paths on the image

Args:
image (array): Image to draw onto

Returns:
array: Result image

end_of_trajectory()

Defines the behaviour to follow when a trajectory has been fully followed

get_map()

get_path_publish_message() → nav_msgs.msg.Path

Concatenates and converts all current paths into a ROS2 Path message ready to be sent to the robot

Returns:
Path: Converted Path ROS2 message

get_robot_real_position()

handle_reroute()

Defines the way reroutes are handled, ie how to recompute a wrong path (varies according to specific mode)

key_pressed(*key*)

mouse_event(*event*, *x: int*, *y: int*)

Mouse event callback to be called when the interface window is clicked on (varies depending on the specific mode)

publish_path()

Publishes all the current paths in one go

reroute_path(*path:* Trajectory)

Recomputes the path with the latest map information available

Args:
path (Trajectory): Path to recompute

reset()

Resets the mode

route_path_from_real_coords(*sx: float*, *sy: float*, *ex: float*, *ey: float*) → *Trajectory*

Generates a new path between two points S and E in real-world coordinates

Args:
sx (float): x coordinate of the S point sy (float): y coordinate of the S point ex (float): x coordinate of the E point ey (float): y coordinate of the E point

Returns:
Path: Computed path between S and E

set_new_algorithm(*new_alg*)

unexpected_obstacle(*path:* Trajectory) → bool

Checks if the path intersects an obstacle (wall)

Args:
path (array): (2,n)-shaped array of the points constituing the path

Returns:
bool: True if an obstacle has been detected on the path, else False

class OpenCV_Interface.**SingleGoalMode**(*algorithm=<function find_path_in_potential_field>*, *node=None*)

Bases: *Mode*

Simple Mode where the user can select a point on the map to calculate a path and send it to the robot

Args:
Mode (Mode): Mode parent class

description()

handle_reroute()

Defines the way reroutes are handled, ie how to recompute a wrong path (varies according to specific mode)

key_pressed(*key*)

mouse_event(*event*, *x*, *y*)

Mouse event callback to be called when the interface window is clicked on (varies depending on the specific mode)

class OpenCV_Interface.**SinglePoint**(*x*, *y*, *color*, *size=4*)

Bases: object

Defines a (x,y) point in real-world coordinates, along with its display color

class OpenCV_Interface.**StateMachine**(*modes={'TSP': <class 'OpenCV_Interface.UnorderedCheckPoint'>, 'checkpoint': <class 'OpenCV_Interface.CheckPointMode'>, 'classic': <class 'OpenCV_Interface.SingleGoalMode'>, 'exploration': <class 'OpenCV_Interface.ExplorationMode'>}*, *node=None*)

Bases: object

Mode Manager object, takes care of interfacing with the Path planning node, and switching modes/algorithms

current_algorithm_name()

draw_result(*image*)

end_of_trajectory_signal_callback()

explaination_text()

handle_reroute()

key_pressed(*key: int*)

mouse_event(*event*, *x*, *y*)

next_mode()

speed_change(*change=10*)

switch_algorithm()

switch_mode(*new_mode: str*)

class OpenCV_Interface.**Trajectory**(*path*, *color*)

Bases: object

Defines a path in real-world coordinates

class OpenCV_Interface.**UnorderedCheckPoint**(*algorithm=<function find_path_in_potential_field>*, *node=None*)

Bases: *Mode*

Mode where the user can add and remove checkpoints for the robot to calculate a path going through all of them and back to the starting point

Args:
Mode (Mode): Parent class

calculate_path()

description()

handle_reroute()

Defines the way reroutes are handled, ie how to recompute a wrong path (varies according to specific mode)

key_pressed(*key*)

mouse_event(*event*, *x*, *y*)

Mouse event callback to be called when the interface window is clicked on (varies depending on the specific mode)

OpenCV_Interface.**find_border_pixels**(*image*)

OpenCV_Interface.**find_closest_border**(*image*, *predefined_pixel*)

Returns the closest pixel to the given one among the pixels on the border between the known empty pixels and unexplored pixels

Args:
image (array): image of the map divided into 3 categories (unexplored, walls, explored empty) predefined_pixel (tuple): reference pixel to compare the distance (typically corresponding to the robot's position)

Returns:
tuple: coordinates of the closest pixel found

2.4.6 PathPlanner module

class PathPlanner.**BezierCurve**(*pis*)

Bases: object

check()

eval(*t*)

class PathPlanner.**CubicSplines**(*points*, *T=1*)

Bases: object

eval(*t*)

get_control_points(*index*)

get_trajectory(*nb_points=300*)

init()

class PathPlanner.**MapSubscriberNode**

Bases: Node

convert_array_map2real(*points*)

convert_array_real2map(*points*)

convert_map2real(*x*, *y*, *publish=False*)

convert_real2map(*point*)

euler_from_quaternion(*quat*)

Convert quaternion (w in last place) to euler roll, pitch, yaw. quat = [x, y, z, w]

get_map_info(*info*)

handle_path(*x*, *y*)

map_callback(*data*)

mouse_callback(*event*, *x*, *y*, *flags*, *param*)

odom_callback(*msg*)

process_map()

publish_path(*path*)

stop()

timer_callback()

update_display(*display*)

PathPlanner.**main**(*args=None*)

2.4.7 PathPlannerV2 module

class PathPlannerV2.**MapSubscriberNode**

Bases: Node

add_bottle(*pos*)

bottle_callback(*msg*)

bottle_lidar_callback(*msg*)

controller_info_callback(*msg*)

convert_array_map2real(*points*)

convert_array_real2map(*points*)

convert_map2real(*x*, *y*, *publish=False*)

convert_real2map(*point*)

Convert (x,y) coordinates from the world frame to the image (map) frame

Args:
point (tuple): Point to convert

Returns:
tuple: (X, Y) pixel coordinates in the map

euler_from_quaternion(*quat*)

Convert quaternion (w in last place) to euler roll, pitch, yaw. quat = [x, y, z, w]

get_map_info(*info*)

map_callback(*data*)

mouse_callback(*event*, *x*, *y*, *flags*, *param*)

odom_callback(*msg*)

process_map()

Processes the map image to be ready to compute trajectories with it

sawtooth(*x*)

scan_callback(*msg*)

stop(*stop_program=False*, *halt_vehicle=True*)

timer_callback()

update_display(*display*)

PathPlannerV2.**main**(*args=None*)

2.4.8 PotentialFieldsUtils module

PotentialFieldsUtils.**attraction_force**(*src*, *pos*, *C=1*)

Calcule une composante d'attraction circulaire autour du point source en fonction de la distance au point

Args:
src (array): point attractif pos (any): tableau des positions sur lesquelles calculer les valeurs de potentiel (sous la forme [X, Y]) C (float, optional): Coefficient de proportionnalité. Defaults to 1.

Returns:
array: matrice des valeurs obtenues à rajouter à la matrice principale des potentiels

PotentialFieldsUtils.**boundary**(*p1*, *p2*, *C=1*)

Calcule une composante de répulsion par rapport à une droite définie par deux points en fonction de la distance à cette droite (notamment utile pour définir les frontières extérieures de la zone)

Args:
p1 (array): Point de contrôle de la droite p2 (array): Point de contrôle de la droite C (float, optional): Coefficient de contrôle de l'intensité du potentiel. Defaults to 1.

Returns:
array: matrice des valeurs obtenues à rajouter à la matrice principale des potentiels

PotentialFieldsUtils.**detect_local_minima**(*Z*)

Retourne les minimums locaux du champ de potentiel passé en paramètre

Args:
Z (array): matrice du champ de potentiel

Returns:
array: matrice (N,2) des minimums locaux où N est le nombre des minimums

PotentialFieldsUtils.**find_path_in_potential_field**(*img: ndarray*, *start: tuple[int, int]*, *goal: tuple[int, int]*, *attraction: float = 30*, *show: bool = True*) → ndarray

Calcule un chemin entre deux points en suivant la méthode des champs de potentiel

Args:
img (array): image binaire (carte) à considérer start (tuple): coordonnées du point de départ goal (tuple): coordonnées du point d'arrivée

Returns:
array: matrice contenant le chemin calculé

PotentialFieldsUtils.**gradient_descent_in_potential_field**(*field*, *x0*, *goal*, *T*, *local_minima*)

Simule la descente de gradient dans le champ de potentiel pour atteindre l'objectif donné

Args:
field (array): _description_ x0 (list): position initiale, départ goal (list): point d'arrivée, objectif T (int): Nombre maximal de points autorisé pouur le chemin local_minima (array): matrice des minimums locaux du champ de potentiel

Returns:
array: matrice contenant le chemin calculé

PotentialFieldsUtils.**is_real**(*cell*, *array*)

Teste si les coordonnées cell correspondent bien à une vrai cellule

PotentialFieldsUtils.**repulsive_force**(*src*, *pos*, *pmax=1*, *coeff=1*, *delta=1*)

Calcule une composante de répulsion autour du point source en fonction de la distance au point

Args:

src (array): Point répulsif pos (any): Tableau des positions sur lesquelles calculer les valeurs de potentiel (sous la forme [X, Y]) pmax (float, optional): Coefficient proportionnel à la valeur maximale de potentiel atteinte. Defaults to 1. coeff (float, optional): Coefficient de diffusion de la répulsion. Defaults to 1. delta (float, optional): Coefficient de diffusion de la répulsion (règle la valeur de lorsque distance -> 0). Defaults to 1.

Returns:

array: matrice des valeurs obtenues à rajouter à la matrice principale des potentiels

2.4.9 TSP_utils module

class TSP_utils.**Edge**(*pt1*, *pt2*, *map*)

Bases: object

Defines an edge with its two points

dist() → float

TSP_utils.**brute_force**(*pts: ndarray*, *map: ndarray*)

Computes the optimal TSP trajectory by calculating all possibilities (only usable with very few points)

Args:

pts (np.ndarray): Set of points to join map (np.ndarray): Current map of the obstacles

Returns:

type: _description_

TSP_utils.**get_unique_permutations**(*lst*)

TSP_utils.**initialize_all_edges**(*P*)

2.4.10 cv2_experiments module

cv2_experiments.**calculate_distance**(*mesures*)

2.4.11 cv2_haarcascade_classifier_dataset_maker module

2.4.12 cv2_haarcascade_classifier_trainer module

2.4.13 node_camera module

class node_camera.**CameraPublisher**

Bases: Node

get_frame()

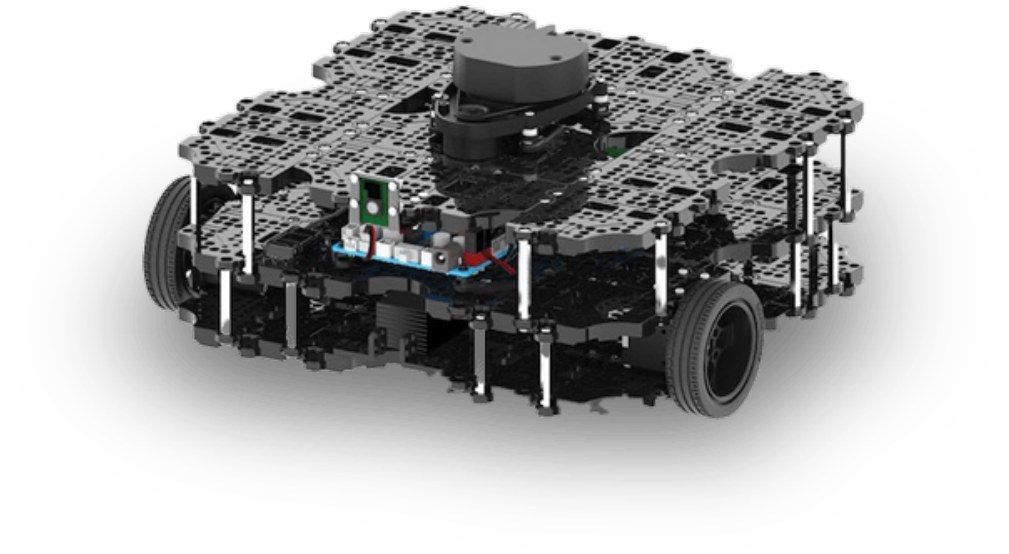
init_camera()

timer_callback()

node_camera.**main**(*args=None*)

2.4.14 opencv_drawing module

opencv_drawing.**line_drawing**(*event*, *x*, *y*, *flags*, *param*)



CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX

D

E

F

G

O

P

R

S

T

U

ENSTA Bretagne

RAPPORT D'EVALUATION
ASSESSMENT REPORT

Merci de retourner ce rapport par courrier ou par voie électronique en fin du stage à :
At the end of the internship, please return this report via mail or email to:

ENSTA Bretagne – Bureau des stages - 2 rue François Verny - 29806 BREST cedex 9 – FRANCE
☎ 00.33 (0) 2.98.34.87.70 / stages@ensta-bretagne.fr

I - ORGANISME / *HOST ORGANISATION*

NOM / *Name* Carl Von Ossietzky University

Adresse / *Address* Ammerländer Heerstraße 114-118, 26129 Oldenburg, Germany

Tél / *Phone (including country and area code)* _ +49 441 798-4195

Nom du superviseur / *Name of internship supervisor*
Prof. Dr.-Ing. habil. Andreas Rauh
Fonction / *Function* _ Head of the Group „Distributed Control in Interconnected Systems" and Dean of the School II Computing Science, Buisiness Administration, Economics, and Law

Adresse e-mail / *E-mail address* andreas.rauh@uni-oldenburg.de

Nom du stagiaire accueilli / *Name of intern* Hugo HOFMANN

II - EVALUATION / *ASSESSMENT*

Veuillez attribuer une note, en encerclant la lettre appropriée, pour chacune des caractéristiques suivantes. Cette note devra se situer entre **A (très bien)** et **F (très faible)**
*Please attribute a mark from **A (excellent)** to **F (very weak).***

MISSION / *TASK*

- La mission de départ a-t-elle été remplie ? (A) B C D E F
  Was the initial contract carried out to your satisfaction?

- Manquait-il au stagiaire des connaissances ? [X] oui/*yes* [ ] non/*no*
  Was the intern lacking skills?

  Si oui, lesquelles ? / *If so, which skills?* /

ESPRIT D'EQUIPE / *TEAM SPIRIT*

- Le stagiaire s'est-il bien intégré dans l'organisme d'accueil (disponible, sérieux, s'est adapté au travail en groupe) / *Did the intern easily integrate the host organisation? (flexible, conscientious, adapted to team work)*

  (A) B C D E F

Souhaitez-vous nous faire part d'observations ou suggestions ? / *If you wish to comment or make a suggestion, please do so here* fully integrated into the research team. The student showed great interest to discuss his achievements with the group members to our full satisfaction

COMPORTEMENT AU TRAVAIL / *BEHAVIOUR TOWARDS WORK*

Le comportement du stagiaire était-il conforme à vos attentes (Ponctuel, ordonné, respectueux, soucieux de participer et d'acquérir de nouvelles connaissances) ?

Did the intern live up to expectations? (Punctual, methodical, responsive to management instructions, attentive to quality, concerned with acquiring new skills)?

(A) B C D E F

Souhaitez-vous nous faire part d'observations ou suggestions ? / *If you wish to comment or make a suggestion, please do so here* quick learning and very eager to acquire new knowledge in accordance with the overall project plan

INITIATIVE – AUTONOMIE / *INITIATIVE – AUTONOMY*

Le stagiaire s'est –il rapidement adapté à de nouvelles situations ? (A) B C D E F
(Proposition de solutions aux problèmes rencontrés, autonomie dans le travail, etc.)

Did the intern adapt well to new situations? (A) B C D E F
(eg. suggested solutions to problems encountered, demonstrated autonomy in his/her job, etc.)

Souhaitez-vous nous faire part d'observations ou suggestions ? / *If you wish to comment or make a suggestion, please do so here* /

CULTUREL – COMMUNICATION / *CULTURAL – COMMUNICATION*

Le stagiaire était-il ouvert, d'une manière générale, à la communication ? (A) B C D E F
Was the intern open to listening and expressing himself /herself?

Souhaitez-vous nous faire part d'observations ou suggestions ? / *If you wish to comment or make a suggestion, please do so here* /

OPINION GLOBALE / *OVERALL ASSESSMENT*

❖ La valeur technique du stagiaire était : (A) B C D E F
Please evaluate the technical skills of the intern:

III - PARTENARIAT FUTUR / *FUTURE PARTNERSHIP*

❖ Etes-vous prêt à accueillir un autre stagiaire l'an prochain ?

Would you be willing to host another intern next year? ☒ oui/*yes* ☐ non/*no*

Fait à Oldenburg, le 02/10/2023
In _____, on _____

Signature Entreprise Andreas Rauh
Company stamp

Carl von Ossietzky Universität Oldenburg
Fakultät II
Informatik, Wirtschafts- und Rechtswissenschaften
Department für Informatik
Abt. Verteilte Regelung in vernetzten Systemen
Prof. Dr. Andreas Rauh
D-26111 Oldenburg

Signature stagiaire *Intern's signature* Hugo Hofmann

Merci pour votre coopération
We thank you very much for your cooperation