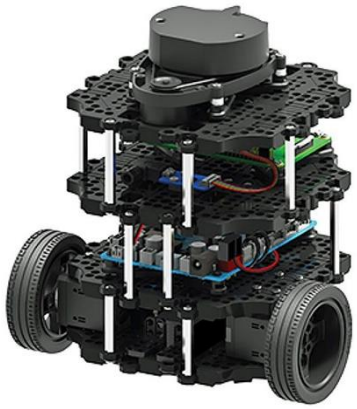
Innovation Project: Final Report

NavigateT: TurtleBot



Students: Marc CHAMBRÉ / Loan MICHAUD / Dhia JENZERI / Reda TRICHA

Training: Electronics, Computing and Systems engineering

Teachers: Ionela PRODAN / Simon GAY

School Year: 2023 - 2024

Table des matières

Abstract

Our robotic navigation project focuses on successfully implementing a PI controller and an obstacle avoidance algorithm for *TurtleBots*, utilizing ROS *2* libraries and *Gazebo* simulation software. After formulating a mathematical model, we initially developed a P-controller, followed by a PI controller. Tests conducted at *Esynov* validated our controllers in terms of path tracking, with enhanced positional accuracy achieved through the integration of infrared cameras. The project's expansion includes the remote control of multiple *TurtleBots* each differentiated by its IP address and namespace, and the integration of an obstacle avoidance algorithm. The project's organization entailed establishing a timeline, dividing tasks, and notably maintaining a documentary trail.

1. Introduction

In the field of robotics, the primary objective of mobile robots is to navigate through environments, whether they are known and mapped or entirely unknown. The purpose of this final report is to showcase our innovative work with a mathematical modeling of a robust system aiming to control one and then multiple robots, implementing a PI-controller and additionally an obstacle avoidance algorithm. A portion of these research efforts has been validated through tests at the *Esynov* platform.

2. Project development plan

2.1 Presentation and first steps of the project

Once the project objectives were introduced by Ionela PRODAN, Simon GAY, and Guillaume SCHLOTTERBECK, we took charge of the project by studying existing works and initiating the configuration of our work environment. To comprehend each stage of the project, we decided to start from scratch by installing the *ROS 2 Humble* and *Gazebo* software/libraries necessary for using the robot. This installation on our personal computers led us to practice Linux commands and conduct initial simulations. To facilitate this delicate and sometimes complex installation stage, we compiled comprehensive documentation, including execution scripts, to facilitate understanding of all these steps.

To understand the necessity of these tools, we define the Robot Operating System (ROS) as a set of tools and software libraries that facilitate the development of robotic applications. This platform enables the creation of nodes for publishing and subscribing to topics, as well as the transfer of data between sensors and actuators. By leveraging pre-existing nodes for *TurtleBot3*, we constructed our application by assembling these components. As for *Gazebo*, it is a 3D simulator widely used for virtual testing, where robots such as the *TurtleBot* burger model, can be simulated in various environments including obstacles.

Subsequently, we utilized a Wi-Fi access point (hotspot) to connect a real *TurtleBot* through a *SSH* link, which has a fixed address. This enabled us to conduct real tests and gain a deeper understanding of the interactions between *ROS* 2, its nodes, and the *TurtleBot*. Following this, we began the mathematical modeling of our system and initiated the implementation of a P controller, followed by a PI controller. Finally, we managed multiple *TurtleBots* using an obstacle avoidance algorithm, which will be detailed in the upcoming sections.

2.2 Mathematical model of the robot

First, we searched for mathematical model to represent the robot. In this way, we represented it as follows:

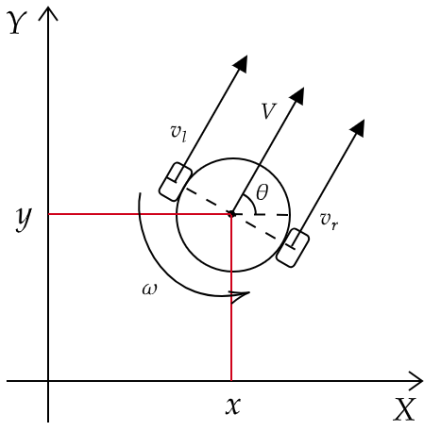


Figure 1 - Turtlebot modeling in a cartesian coordinate system

We can write the dynamics of the robot and the command vector as follows :

$$\xi = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix} \qquad \dot{\xi} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} V\cos\theta \\ V\sin\theta \\ \omega \end{bmatrix} \qquad u = \begin{bmatrix} V \\ \omega \end{bmatrix}$$

We express the speed V_{wheel} of one wheel:

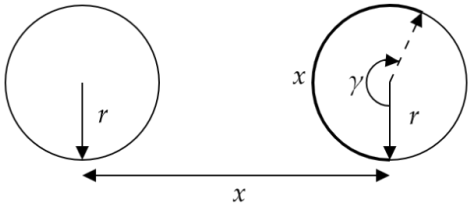


Figure 2 - Distance travelled by a r-radius wheel

We know that the distance travelled by a r-radius wheel is:

$$x(t) = \gamma(t).r \quad \begin{cases} \gamma(t): \text{angle travelled at time t [rad]} \\ x(t): \text{distance travelled at time t [m]} \end{cases}$$

Then, we express the speed as:

$$\dot{x} = \dot{\gamma}r \rightarrow V_{\text{wheel}} = \omega_{\text{wheel}}.r \quad \begin{cases} V_{\text{wheel}}: \text{speed } [\text{m.s}^{-1}] \\ \omega_{\text{wheel}}: \text{angular velocity } [\text{rad.s}^{-1}] \end{cases}$$

Now we define the components of the state vector as a combination of both right and left motor angular velocities, respectively V_r and V_l. We must consider r [m], the radius of each wheel, and L [m], the length between the center of the two wheels.

We assume that the linear velocity of the robot and the angular velocity are written as:

$$V = \frac{V_l + V_r}{2} = \frac{r(\omega_l + \omega_r)}{2} \quad \text{and} \quad \dot{\theta} = \frac{1}{L}(V_r - V_l) = \frac{r}{L}(\omega_r - \omega_l)$$

Finally, we rewrote the dynamic of the state vector as:

$$\dot{\xi} = \begin{bmatrix} \frac{r(\omega_l + \omega_r)}{2}\cos\theta \\ \frac{r(\omega_l + \omega_r)}{2}\sin\theta \\ \frac{r}{L}(\omega_r - \omega_l) \end{bmatrix}$$

2.3 Mathematical design of the controller

Thanks to our previous mathematical model, we started to design a controller. We previously found out this relation:

$$\begin{bmatrix} V \\ \omega \end{bmatrix} = \begin{bmatrix} \frac{r}{2} & \frac{r}{2} \\ \frac{r}{L} & -\frac{r}{L} \end{bmatrix} \begin{bmatrix} \omega_r \\ \omega_l \end{bmatrix}$$

The vector $\begin{bmatrix} \omega_r \\ \omega_l \end{bmatrix}$ being the command of the model, we have to calculate it by reversing the previous expression:

$$\begin{bmatrix} \omega_r \\ \omega_l \end{bmatrix} = \begin{bmatrix} \frac{r}{2} & \frac{r}{2} \\ \frac{r}{L} & -\frac{r}{L} \end{bmatrix}^{-1} \begin{bmatrix} V_{ref} \\ \omega_{ref} \end{bmatrix} = \begin{bmatrix} \frac{1}{r} & \frac{L}{2r} \\ \frac{1}{r} & -\frac{L}{2r} \end{bmatrix} \begin{bmatrix} V_{ref} \\ \omega_{ref} \end{bmatrix}$$

We generate the reference vector $\begin{bmatrix} V_{ref} \\ \omega_{ref} \end{bmatrix}$ by generating the reference velocity using a proportional gain as:

$$v_{ref}(t) = \begin{bmatrix} v_x^r(t) \\ v_y^r(t) \end{bmatrix} = k_p\big(p^r - p(t)\big) = k_p \begin{bmatrix} x(t) - x^r \\ y(t) - y^r \end{bmatrix}$$

Then, we take the module and the argument of this $v_{ref}(t)$ to get the linear velocity V_{ref} and the angle θ_{ref}.

$$V_{ref}(t) = \|v_{ref}(t)\| = k_p\sqrt{\big(x^r - x(t)\big)^2 + \big(y^r - y(t)\big)^2}$$

$$\theta_{ref}(t) = \arg\Big(v_{ref}(t)\Big) = \text{atan2}\big(y^r - y(t); x^r - x(t)\big)$$

Finally, we generated ω_{ref} with another proportional gain: $\omega_{ref} = k_\theta\left(\theta_{ref}(t) - \theta(t)\right)$

Next, we generated the command vector with the relation previously determined:

$$\begin{bmatrix} \omega_r \\ \omega_l \end{bmatrix} = \begin{bmatrix} \frac{1}{r} & \frac{L}{2r} \\ \frac{1}{r} & -\frac{L}{2r} \end{bmatrix} \begin{bmatrix} V_{ref} \\ \omega_{ref} \end{bmatrix}$$

The block diagram is the following:

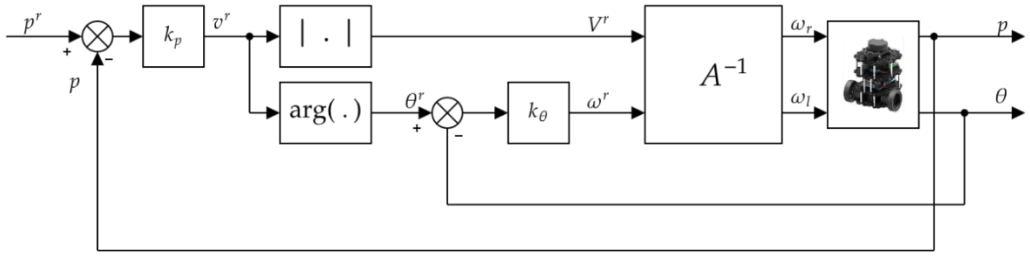


Figure 3 – TurtleBot schematical control loop

The Simulink model is in Appendix 1 and the corresponding Matlab code is available in Appendix 2.

This is the final result of the Matlab code:

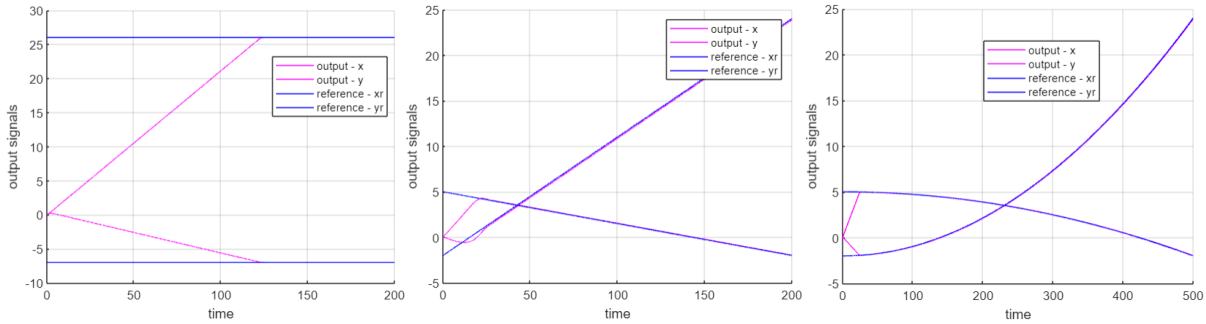


Figure 4 - Setpoint tracking with constant setpoint reaching, linear and quadratic path following

By doing these preliminary tests, we found out that it was primordial to use the function atan2 instead of atan, to consider the signs on x and y. We also found that it was primordial to unwrap the angle of the robot with the reference angle to make the controller choose the shortest angle to the target. We tested some path such a setpoint, a linear path, a quadratic one and an ellipsoidal one, which can be

found in Appendix 3. On the captures above, we can see that the system successfully follows the reference.

2.4 Generating trajectories and controller implementation

Once we determined the mathematical model of our P-controller, we tried to transform it into a *ROS* node in Python to implement it on the real *TurtleBot*. The development of the controller in Python took some time as we went through different stages of complexity (robot saturation limits, angle wrapping issues, requested command not considered, etc.). We were able to detail all the steps of this implementation during the process in our weekly report. To solve these problems, we took a closer look at the manufacturer's data. We saw that, initially, we were giving the robot commands above the robot's saturation limits. With further tests on the real *TurtleBot*, we saw that the robot cannot handle a too fast angular and linear acceleration, so we also saturated them. The manufacturer claims a maximum linear speed of $\pm 0.22\ \mathrm{m.s^{-1}}$ and an angular speed of $\pm 2.82\ \mathrm{rad.s^{-1}}$.

Once we got to a working code, we made simple tests like moving the robot straight forward for 10 seconds or moving the robot from point A to B. The Python codes for the P-controller are available from a GitHub link mentioned in the 3rd part.

We also did trajectory tracking tests. To do that, we tried to generate different trajectories to test our P controller with different possible paths. We generated both linear and random trajectories to visualize the Turtlebot behavior. Some examples of trajectories are available in Appendix 4 and the random generation trajectories Matlab code associated in Appendix 5. We were able to test the P-controller algorithm with the previously generated trajectories, results are in Appendix 6. The results of our P-controller are quite satisfactory, as we can see that the TurtleBot follows the trajectory well, even if there is a tracking error at the corners. We therefore tried to improve our algorithm by adding an integrator to our controller to make a PI. We tested this by adding an integrator on angle and speed on the same trajectories, results are in Appendix 7. According to our results, the PI controller was not much better than the P controller in terms of trajectory tracking. In addition, at the end of each trajectory, the *Turtlebot* tended to turn on itself. On the other hand, we noted that the PI controller performed slightly better than the P controller in terms of average tracking error.

To confirm our results with the data retrieved by *TurtleBot's* odometry topic, we scheduled several sessions at *Esynov* to use the *Qualysis* capture software and gain greater precision in retrieving TurtleBot's position. To retrieve this data, we installed additional packages on our computer to access the data on *ROS 2*, with our controller. Once the installation was complete, we were able to access the "*rigid_bodies*" topic provided by *Qualisys*, which contains the same information as the "*odometry*" topic, and we implemented a function in our PI controller algorithm to listen to and retrieve *TurtleBot* position data in real-time. In this way, we were able to carry out the same trajectory tests, but with data from the *Qualysis* camera systems. The results are shown in Appendix 8. We have grouped the results of our tests in the following table:

$K_{p_{angle}}$	$K_{p_{distance}}$	$K_{i_{angle}}$	$K_{i_{distance}}$	Average tracking error	TurtleBot's behaviour
0.9	0.9	/	/	0.054031	Follows perfectly the trajectory
0.9	0.9	0.02	/	0.54151	Follows perfectly the trajectory
0.9	0.9	0.02	0.05	0.02856	Mainly follows the trajectory but drifts occasionally
0.9	0.9	0.02	0.02	0.041117	Follows perfectly the trajectory
0.9	0.9	0.05	0.05	0.028556	Mainly follows the trajectory but drifts occasionally

According to the tests we carried out in *Esynov* and the results we obtained from the previous table, we have an optimal result for $K_{p_{distance}}$ and $K_{p_{angle}}$ equal to 0.9 and a $K_{i_{distance}}$ and $K_{i_{angle}}$ equal to 0.02. We also noticed that by retrieving the coordinates with the cameras using the "*rigid_bodies*" topic, we no longer had the *Turtlebot's* spinning behaviour and trajectory tracking problems that we had when using the coordinates with the "*odom*" topic. So, we were very pleased to successfully test our *PI controller* with *Esynov* cameras, which provide even greater precision.

2.5 Remote control of multiple TurtleBot

We also worked on the separate control of the *TurtleBots*. When we connected both robots to the same hotspot via a SSH link, both robots were responding to the same commands simultaneously. Running multiple TurtleBots on the same network can be risky due to their shared topic names and node names, which might disrupt their individual functionalities. To enable the smooth operation of multiple TurtleBots within a single network, the objective was first to distinguish them by their IP address and then to distinguish them by assigning a distinct namespace to each robot; to be able to assign different commands to each *TurtleBot*. We created a package on each Raspberry Pi containing a "*bringup.py*" file to initialize motor drivers, sensors, etc. Then, to differentiate them, we customized the names of our *TurtleBots*: *tb3_1* and *tb3_2* in such a way that we could listen to and transmit data on this topic name. By executing a "*rqt_graph*" command (Appendix 9), we could verify that our *TurtleBots* were properly configured, and that we were receiving topics such as *odom, battery_state, scan, cmd_vel*, etc. Once configured, we added the tb3_1 and 2 prefixes in the algorithm of our PI-controller to address the corresponding *TurtleBot's* topic correctly.

Furthermore, we tested both *TurtleBots* simultaneously and we succeeded to control them each one with its own trajectory and functioning. Also, we managed to apply an obstacle avoidance scenario where the two *TurtleBots* avoided each other and continued their trajectory as intended when they met at a certain point.

2.6 Obstacle avoidance algorithm and mapping

Before implementing an obstacle detection and avoidance algorithm, we began to study the functioning of the LiDAR sensor and the recovery of its data. We did some tries in simulation and with the real *Turtlebot*, and we discovered that the data from the LiDAR was not the same.

- **In simulation:**

The LiDAR sends a 360-long table that represents the distance between the LiDAR and obstacles for each degree all around the *Turtlebot*. Also, the 0th element represents the angle 0°, the 180th element represents the angle 180° and so on; with 0° being along the x-axis, in front of the *Turtlebot*. Here is how we get the angle and the distance of a detected obstacle:

$$\text{angle} = \text{index}$$

$$\text{distance} = table(\text{index})$$

with $table(index)$ to be understood in Python, being the element of the array at a given index.

- **With the real *Turtlebot*:**

The LiDAR actually only sends a 230-long table (on average) to represent the whole space. Furthermore, the 0th element is not axed on 0° but on 180°, at the back of the *Turtlebot*. Here is the updated way to retrieve the angle and the distance of a detected obstacle:

$$(\text{angle} + 180)\,[360] = \text{index} \times \frac{360}{\text{length}(table)} \Leftrightarrow \text{angle} = \left(\text{index} \times \frac{360}{\text{length}(table)} - 180\right)[360]$$

$$\text{distance} = table(\text{index})$$

The first objective of obstacle avoidance is to stop in front of obstacles first, and then to avoid them with simple linear trajectories. We must compute a criterion to decide if an obstacle is hazardous or not:

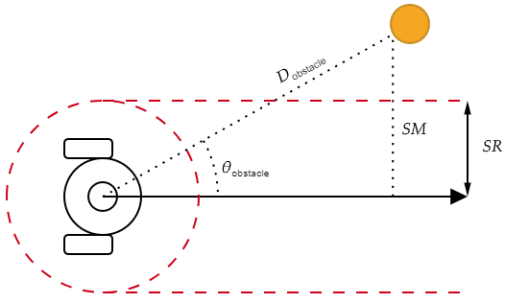


Figure 5 - Detection if the obstacle is in the front field

We defined SR a safety radius around the Turtlebot that ensures the absence of collisions. The LiDAR provides information about the considered obstacle, such as its orientation around the *Turtlebot* θ_{obstacle}, and the distance separating them D_{obstacle}. Then we computed the safety margin SM of the borders of an obstacle:

$$SM_1 = D_{\text{obstacle}_1} \sin(\theta_{\text{obstacle}_1})$$

$$SM_2 = D_{\text{obstacle}_2} \sin(\theta_{\text{obstacle}_2})$$

Finally, we easily defined if the obstacle was hazardous or not by verifying the following condition (you may verify it by looking at the Figure 5) :

```
if abs(SM1) > SR and abs(SM2) > SR :
   hazardous = False
else:
   hazardous = True
```

Then we imagined a simple *emergency* controller that brings the obstacle to be safe. Once the robot has detected a hazardous obstacle, and has stopped, it rotates to change its forward direction :

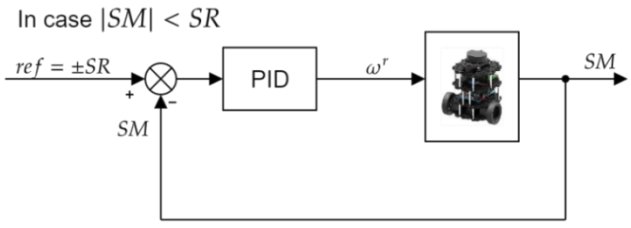


Figure 6 - Emergency controller, only active if |SM|<SR

Using this controller, we defined if the reference is $\pm SR$ using the following algorithm:

```
if SM > 0.0:
   ref = + SR
else:
   ref = - SR
```

For now, this detection is only relevant in the front halfplane because if the trajectory is linear, we don't care about the obstacles behind the robot. We can determine the radius of a circular obstacle by solving the following problem:

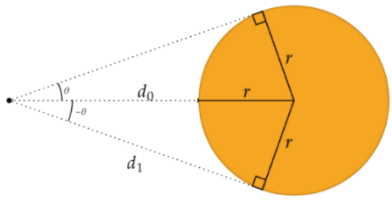


Figure 7 - Calculation of a circle's radius using both tangent lines from an observing point

Using Pythagoras' theorem, we got:

$$(d_0 + r)^2 = d_1^2 + r^2 \Leftrightarrow d_0^2 + 2d_0 r + r^2 - d_1^2 - r^2 = 0 = d_0^2 + 2d_0 r - d_1^2$$

$$r = \frac{d_1^2 - d_0^2}{2d_0} \quad (1)$$

$$r = \frac{d_1^2}{2d_0} - \frac{d_0}{2} \quad (2)$$

We have favoured the second form (2) for the implementation, which is numerically more stable. Using this computation, we can determine where to resume the trajectory after passing the obstacle. For non-cylindric obstacles, we can first approximate them with their circumscribed circle by choosing the greatest value of r computed while passing the obstacle.

If this works well with linear trajectories, the next objective is to decide which side to avoid the obstacle for non-linear trajectories. We can do it by giving the Turtlebot a temporary prediction horizon N_p at time k, and computing the angle between the terminal point $\left(x_{\text{ref}}(k+N_p); y_{\text{ref}}(k+N_p)\right)$ and the *Turtlebot's* position $\left(x(k); y(k)\right)$. Then we can easily define it with the following algorithm:

```
angle_error = atan2(yref(k+Np)-y,xref(k+Np)-x) - theta
if ( angle_error > 0 ):
   ref = + SR
else:
   ref = - SR
```

The reference refers to figures Figure 5 - Detection if the obstacle is in the front field and Figure 6.Here is an example of a situation where we can decide on the side to avoid the obstacle:

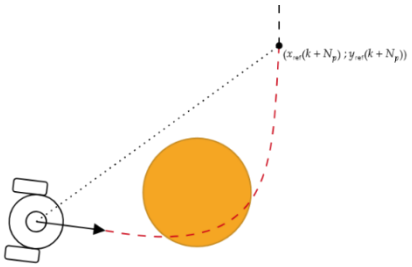


Figure 8 - Turtlebot situation having to avoid an obstacle while looking at a prediction horizon

If we did not care about a prediction horizon, the best side for the *Turtlebot* to avoid the obstacle would have been the one at the bottom. When actually looking at a prediction horizon, the *Turtlebot* can notice that it may be smarter to avoid the obstacle from the top (in the XY plan, obviously). Unfortunately, we didn't have time to implement this prediction horizon in our code.

To get back at the avoidance algorithm, once the Turtlebot has stopped, has chosen the side by which it is going to avoid the obstacle, it keeps the safety distance from the obstacle constant so it follows the border of the obstacle. When the Turtlebot is oriented to the trajectory recovery point, and has fully passed the obstacle, it simply gets back on the trajectory tracking.

Then, we did some obstacle mapping while the robot was navigating in its environment. We processed as following. The LiDAR can provide the distance and the angle in its own coordinates system:

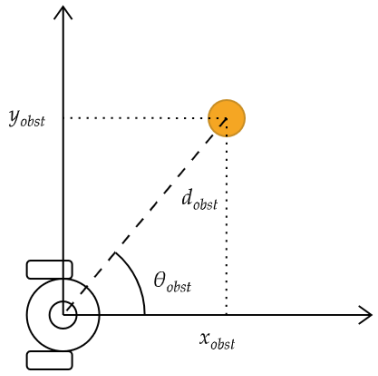
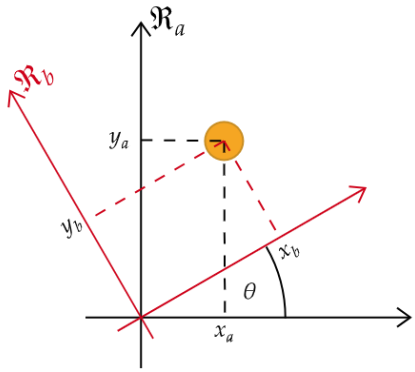


Figure 9 - Coordinates retrieving using distance and angle to target

Using basic trigonometry, we computed $\begin{bmatrix} x_{\text{obst}} \\ y_{\text{obst}} \end{bmatrix} = d \begin{bmatrix} \sin(\theta_{obst}) \\ \cos(\theta_{obst}) \end{bmatrix}$ to get the coordinates of the obstacle within the Turtlebot coordinate system. Once we get these coordinates, we must rotate them to the initial and fixed referential. We know that for any element, we can switch between two coordinate systems using a rotation matrix:



Let's note $p_b = \begin{bmatrix} x_b \\ y_b \end{bmatrix}$ in referential $\Re_b$, and $p_a = \begin{bmatrix} x_a \\ y_a \end{bmatrix}$ in the initial referential $\Re_a$. The rotation matrix is $R_{b\to a}(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$, so we got $p_a = R_{b\to a}(\theta) p_b$. Next, we retrieved the coordinates of the obstacles in the initial referential:

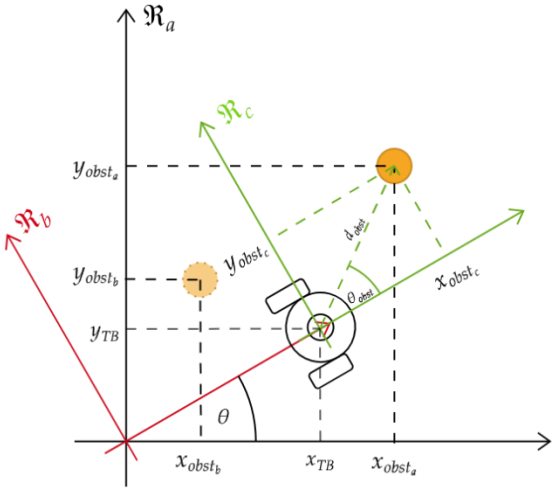


Figure 10 - Obstacle mapping through rotation

With the notations of the previous figure:

$$\begin{bmatrix} x_{\text{obst}_c} \\ y_{\text{obst}_c} \end{bmatrix} = d_{\text{obst}} \begin{bmatrix} \sin(\theta_{obst}) \\ \cos(\theta_{obst}) \end{bmatrix}$$

$$\begin{bmatrix} x_{\text{obst}_b} \\ y_{\text{obst}_b} \end{bmatrix} = R_{b\to a}(\theta) \begin{bmatrix} x_{\text{obst}_c} \\ y_{\text{obst}_c} \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x_{\text{obst}_c} \\ y_{\text{obst}_c} \end{bmatrix}$$

The rotation matrix normally transforms 0-centered rotated coordinates into 0-centered initial coordinates system, but $\begin{bmatrix} x_{\text{obst}_c} \\ y_{\text{obst}_c} \end{bmatrix}$ are in the Turtlebot-centered coordinated system. To get the final $\begin{bmatrix} x_{\text{obst}_a} \\ y_{\text{obst}_a} \end{bmatrix}$ coordinates in the initial referential, we must add them to the *Turtlebot's* coordinates:

$$\begin{bmatrix} x_{\text{obst}_a} \\ y_{\text{obst}_a} \end{bmatrix} = \begin{bmatrix} x_{\text{obst}_b} \\ y_{\text{obst}_b} \end{bmatrix} + \begin{bmatrix} x_{TB} \\ y_{TB} \end{bmatrix}$$

We implemented this in our code and made the robot avoid an obstacle in simulation. This is what we got, in the final plot:

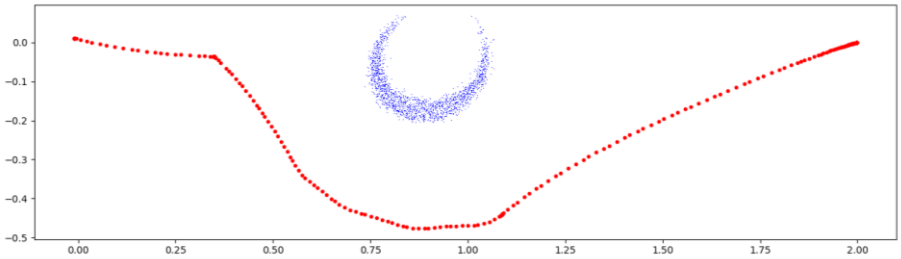


Figure 11 - Obstacle avoidance while mapping the obstacle in a simulation environment

Here, the robot was told to move from $(0;0)$ to $(2;0)$, with an obstacle on his road. Its trajectory is drawn in red, and the computed coordinates of the obstacle are shown in thin blue dots. We can see that it is pretty well shaped. The imprecision of the shaping is due to the precision of the LiDAR in distance and angle estimation, and to the precision of the odometry coordinates and angle. Using this algorithm, we are now able to retrieve the sampled points of the obstacle in two matrixes:

$$x_{\text{obst}} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{bmatrix} \quad y_{\text{obst}} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix}$$

Now, the goal is to perform linear regression to get an approximation of the obstacle using that data. We know the equation of a circle is:

$$(x - x_0)^2 + (y - y_0)^2 = r^2$$

Let's expand the expression and put it in the matrixial form $AX = B$ with X the unknown vector. Here, our unknown variables are x_0, y_0 and r:

$$(x - x_0)^2 + (y - y_0)^2 = x^2 - 2xx_0 + x_0^2 + y^2 - 2yy_0 + y_0^2 - r^2 = 0$$

$$x^2 + y^2 = 2(xx_0 + yy_0) - (x_0^2 + y_0^2) + r^2$$

$$\begin{bmatrix} 2x & 2y & -1 \end{bmatrix} \begin{bmatrix} x_0 \\ y_0 \\ x_0^2 + y_0^2 - r^2 \end{bmatrix} = x^2 + y^2$$

Now that we have the expression in 1D (with one sample), let's bring it to dimension N:

$$\begin{bmatrix} 2x_{\text{obst}} & 2y_{\text{obst}} & -1 \end{bmatrix} \begin{bmatrix} x_0 \\ y_0 \\ x_0^2 + y_0^2 - r^2 \end{bmatrix} = x_{\text{obst}}^2 + y_{\text{obst}}^2$$

$$\begin{bmatrix} 2x_1 & 2y_1 & -1 \\ 2x_2 & 2y_2 & -1 \\ \vdots & \vdots & \vdots \\ 2x_N & 2y_N & -1 \end{bmatrix} \begin{bmatrix} x_0 \\ y_0 \\ x_0^2 + y_0^2 - r^2 \end{bmatrix} = \begin{bmatrix} x_1^2 + y_1^2 \\ x_2^2 + y_1^2 \\ \vdots \\ x_N^2 + y_N^2 \end{bmatrix}$$

So, we got the form $AX = B$. From this matrixial equation, we got N equations for 3 unknowns, so the unique solution X is not computable by doing $X = A^{-1}B$. Indeed, A is a $N \times 3$ matrix, so it does not have any inverse. To solve the problem and get a unique solution, let's write the following equation:

$$A^T AX = A^T B$$

With $A^T A$ being a 3×3 matrix, so the inverse is computable, and we successfully get 3 equations for 3 unknowns. Finally, we compute the unique solution:

$$X = (A^T A)^{-1} A^T B = \begin{bmatrix} X_1 \\ X_2 \\ X_3 \end{bmatrix} = \begin{bmatrix} x_0 \\ y_0 \\ x_0^2 + y_0^2 - r^2 \end{bmatrix}$$

We can now retrieve the presumed characteristics of the circular obstacle:

$$\begin{bmatrix} x_0 \\ y_0 \\ r^2 \end{bmatrix} = \begin{bmatrix} X_1 \\ X_2 \\ \sqrt{X_1^2 + X_2^2 - X_3} \end{bmatrix}$$

The Matlab implementation code of the linear regression of a circular obstacle is available in Appendix 10. The output data of the script are the following:

$$\begin{cases} x_0 = 0.905372 \\ y_0 = -0.036440 \\ r = 0.134957 \end{cases} \quad \text{with} \quad \begin{cases} x_{0_{\text{real}}} = 0.889429 \\ y_{0_{\text{real}}} = -0.044138 \\ r_{\text{real}} = 0.139385 \end{cases}$$

With a maximum error of 1.59 cm on x_0.We plot the sampled point, the computed obstacle and the real obstacle:

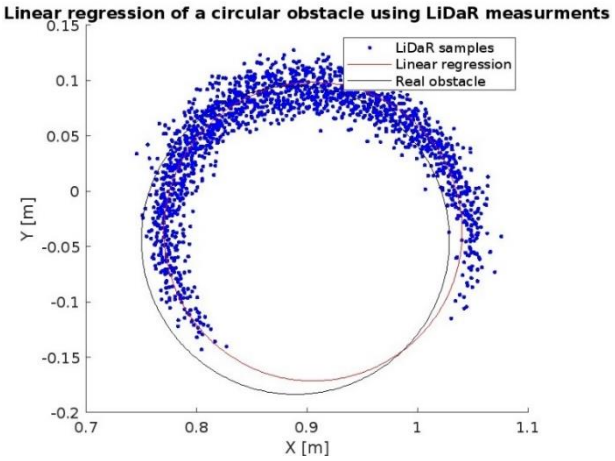


Figure 12 - Linear regression of a circular obstacle

2.7 User interface application

In addition to developing our PI-controller and obstacle detection and avoidance algorithm, we also developed user interface. Indeed, during our tests, we had to manually adjust some parameters: selection of a .csv trajectory, x, y, θ coordinates, display of plots, activation or deactivation of the obstacle avoidance algorithm, and so on. Thus, the user interface offers several fields to be ticked or filled in, and when we press the "OK" button, all these parameters are sent as input to the Python code which contains the code relating to the PI-controller and the obstacle avoidance algorithm.

The user interface is minimalist. The aim is to simplify the options present in our code. There are two main options to check (they cannot be checked at the same time):

- **Reference Tracking:** It allows us to select a .csv file corresponding to a trajectory, which must be previously generated. If this field is checked, we can choose the "Reach start point first & orient robot" option, which allows us to specify whether the TurtleBot should start following the path from its current position or whether it should first reach to the initial position and angle of the trajectory.

- **Setpoint Tracking:** It allows us to manually fill in the x, y and θ coordinates we want to give to the TurtleBot.

Next, a field is provided for selecting the *TurtleBot* which we wish to give a command. So, we can choose between the two *TurtleBots* we've configured, i.e. tb3_1 and tb3_2, the default choice corresponding to a *TurtleBot* with a fixed IP address but not configured with a namespace. We've added red or green dots next to the names of the *TurtleBots* we're using to indicate whether they're connected via SSH. The dot is red if the TurtleBot's IP address is not connected, and green if it is. We've also added several checkboxes:

- **Odometry:** retrieves position data from the *TurtleBot's* internal calculations using motor rotation. If unchecked, this corresponds to the "rigid bodies" section, which retrieves data from the *Esynov* room cameras (Qualisys capture).

- **Real:** this option adds 180° to the data received from the LiDAR. As said before, in the simulation, the 0° LiDAR sensor is at the front of the *TurtleBot*, whereas on the real *TurtleBot* it is at the rear. We therefore add 180° on the real TurtleBot to have the 0° in front of it.

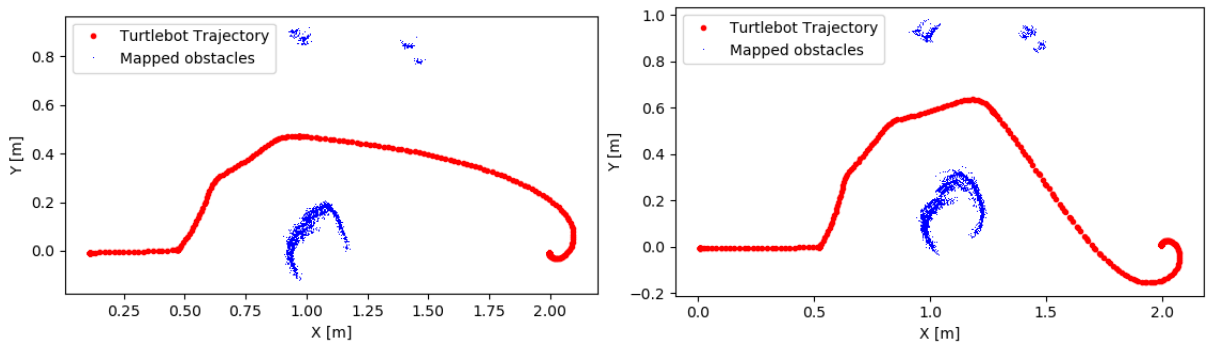
- **Plots:** this option displays data in graphical form at the end of a trajectory.

- **Obstacle avoidance:** this option enables or disables the obstacle avoidance algorithm.

We've also added a "stop process" button to stop the controller's background execution and immediately shut down the TurtleBot. This button can be used when the TurtleBot receives a trajectory to execute, or when it stops responding to commands. We hope this user interface liked to our Python codes will make it easier for future developers to use our obstacle avoidance and control algorithm. The application's graphical interface is in Appendix 11.

2.8 Results and critical explanations

We tested obstacle avoidance on the real TurtleBot using the theory developed above, and our user interface. The main test we carried out was to go from point $(0; 0)$ to point $(2; 0)$, with an obstacle in the way. Here are the results we obtained:



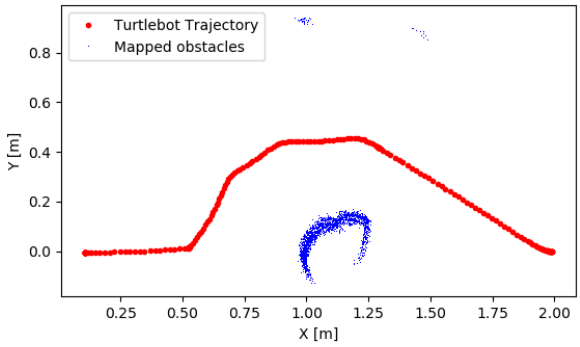


Figure 13 - Different types of results found for obstacle avoidance.

As for the trajectory, some of them were precise, while others took a detour to reach the end point. We found several problems that made the operation less good:

- **Odometry precision**

Odometry accuracy is not very good, and the reference frame shifted a lot over.

- **Speed of the wheels**

We noticed that, for one TurtleBot more than the other, the wheels didn't turn at the same speed when they should. This explains why straight trajectories sometimes become curved. As the odometry reference frame is based on theoretical data for the wheels, and not real data, our algorithm is not able to detect this shift, as odometry doesn't do this either.

- **LiDAR precision**

LiDAR is a major difference between simulation and reality. As we have explained, LiDAR samples space at just 230 points, or one sample every 1.57 degrees. This may not sound like much, but it is, especially when it comes to detecting the edges of obstacles, as this represents an uncertainty of around 5.5 cm for a detection range of 2 meters. Distance estimation is not very good either. As you can see in the screenshots above, the blue obstacle is perfectly circular, but the LiDAR maps it with great uncertainty and is not circular. We noted an accuracy of around 5 cm.

These results are nonetheless very interesting for a first version of an obstacle avoidance algorithm. Nevertheless, our algorithm can still be improved. For now, the obstacle avoidance in only configured for setpoint tracking and not for reference tracking. We thought a lot about it, but we didn’t have time to implement it. What's more, we didn't have the time, but it would have been interesting to test our algorithm with an obstacle in Esynov and compare our curves with the precision of camera systems instead of odometry. Simon Gay also gave us some ideas for improving obstacle avoidance using B-splines. It's true that this is an interesting approach. We started to study this approach during the last PX session, but we didn't have time to go any further into the theory and implementation of this technique.

3. Task management and planning

3.1 Project organisation

To organize our project and keep it on track, we implemented a set of different tools. Firstly, to meet deadlines and track the progress of our project, we decided to create a project schedule in the form of a Gantt chart (APPENDIX). It allowed us to allocate specific days and weeks for each stage of our project and adjust our schedule based on the technical constraints we encountered. The second tool we chose to use is the Trello project management tool. It assisted us in various ways: organizing sessions with checklists, sharing documents, tracking tasks in progress, or completed, distributing

work, and visualizing the overall progress. To access our Trello project, please click on the following link:

https://trello.com/b/4TAB7QMB

To enhance productivity, we established several communication and document backup tools. We set up different channels such as Discord and Messenger to stay in touch and communicate on the project. For document backup, we established a shared Google Drive. Towards the end of the project, we also implemented a GitHub repository to organize our code versions. In the GitHub repository, we included all documentation, tutorials, bash scripts, and ROS nodes to provide maximum technical resources for future individuals working on this project. To access our GitHub project, please click on the following link:

https://github.com/Loan5A/NavigateT-Turtlebot.git

Additionally, to share the progress of our project and motivate ourselves each week, we compiled a weekly report detailing what we accomplished or attempted. This allowed us to maintain a written record of our work and visualize our progression. Each week, we shared this report with Ionela PRODAN, who monitored our progress throughout the project and provided insights at various stages. Without these tools, we would not have been able to make such significant progress in this project. It is important to note that the implementation of these tools is crucial and will be very useful in future professional or personal projects we may undertake.

3.2 Tasks distribution

From the beginning of the project, we decided to divide the project tasks into two distinct teams: the automation team and the programming team. The first team, consisting of Marc CHAMBRÉ and Reda TRICHA, worked on robot modeling and subsequently designed and tuned a controller. They implemented this system on *Matlab/Simulink*, conducted simulations, and found optimal settings. Consequently, their work involved mathematical modeling, controller design, feedback linearization, MPC, and obstacle avoidance.

As for the second team, composed of Dhia JENZERI and Loan MICHAUD, they handled the hardware setup (TurtleBot3) and software configuration (ROS 2, Gazebo, etc.). They conducted initial programming tests where they created ROS nodes in Python and performed simulations in Gazebo. Their objective was to then implement the controller developed by the automation team, manage the control of multiple robots, and work on obstacle avoidance.

In the middle of the project, we decided to merge the two teams because the Python implementation of the nodes was a particularly long and complex step. Marc CHAMBRÉ joined the programming team to provide support, allowing us to progress more efficiently and quickly.

4. Societal innovation challenges

Our robotic navigation project is based on principles aimed at ensuring responsible programming of autonomous systems by the standards of our society. Optimizing trajectories and detecting obstacles, with a focus on energy efficiency, underlines our commitment to Corporate Social Responsibility (CSR). By focusing on sustainable practices, our project boosts technological capabilities while taking a thoughtful approach to environmental impact.

In addition, our emphasis on improving memory optimization algorithms represents a proactive approach to sustainability, thus extending the life of electronic systems and contributing to the reduction of electronic waste, aligned with the overall goal of minimizing our ecological footprint.

The critical focus on safe navigation and obstacle avoidance in our project demonstrates the reliability of autonomous systems. We recognize the societal responsibility to deploy technologies that, beyond meeting their initial objective, actively mitigate potential risks and challenges. Achieving

this balance is essential to building confidence in the adoption of autonomous systems, thus ensuring a positive contribution to society. This approach not only extends the benefits of our robotic navigation technology to various fields, but it also forges a path towards a more sustainable and responsible approach.

Conclusion

To conclude this project, we found it particularly interesting, and it provided us with a wealth of knowledge and skills. It was interesting to start from scratch and to have set up our working environment and a fairly structured organization with a schedule and project management software. In this way, we were able to set up a P and then a PI controller, which we put into practice with generated trajectories. We were also able to test the Esynov platform, which enabled us to use the cameras and Qualysis capture software. Finally, we were able to work on controlling several robots independently and developed an obstacle detection and avoidance algorithm. Given the time available, we're very satisfied with our results, even if there's still room for improvement. We have tried to produce as much documentation and tutorials as possible, to leave a well-constructed piece of work and avenues for improvement for future people working on this project.

Bibliography

- Rajaonson,L. (2023). Internship report for the Systems Design and Integration Laboratory (LCIS).
- Karaki,A. (2023). Internship report for the Systems Design and Integration Laboratory (LCIS).
- Gay, S.; Le Run, K.; Pissaloux, E.; Romeo, K.; Lecompte, C. Towards a Predictive Bio-Inspired Navigation Model. Information 2021, 12, 100. https://dx.doi.org/10.3390/ info12030100
- Remell, K. (2021). Mathematical Modeling of a Two Wheeled Robotic Base.Mechanical Engineering Undergraduate Honors Theses Retrieved from https://scholarworks.uark.edu/meeguht/96
- Vinukonda, N., Madduri, R., Chadha, R., & Desai, V. TurtleBot Path Tracking using PID Controller.

Sitography

- *ROS 2 Documentation: Humble*[online]. Open Robotics, 2024 [accessed September 26, 2023]. Available at: https://docs.ros.org/en/humble

- *Gazebo*[online]. Open Robotics, 2024 [accessed September 26, 2023]. Available at: https://gazebosim.org/docs

- *TurtleBot3 Simulation*[online]. ROBOTIS, 2024 [accessed September 26, 2023]. Available at: https://emanual.robotis.com/docs/en/platform/turtlebot3/simulation/

- *TurtleBot3 Simulation*[online]. ROBOTIS, 2024 [accessed September 26, 2023]. Available at: https://emanual.robotis.com/docs/en/platform/turtlebot3/simulation/

- *Navigation Using Model Predictive Control with TurtleBot3*[online]. Medium, 2024 [accessed October 4, 2023]. Available at: https://emanual.robotis.com/docs/en/platform/turtlebot3/simulation/

- *PID Tuning via Classical Methods Simulation*[online]. LibreTexts Engineering, 2024 [accessed September 26, 2023]. Available at: https://eng.libretexts.org/Bookshelves/Industrial_and_Systems_Engineering/Chemical_Process_Dynamics_and_Controls_(Woolf)

- *ROBOTIS-GIT: TurtleBot3*[online]. GitHub, 2024 [accessed November 17, 2023]. Available at: https://github.com/ROBOTIS-GIT/turtlebot3

- *Tinker-Twins: TurtleBot3*[online]. GitHub, 2024 [accessed October 10, 2023]. Available at: https://github.com/Tinker-Twins/TurtleBot3

- *MOCAP4ROS2-Project*[online]. GitHub, 2024 [accessed November 17, 2023]. Available at: GitHub - MOCAP4ROS2-Project/mocap4ros2_qualisys

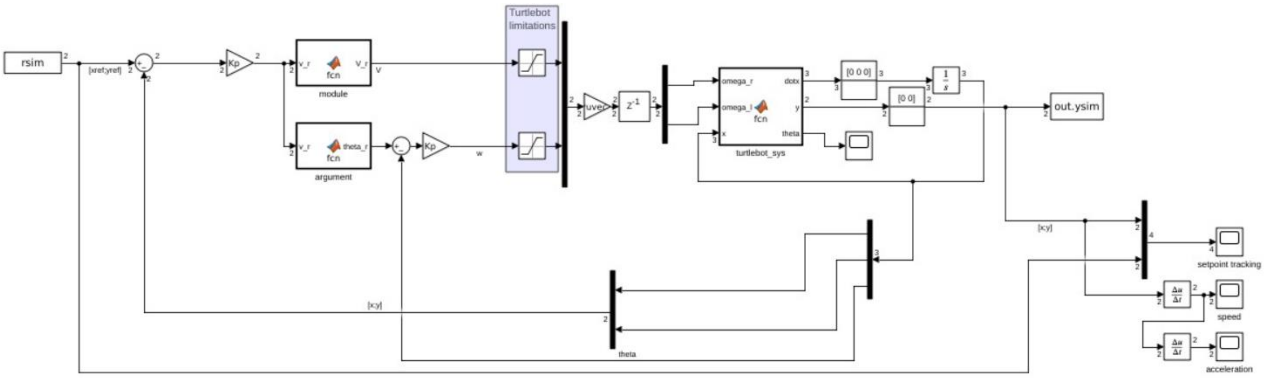
- *Giving a TurtleBot3 a Namespace for Multi-Robot Experiments*[online]. ROS Discourse, 2024 [accessed December 5, 2023]. Available at: https://discourse.ros.org/t/giving-a-turtlebot3-a-namespace-for-multi-robot-experiments/10756

List of Figures

APPENDIX

TABLE OF APPENDIX

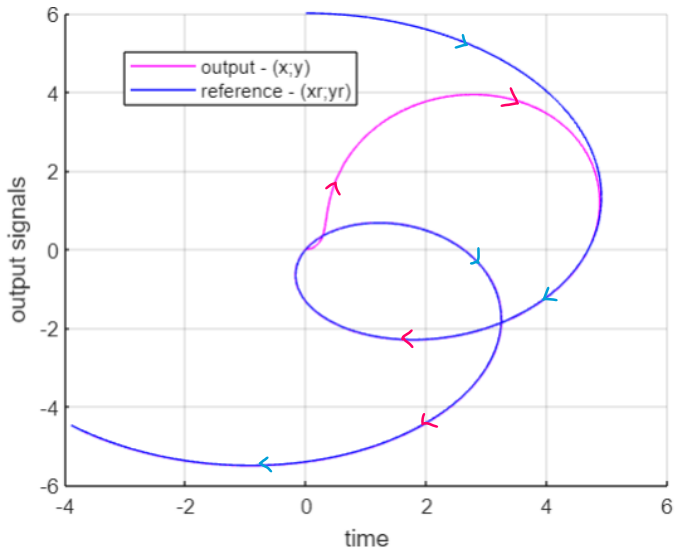
Appendix 1: Simulink model of our P-controller

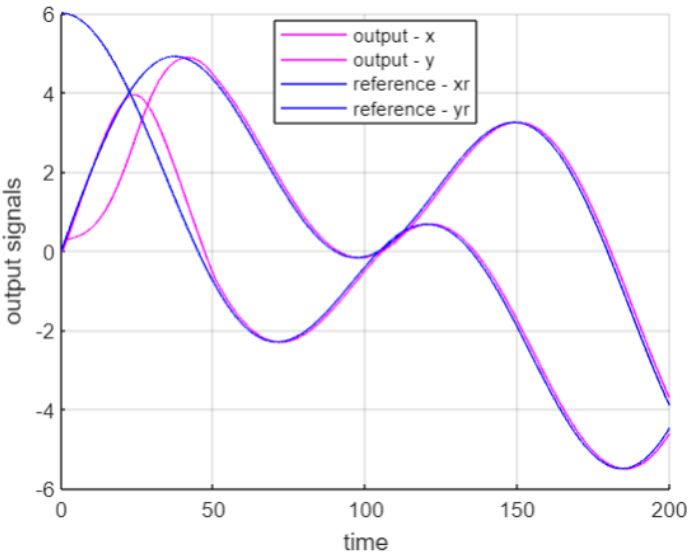


Appendix 2: Control loop Matlab code

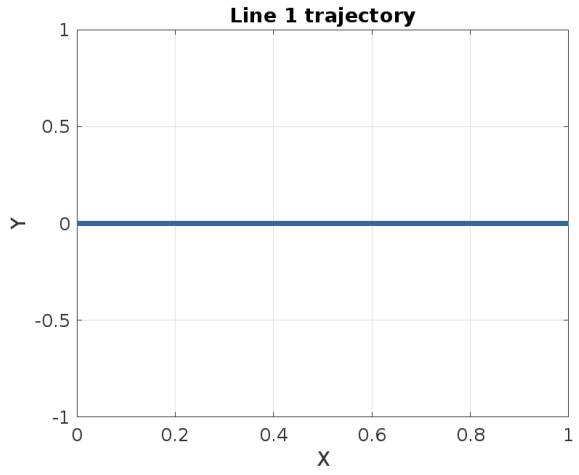
```
clear all; clc; close all;
%% Definition of parameters
% we construct the structure that can be given as an argument to a block
To Workspace
T=200;                              % simulation time
timp=linspace(0,T,1e4);
Ts = timp(2)-timp(1);

Kp=0.9; %Controller gain
r = 33e-3; %Wheel radius
d = 160e-3; %Distance between wheels
A = [r/2 r/2; r/d -r/d]; %[V;w]=A*[wr;wl]
%% Reference for path tracking
%reference end point for constant and linear references
xref=-7;
yref=26;

%Circular references
R=3; f1=0.02; f2=0.05;
position_ref=[R*sin(f1*timp)+R*sin(f2*timp);
R*cos(f1*timp)+R*cos(f2*timp) ];
%position_ref=[(sin(f1*timp));(cos(f1*timp))];
%% turtlebot simulation
load_system('turtlebot');  % we load the simulink model into memory
set_param('turtlebot', 'StopTime', num2str(T)) % set the simulation time

rsim=timeseries(position_ref',timp);   % we build the structure that is
received by the From Workspace block
out=sim('turtlebot');  % we run the simulink model, at the end of the
simulation we have stored the output in the ysim structure
%% plot the results
figure; grid on; hold on
plot(out.ysim.time,out.ysim.signals.values, 'm')
plot(rsim.Time,rsim.Data,'b')
legend('output - x ','output - y ','reference - xr','reference - yr')
xlabel('time')
ylabel('output signals')

figure; grid on; hold on
plot(out.ysim.signals.values(:,1) ,out.ysim.signals.values(:,2), 'm')
plot(rsim.Data(:,1),rsim.Data(:,2),'b')
legend('output - (x;y) ','reference - (xr;yr)')
xlabel('time')
ylabel('output signals')
```

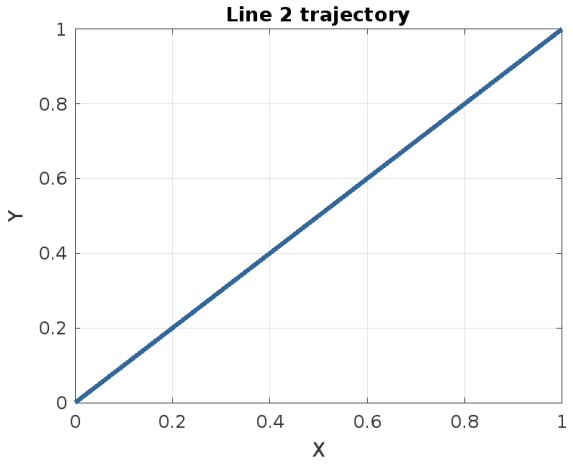
Appendix 3: Elliptical path tracking

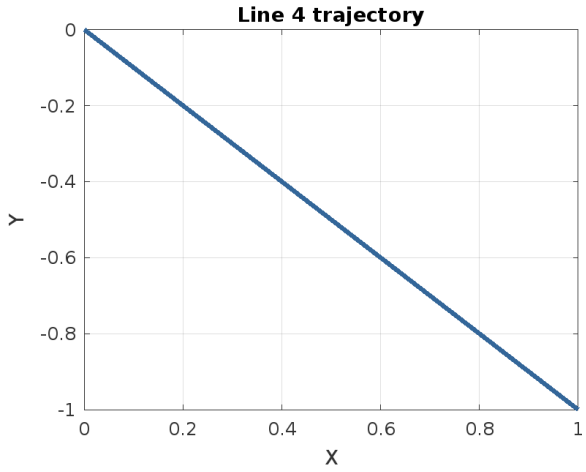


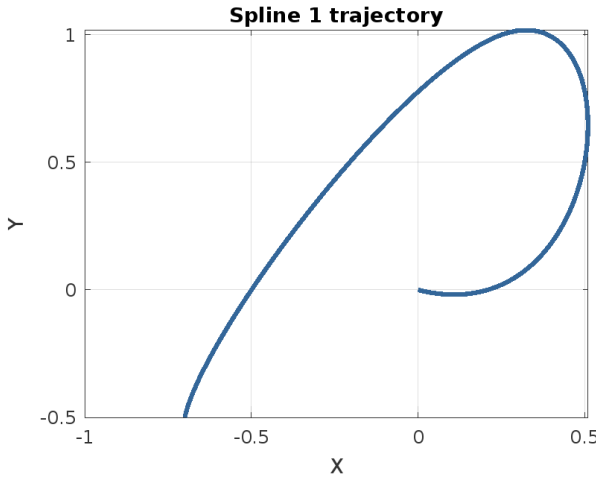


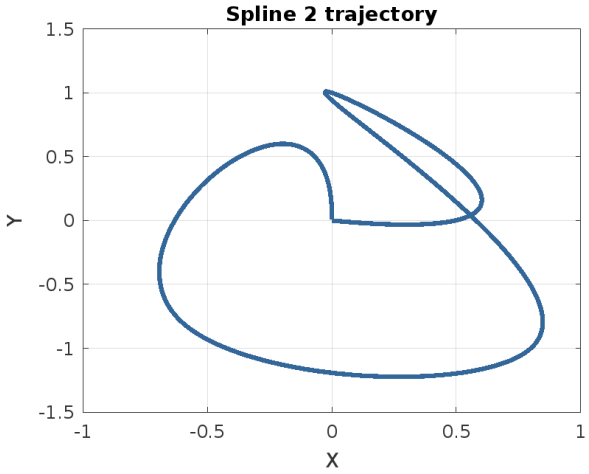
Appendix 4: Examples of trajectories to test our P-controller









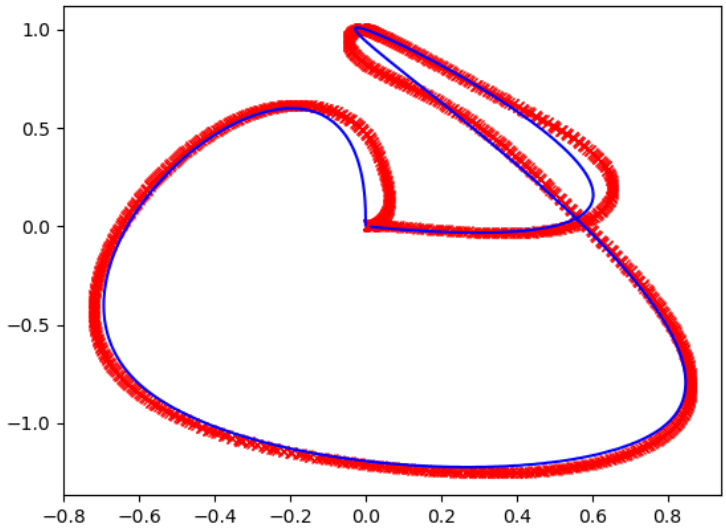


Appendix 5: Matlab script to generate splines through specified points

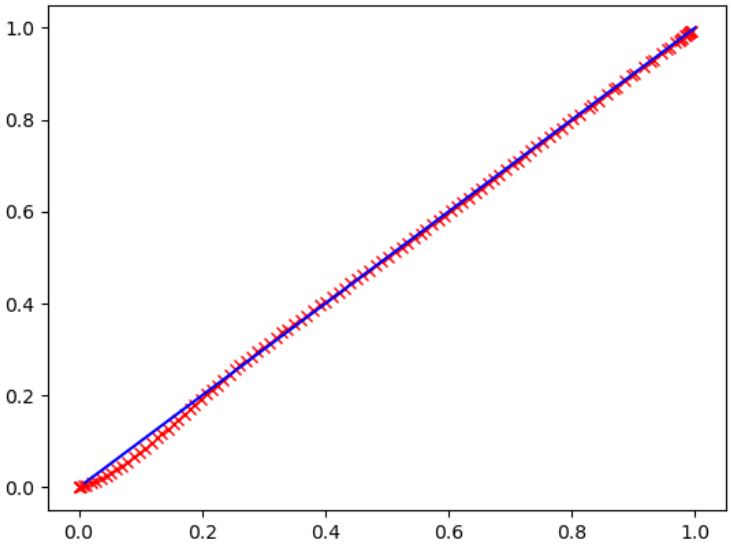
```
clear all; close all; clc;
% Name of the file storing data
file = 'spline.csv'

% Delete "spline.csv" before writing it
% Name of the file storing data
file = 'spline.csv'

% Vérifier si le fichier existe
if exist(file, 'file')
    % Si le fichier existe, le supprimer
    delete(file);
    disp(['Fichier ' file ' supprimé.']);
else
    disp(['Le fichier ' file ' n''existe pas.']);
end

% Define points that the spline will pass through
x = 1/5*[0 3 0 4 -3 -1 0];
y = 1/5*[0 1 5 -5 -4 3 0];

% Sample time
Ts = 0.1;

% Time vector
t = [0 10 20 30 40 50 60]; %Specify time spent reaching each checkpoint
time = 0:Ts:max(t);

% Computing spline
xspline = spline(t,x,time);
yspline = spline(t,y,time);

figure;
plot(xspline, yspline);
hold on;
plot(x,y, 'xr');
title('Trajectory passing through specified points');
xlabel('x_1');
ylabel('x_2');
xlim([min(xspline) max(xspline)]);
ylim([min(yspline) max(yspline)]);

figure;
plot(time, xspline);
hold on;
plot(t,x, 'xr');
title('X component of the trajectory');
xlabel('time (s)');
ylabel('x');
ylim([min(xspline) max(xspline)]);

figure;
plot(time, yspline);
hold on;
plot(t,y, 'xr');
title('Y component of the trajectory');
xlabel('time (s)');
ylabel('y');
ylim([min(yspline) max(yspline)]);
```

```
% Verifying speeds
vx(1) = 0
for i = 1:length(xspline)-1
  vx(i+1) = (xspline(i+1) - xspline(i))/Ts;
end

vy(1) = 0
for i = 1:length(yspline)-1
  vy(i+1) = (yspline(i+1) - yspline(i))/Ts;
end

figure;
plot(time, vx)
title('x-axis speed');
xlabel('time (s)');
ylabel('vx');

figure;
plot(time, vy)
title('y-axis speed');
xlabel('time (s)');
ylabel('vy');

% Exporting to CSV
spline = [xspline', yspline'];
dlmwrite(file,spline,";", "precision", 4);
```

Appendix 6: Testing trajectories generated with the P-controller

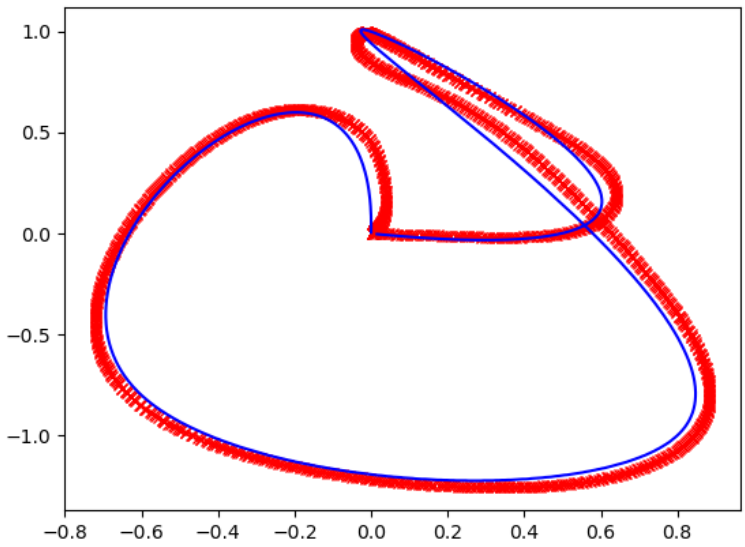


Real Turtlebot following spline 2 trajectory with P Controller

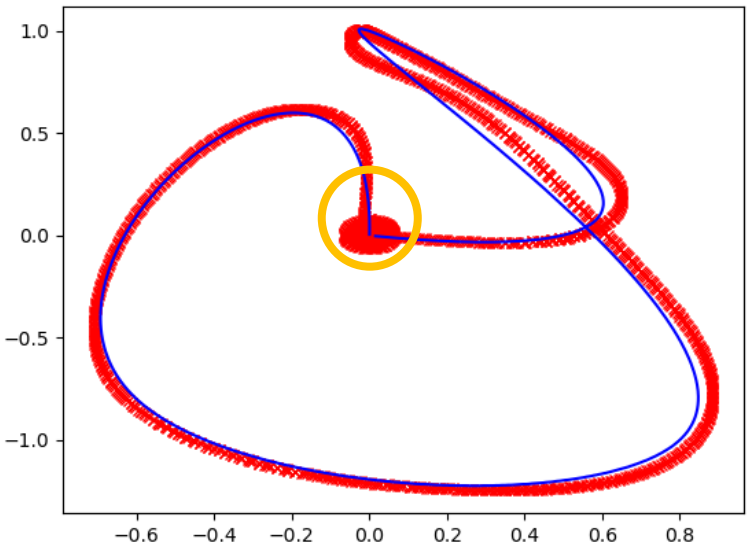


Real Turtlebot following line 2 trajectory with P Controller

Appendix 7: Testing trajectories generated with the PI-controller

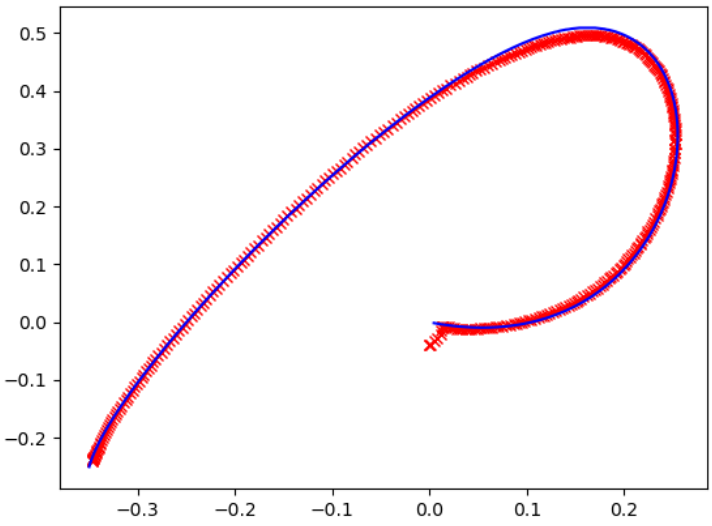


Real Turtlebot following spline 2 trajectory with PI Controller with $K_{i_{angle}} = 0.02$

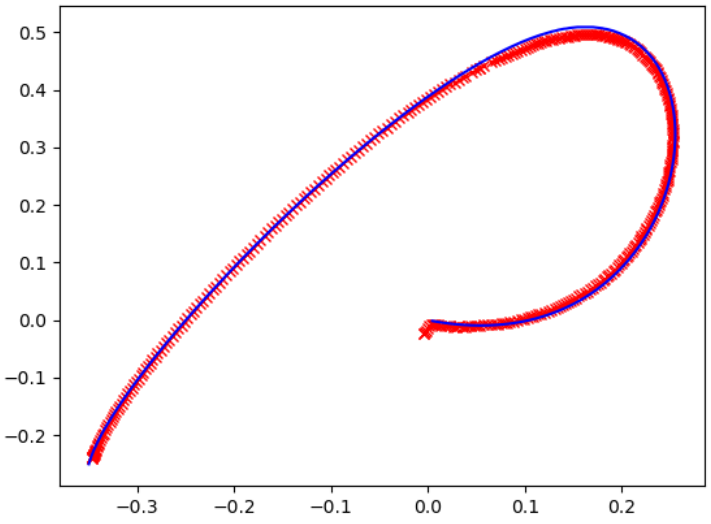


Real Turtlebot following spline 2 trajectory with PI Controller with $K_{i_{angle}} = 0.02$ *and* $K_{i_{distance}} = 0.05$

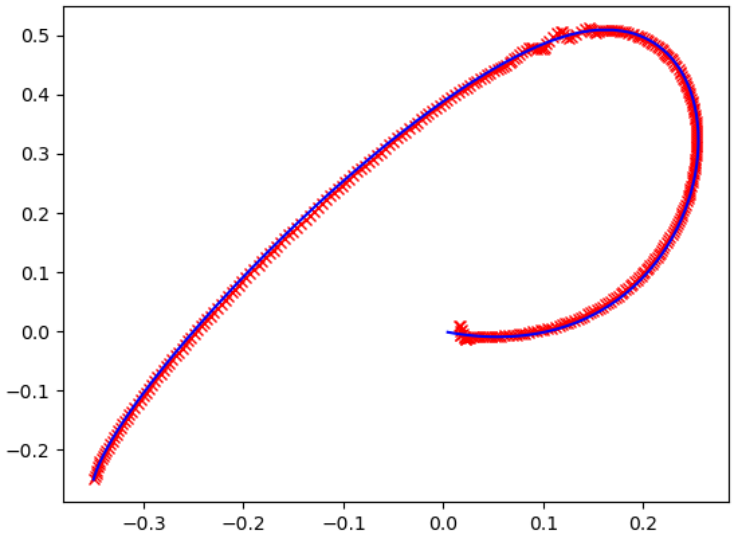
Appendix 8: Testing trajectories with the P and PI-controller at Esynov



Spline with P Controller using camera coordinates

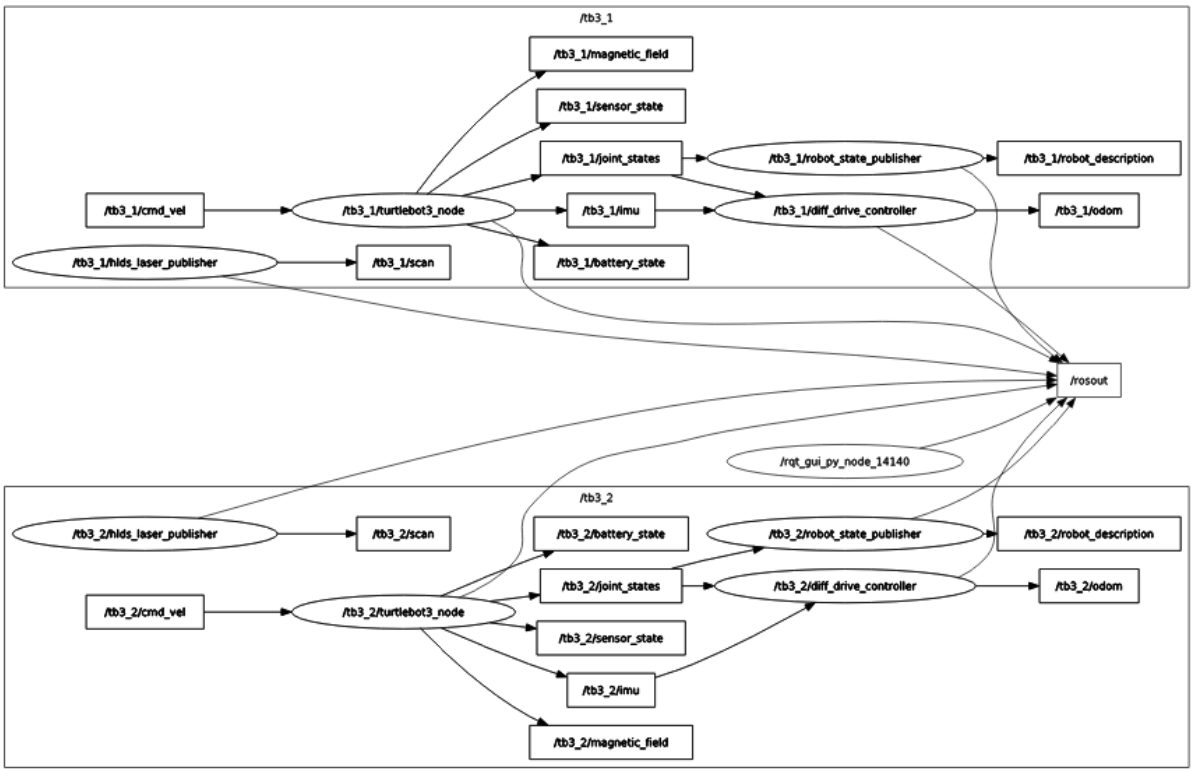


Spline with PI Controller with $K_{i_{angle}} = 0.02$ *and using camera coordinates*



Spline with PI Controller with $K_{i_{angle}} = 0.02$ *and* $K_{i_{distance}} = 0.05$ *and using camera coordinates*

Appendix 9: Separate TurtleBot topics and nodes



Appendix 10: Matlab implementation of the linear regression of a circular obstacle

```
clear all; close all; clc;

% File with datapoints of the obstacle named xcarthography and ycarthography
load output_data.mat;

% Number of points:
N = length(xcarthography)

% xcarthography and ycarthography are row vectors
% We want to solve a matrix equation AX=B with:
A = [-2*xcarthography' -2*ycarthography' ones(N,1)];
B = -(xcarthography').^2 + -(ycarthography').^2;

% We compute A'AX=A'B, so X = inv(A'A)*A'B

X = inv(A'*A)*A'*B;

% Since X=[x0;y0;x0^2+y0^2-r^2], we retrieve x0, y0, and r
x0 = X(1)
y0 = X(2)
r=sqrt(x0^2 + y0^2 - X(3));

% Vectors for the computed circle
theta = 0:2*pi/360:2*pi;
x = (r*cos(theta)+x0);
y = (r*sin(theta)+y0);

% Exact values given in the simulation
r_real = 0.139385;
x0_real = 0.889429;
y0_real = -0.044138;

x_real = (r_real*cos(theta)+x0_real);
y_real = (r_real*sin(theta)+y0_real);

% Plot results
figure; hold on;
plot(xcarthography, ycarthography, '.b');
plot(x,y,'-r');
plot(x_real,y_real,'-k');
daspect([1 1 1])
legend('LiDaR measurments', 'Linear regression', 'Real obstacle')
xlabel('X [m]')
ylabel('Y [m]')
title('Linear regression of a circular obstacle using LiDaR measurments')
saveas(gcf,'Obstacle_linear_regression.png')

r_error = abs(r_real-r)
x0_error = abs(x0_real-x0)
y0_error = abs(y0_real-y0)

errors = [r_error x0_error y0_error];
max_error = max(errors)
```

Appendix 11: User interface application