Autonomous Vehicles course

Report on Assignments

Master of Science in Mechanical Engineering – Mechatronics and Robotics

Author**: Jacopo Porzio**

Student ID: 988718
Lecturers: Prof. Francesco Braghin, Dr Stefano Arrigoni
Academic Year: 2022-23

GitHub Repository Link*:* ***https://github.com/JacopoPorzio/AutonomousVehicles***

CONTENTS

I. Introduction

The following report presents the proposed solutions to assignments and projects for the **Autonomous Vehicles** course, entailing the use of ROS, Gazebo, MATLAB, Simulink and Python. Follows a brief overview of the topics covered in the assignments:

- Assignment 1 (Section II-A): analysis of `rosbag`, estimation of control commands, feedforward control;
- Assignment 2 (Section II-B): feedback tracking of waypoint-based trajectory, Python publisher;
- Assignment 3 (Section II-C): image processing, feedback control;
- Assignment 4 (Section II-D): graph-search methods;
- Assignment 5 (Section II-E): parking gate simulation for Finite State Machines;
- Final Project 2 (Section III-A): optimisation of graph-search method, SLAM with `gmapping`, feedback tracking of trajectory based on map waypoints;
- Final Project 3 (Section III-B): feedback tracking of waypoint-based trajectory and conditional obstacle avoidance.

The source code is publicly accessible on GitHub at the following URL: https://github.com/JacopoPorzio/AutonomousVehicles.

II. Assignments

A. Assignment 1

In the initial assignment, the objective is to analyse a pre-recorded `rosbag` to extract pertinent information. This `rosbag` encompasses odometry data and scans of the environment surrounding the Turtlebot Waffle Pi. The Turtlebot followed a user-defined and unknown trajectory within the standard Gazebo Turtlebot world (see Figure 1).

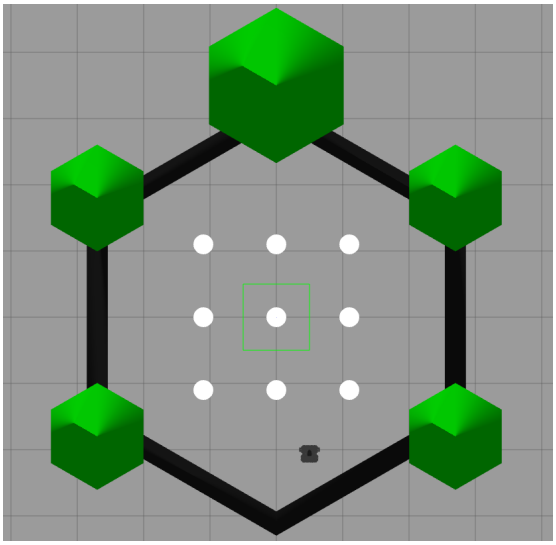

Figure 1: The Waffle Pi inside the reference world.

The analysis starts with estimating the command velocity applied to the Turtlebot. This necessitates identifying both linear and angular displacements. Given the context, only the yaw angle is pertinent, leading us to disregard pitch and roll behaviours. Extracting linear odometry, shown in Figure 2, from the `rosbag` poses no significant challenges. Instead, the angular coordinates (see Figure 3) have to be converted from quaternions into Euler angles.

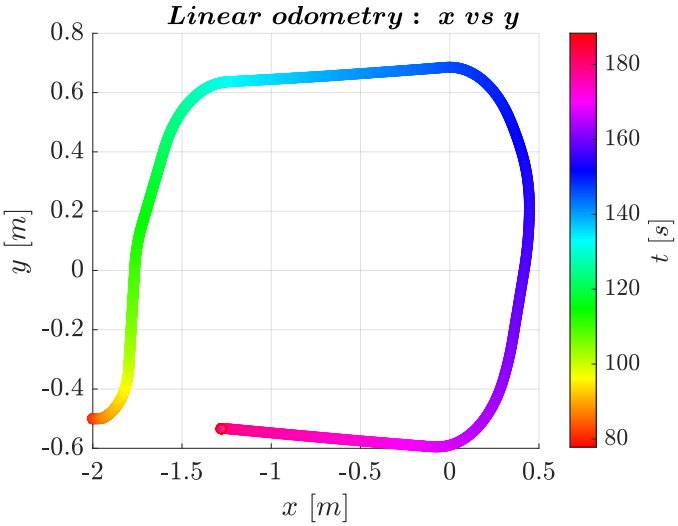

Figure 2: Linear odometry.

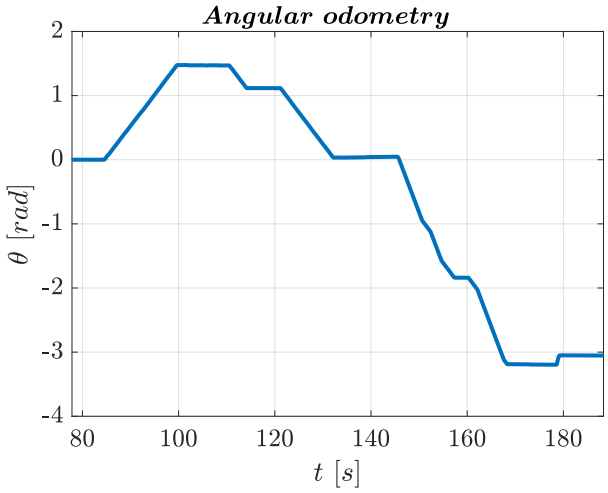

Figure 3: Angular odometry.

After extracting the position information, the analysis progresses to estimate the imposed command velocity. First, it is necessary to estimate the instantaneous linear and angular velocities, whose evolution is reported in Figure 4. This is achieved by approximating velocity using the forward finite difference scheme to the odometry data. Subsequently, a moving average is applied to smooth this initial velocity estimation, reducing noise resulting from numerical differentiation.

Forward finite difference approximation: $\left.\frac{dq}{dq}\right|_k \approx \frac{q_{k+1} - q_k}{\Delta t}$

Moving average scheme: $\bar{q}_k = \frac{1}{k} \sum_{i=n-k+2}^{n+1} q_i$

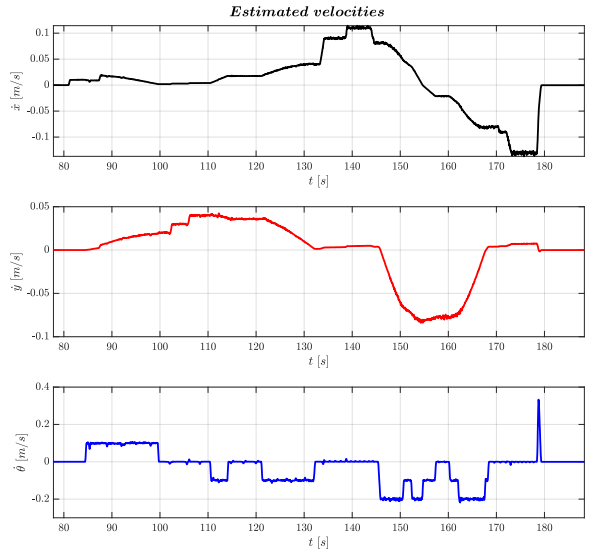

Figure 4: Estimated linear and angular velocities.

The Turtlebot's control inputs consist of longitudinal velocity and angular velocity. For the angular velocity, no further calculation is necessary. However, to obtain the longitudinal velocity, we calculate the Pythagorean addition of the velocities in the x and y directions. Figure 5 shows their estimations.

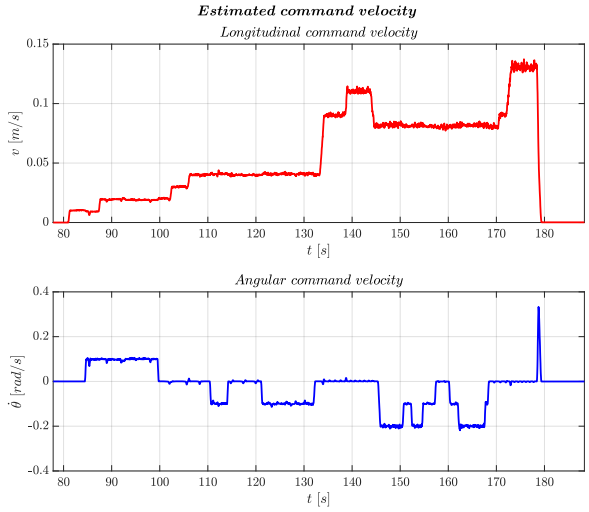

Figure 5: Estimated longitudinal and angular command velocities.

With the estimated command velocities at hand, the next step required from the assignment is to publish them to the robot in a feedforward manner. This enables us to draw conclusions regarding the accuracy of the estimation process. The estimated commands are published to the ROS topic `/cmd_vel` using a Simulink scheme, illustrated in Figure 6. The blocks on the left of the model serve to input two look-up tables (the two blocks with a question mark) with the correct time instant in the

Simulink reference frame. This is achieved by calculating the difference between the actual time instant and the time when the simulation started in the ROS reference frame. Each of the two look-up tables outputs the value at of the desired command velocity, longitudinal and angular respectively, at a given time instant, interpolating it from the estimated command velocities and the associated time steps obtained from the original `rosbag`.

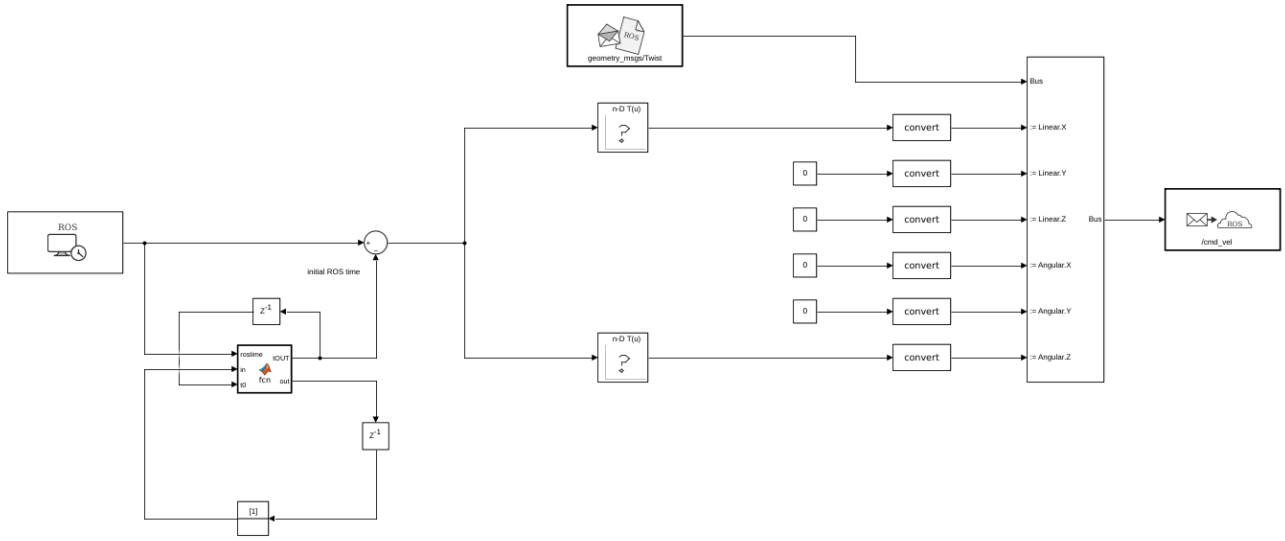

Figure 6: Simulink publication model.

After publishing the command velocities and recording a `rosbag`, the operations conducted for the analysis of the first provided `rosbag` are repeated, in order to draw comparisons between the two. In Figure 7, a slight difference between the original trajectory and the one obtained by imposing the estimated command velocities is evident. This disparity likely arises from the nature of the feedforward imposition and from the fact that the control commands are derived from numerical differentiation and simple moving average filtering may not achieve perfect accuracy. The differences between the original scenario and our attempt to replicate are minimal, as highlighted by Figures 8 to 10, indicating that deviations from the original trajectory stem primarily from the noisy command velocities imposed in a feedforward fashion, compounded by inherent software noise. Additional attempts to mitigate this behaviour by low-pass filtering the command velocity do not yield significant improvements, as the noise levels are relatively minor. In an effort to achieve closer adherence to the original trajectory, a procedure utilising peaks in acceleration to further refine the velocity estimate was explored. However, no significant improvements were observed[1].

The assignment also requires estimating the minimum distance between the Turtlebot and the map obstacles along its original trajectory. This task is straightforward since the scanned space from the LIDAR sensor is available in the `rosbag`. The scanned points are provided in the local Turtlebot reference frame, represented as ranges $\mathbf{scan}_k \in \mathbb{R}^{360}$, with one scanned point per degree. Therefore, the first step is to associate the *polar* information associated to each $\mathbf{scan}_k$ to Cartesian coordinates $\boldsymbol{x}_k$, $\boldsymbol{y}_k$ at each time instant k:

$$\boldsymbol{\theta} = (1, ..., 360)^T$$
$$\boldsymbol{x}_k = \mathbf{scan}_k \odot \cos \boldsymbol{\theta}$$
$$\boldsymbol{y}_k = \mathbf{scan}_k \odot \sin \boldsymbol{\theta}$$

The distance between an obstacle and the Turtlebot at the k^{th} instant is:

$$\boldsymbol{d}_k = \sqrt{\boldsymbol{x}_k^2 + \boldsymbol{y}_k}$$

With this conversion complete, the only remaining operation is to find the minimum value of $textbfd_k$ at each time instant k, whose evolution is reported in Figure 11.

For further insights beyond the scope of the present analysis, it is recommended to refer to the GitHub repository[2]. This

[1]This procedure took advantage of the specific situation under analysis. Traits exhibiting nearly constant velocity were identified in the time evolution of the acceleration by two prominent spikes, marking the beginning and end of each trait. Leveraging this insight, an averaged and constant value was assigned to these traits. However, it is important to note that this process introduces its own margin of error. For further details, please refer to the GitHub repository: https://github.com/JacopoPorzio/AutonomousVehicles.

[2]https://github.com/JacopoPorzio/AutonomousVehicles

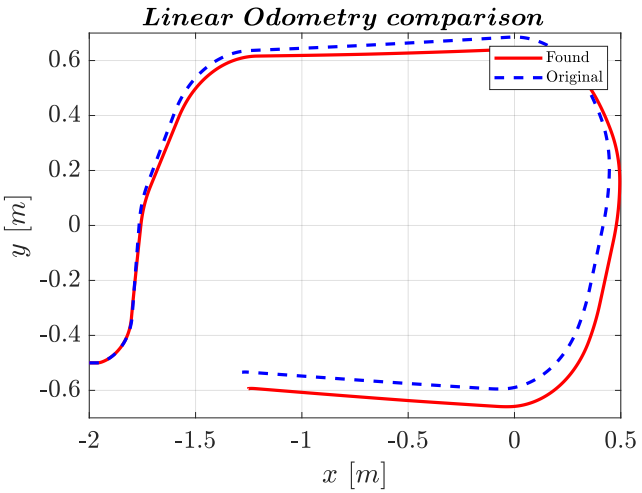

Figure 7: Comparison between the found and original linear odometry.

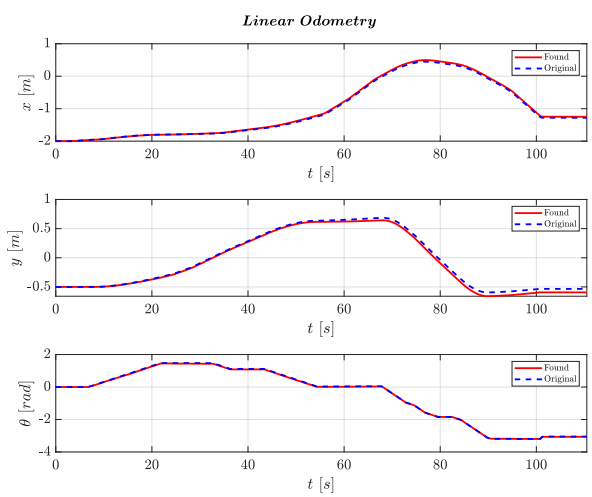

Figure 8: Comparison between the obtained (found) and original odometry.

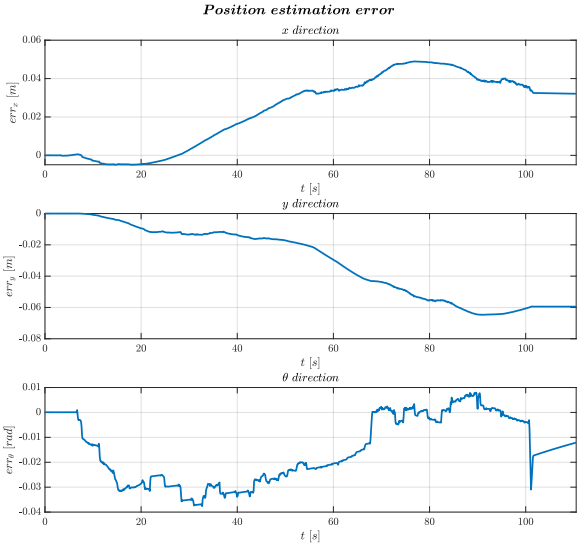

Figure 9: Error between the odometries.

includes secondary considerations such as an animated `.gif` illustrating the motion of the Turtlebot in the scanned space, with a black line highlighting the distance between itself and the nearest obstacle.

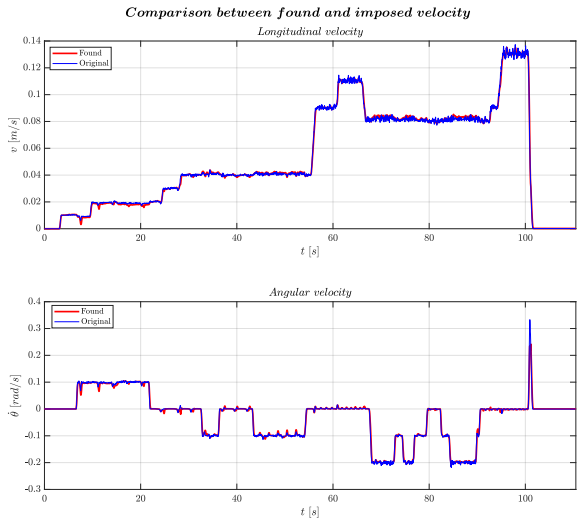

Figure 10: Comparison between the imposed (found) and the original command velocities.

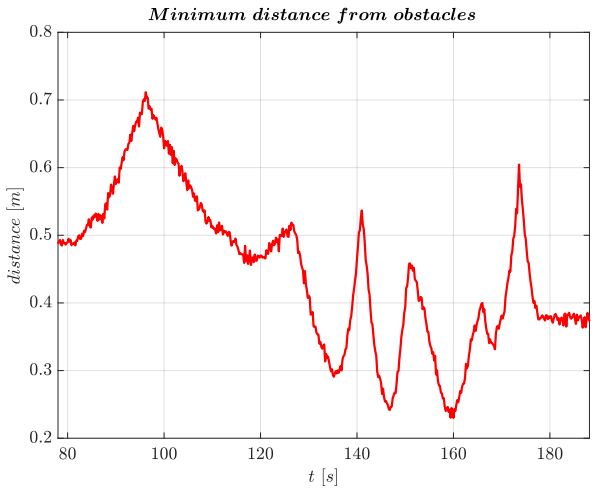

Figure 11: Minimum Turtlebot-obstacle distance along the trajectory.

B. Assignment 2

For the second assignment, we are provided with a list of 10 waypoints that the Turtlebot Burger must pass through, schematically shown in Figure 12. Implementing a feedback control policy is deemed the easiest and safest method to ensure the Turtlebot reaches these waypoints. Computing an open-loop control law would unnecessarily complicate matters and offer no guarantee of the Turtlebot arriving at the waypoints effectively. Therefore, a simple proportional feedback control action is implemented, where the longitudinal and angular command velocities are calculated proportionally to the difference between the Turtlebot's pose and the goal pose.

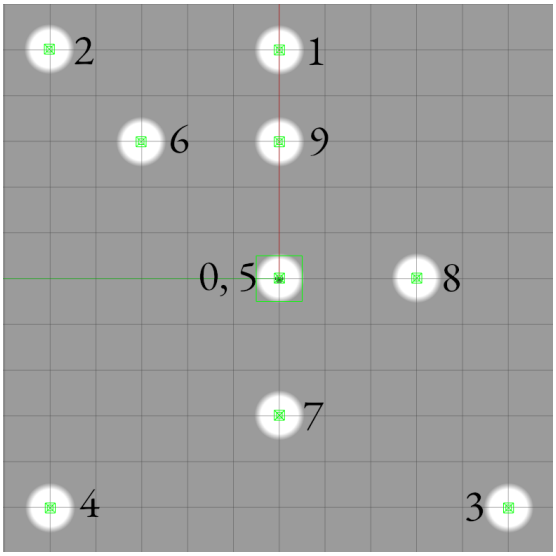

Figure 12: Waypoints to reach with numeration (0 is starting position, 10 is final goal).

The waypoints come with both position and orientation requirements. The strategy employed to meet these two requisites is as follows: starting from the $(i-1)^{\text{th}}$ waypoint, the magnitude of the distance vector $\boldsymbol{d}$ between the Turtlebot and the i^{th} waypoint is used to define a proportional law for the longitudinal command velocity v. Simultaneously, the angle of the distance vector is utilised to calculate a proportional control policy for the angular command velocity $\dot{\theta}$.

Mathematically speaking, if θ represents the yaw of the Turtlebot and $\boldsymbol{d}$ denotes the distance vector, the feedback policy is the following:

$$\begin{aligned} \boldsymbol{d} &= de^{j\phi} \\ v &= k_{\text{p,LON}} d \\ \dot{\theta} &= k_{\text{p,ANG}}(\phi - \theta) \end{aligned}$$

The feedback law described above is employed until the Turtlebot approaches the i^{th} waypoint sufficiently closely; at that point, it comes to a stop, namely $v = 0$, and the reference angle ϕ in the feedback calculation for $\dot{\theta}$ is substituted by the orientation of the waypoint. Once the difference between the Turtlebot's orientation and the waypoint's orientation falls below a certain tolerance, the Turtlebot resumes motion to reach the new waypoint. It does so by applying the feedback policy described earlier, considering the distance vector $\boldsymbol{d}$ to the next waypoint $(i+1)^{\text{th}}$.

One challenge encountered pertains to achieving correct Turtlebot movement around curves. The odometry topic outputs the orientation in quaternions, which are then converted to Euler angles using a built-in MATLAB function. The obtained orientation of the Turtlebot ranges from $-\pi$ to π, while the orientation of the waypoints spans from 0 to 2π. Simply converting the latter to match the Turtlebot's orientation is not sufficient for achieving smooth transitions in curves. In fact, let us consider the scenario where the Turtlebot is initially oriented at $\theta = 0.75\pi$ and it needs to rotate to reach a heading of $\theta_R = 1.25\pi$. Implementing the feedback policy to make the Turtlebot rotate $\dot{\theta} = k_{\text{p,ANG}}(\phi - \theta)$, makes a problem arise: the Turtlebot will continually spin back and forth without ever reaching the reference heading, as θ cannot exceed π or fall below $-\pi$ (see note[3]). Conversely, if θ_R is adjusted to fall within the interval $[-\pi, \pi]$, namely $\theta_R = 0.75\pi$, the Turtlebot's movement becomes unnatural. Instead of rotating left as intended, it spins around, turning to the right and rotating more than necessary. To address this issue, a conversion logic has been incorporated into the MATLAB function responsible for computing the

[3]This happens because when the yaw of the Turtlebot crosses π, it becomes negative as it simultaneously crosses $-\pi$. Consequently, when attempting to reach 1.25π, the Turtlebot will continue to revolve until π iis reached again, initiating the process anew, resulting in an endless loop.

feedback law within the Simulink model (shown in Figure 13) used for simulation. The longitudinal and angular command velocities, respectively denoted as `CVL` and `CVA`, are fed through saturation blocks to limit their maximum and minimum values. Figure 15 shows the evolution of the instantaneous longitudinal and angular command velocities.

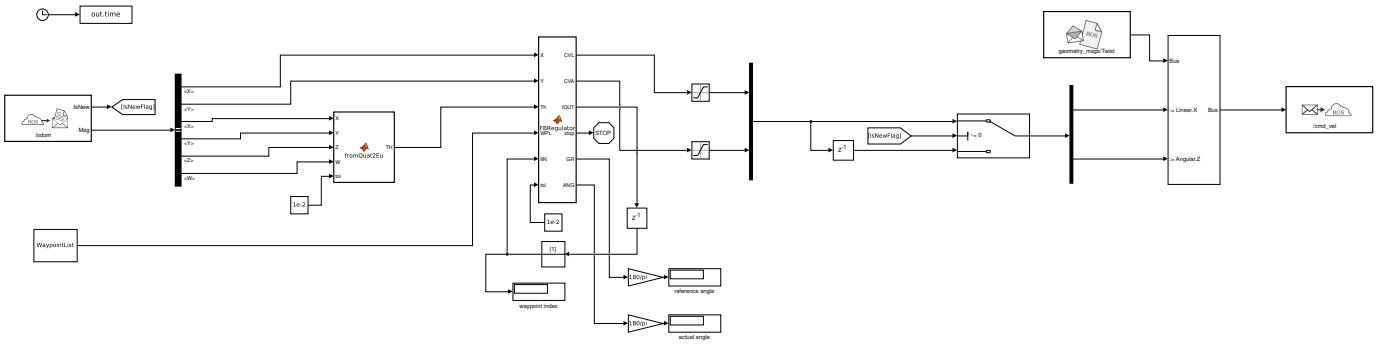

Figure 13: Simulink model used to publish `/cmd_vel`.

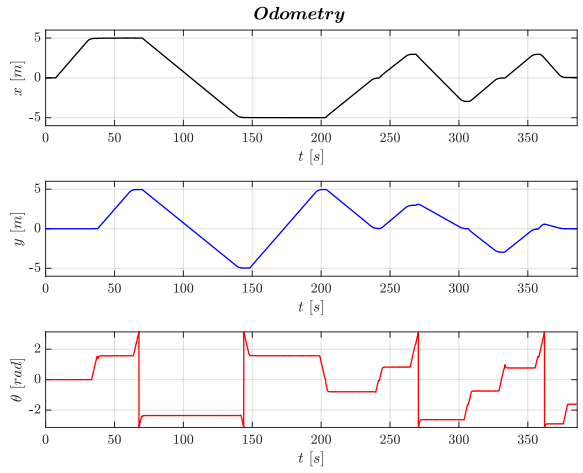

Figure 14: Recorded odometry (no wrapping on θ).

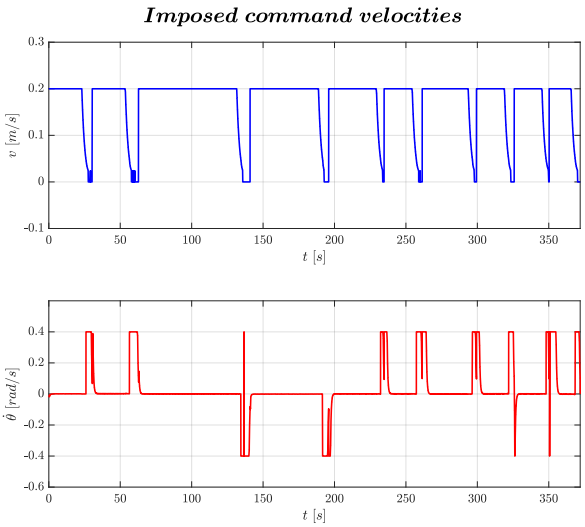

Figure 15: Imposed command velocities.

As evident from Figure 16, the waypoints have been reached with sufficient accuracy, and the Turtlebot's behaviour in the curves is acceptable. However, Figure 14 suggests that there may be instances where the Turtlebot slows down unnecessarily. This is primarily due to the selection of a general feedback control law, which does not account for specific transitions. To mitigate time losses in these instances, modifications can be made to the feedback control law calculation. For instance, a more flexible policy on the stopping criterion could be implemented, slowing down only when the curve is not tight. Additionally, a *forecasting* approach could be adopted, whereby the next waypoint is considered when approaching the current one, facilitating smoother transitions. While these aspects hold significant potential for improvement, they fall outside the scope of the present assignment and their implementation was not considered. Instead, as depicted in Figure 17, one attempted solution involved eliminating the stopping criterion in the proximity of the waypoint: as indicated by the colorbar, this attempt resulted in slightly faster movement, albeit at the expense of missing the orientation requirement of the waypoints.

For specific insights into the content of the MATLAB function responsible for calculating the feedback law and further details, readers are encouraged to refer to the GitHub repository[4].

As an additional bonus request, the assignment tasked creating a publisher to output the Cartesian and angular requirements of the list of waypoints in the ROS network using a `PoseWithCovariance` message. To address this requirement, we developed a Python code capable of handling the request within the ROS framework, reported in Listing 1.

[4]https://github.com/JacopoPorzio/AutonomousVehicles

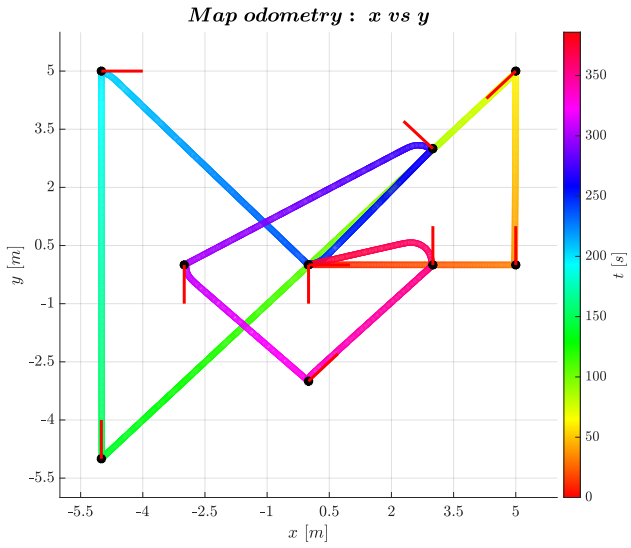

Figure 16: Cartesian plot of the trajectory. The waypoints are represented by the black bars, while the red bars indicate the waypoint's orientation.

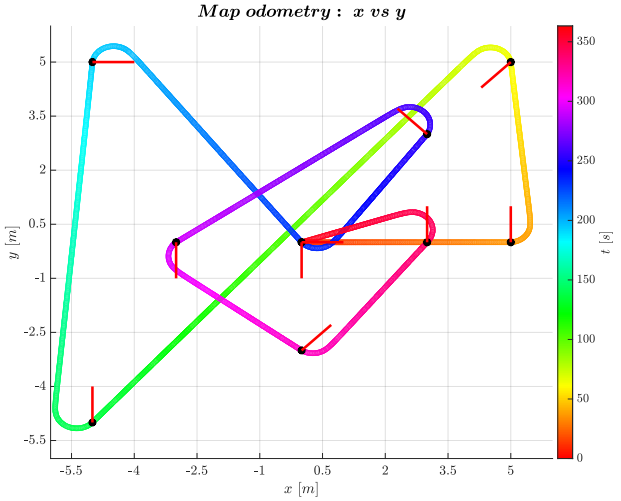

Figure 17: Cartesian plot of the trajectory. No stopping criterion at waypoints.

```python
import rospy
from geometry_msgs.msg import PoseWithCovariance
from std_msgs.msg import Float64
from math import pi

def wayp_pub():
        pub = rospy.Publisher('Waypoint_Publisher', PoseWithCovariance, queue_size=10)
        rospy.init_node('way_pub_node', anonymous=False)
        rate = rospy.Rate(10)
        while not rospy.is_shutdown():
                msg = PoseWithCovariance()
                msg.covariance = [ 5, 0, pi/2,  5, 5, pi*5/4,  -5, -5, pi/2,  -5, 5, 0,
                 ↪  0, 0, 0,  3, 3, pi*3/4,  -3, 0, pi*3/2,  0, -3, pi/4,  3, 0, pi/2,  0,
                 ↪  0, pi*3/2, 0, 0, 0, 0, 0, 0]
                pub.publish(msg)
                rate.sleep()
```

```
15
16 if __name__ == '__main__':
17         try:
18                 wayp_pub()
19         except rospy.ROSInterruptException:
20                 pass
```

Listing 1: Publisher code.

C. Assignment 3

The third assignment specifies the use of the Turtlebot Waffle Pi due to the presence of an onboard camera. The proposed task entails placing one red ball and one pink ball in the Gazebo empty world and leveraging the camera view to implement a feedback law. This law is intended to guide the Turtlebot towards the red ball and cause it to collide with it.

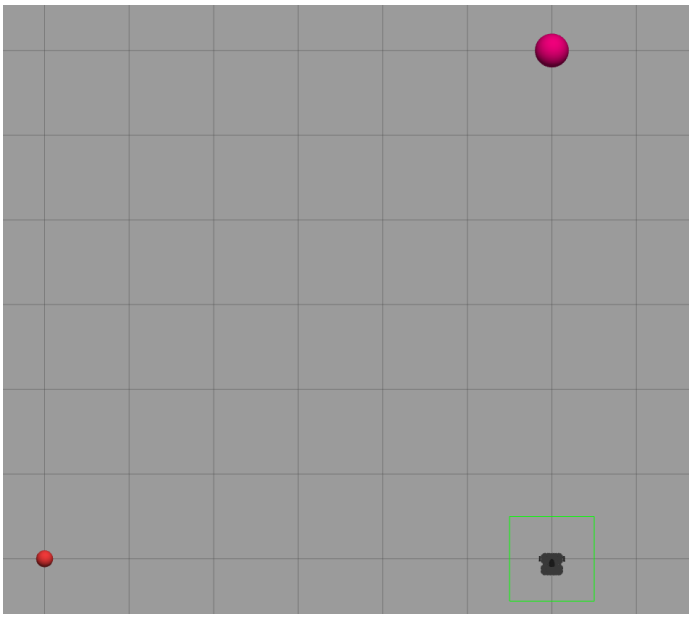

Figure 18: The Gazebo empty world with the Turtlebot and two coloured balls.

The implemented strategy is as follows: the Turtlebot initiates rotation to search for the ball. Once the red ball is detected, it adjusts its heading to center the ball in the image. After a short delay, it begins longitudinal movement. The implemented feedback policies for the longitudinal and angular command velocities, denoted as v and $\dot{\theta}$ respectively, utilise the area of the red ball, denoted as A, and the position of its centroid, denoted as CP, inside the image frame. Specifically, the longitudinal command velocity v is proportional to the area A of the red ball seen from the onboard camera. Instead, the angular command velocity $\dot{\theta}$ is adjusted to keep the horizontal position of the centroid CP_x of the red ball aligned with the horizontal center of the image, which has a width w_i equal to 640 pixel in our scenario:

$$v = k_{\text{p,LON}} A$$
$$\dot{\theta} = k_{\text{p,ANG}}(\frac{w_i}{2} - CP_x)$$

Since an unstable wobbling behaviour can be easily excited, a synergistic approach was adopted to keep it under control. For instance, the law for the longitudinal velocity could have been *inverted*, making it depend on the inverse of the area, to enable faster initial movement followed by deceleration. However, this approach lead to a wobbling phenomenon during testing. To mitigate the wobbling problem, a proper tuning of the feedback law was performed, adjusting gains appropriately and introducing a short initial delay for starting longitudinal motion when the red ball is first spotted. The initiation of longitudinal movement is delayed until the Turtlebot has centered the ball in its view, thus eliminating the wobbling problem. This delay was implemented by introducing a variable that is outputted when the Turtlebot first detects the ball, with the controller for longitudinal velocity only acting upon it after a short delay. Otherwise, initiating movement immediately upon sighting the ball would cause it to shift off-center in the image. Consequently, the error associated with angular velocity $\dot{\theta}$ would increase, prompting the Turtlebot to push with angular velocity to recenter the ball. If the resulting actions of both longitudinal v and angular velocity $\dot{\theta}$ are too strong, the Turtlebot may begin to wobble and there is a possibility for an unstable motion to occur.

In the complete Simulink model (whose complete picture is not shown for the sake of brevity) responsible for implementing our solution, the first block to encounter is responsible for extracting the image, represented by the variable `Video`, from the ROS network, as depicted in Figure 19. The variable `IsNewFlag` is utilised to prevent inputting a new signal to the Turtlebot when no new image is received.

Then, as shown in Figure 20, the variable `Video` flows directly into the block responsible for converting the RGB image into a binary representation `BW`, after undergoing various processing procedures to extract red objects from the image. For this purpose, the HSV representation proves to be a valuable aid. According to color theory, red falls within the hue interval $[0°, 60°]$, which, after normalisation, becomes $[1, \frac{1}{6}]$. In the Gazebo world, the ball we are searching for is red, while the floor

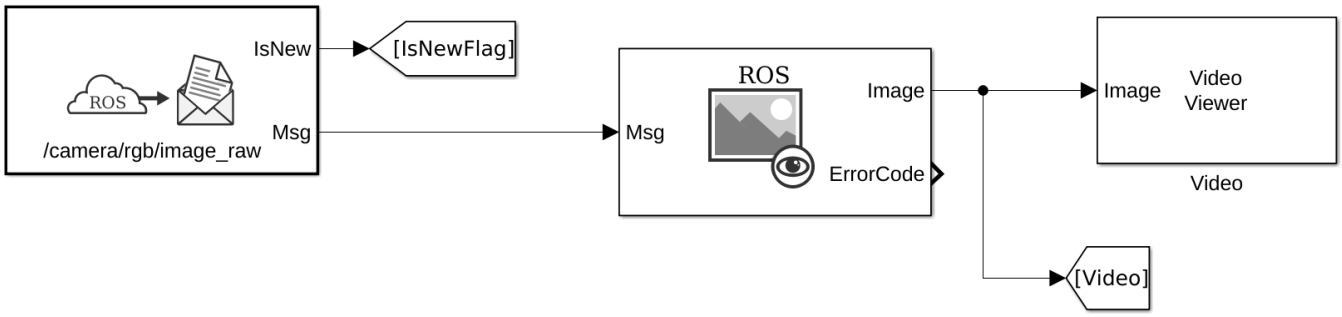

Figure 19: Image extraction from ROS network.

and sky are two different shades of gray. However, the conventions used for HSV color representation assign a hue of 0 to gray, rather than to red only. If we were searching for any color other than red, we could rely solely on hue. However, in this case, we need to leverage saturation because the saturation for red is significantly higher than that for gray. To address this, a matrix multiplication is performed to obtain a *saturated hue image*. From this image, we extract only the high saturation values in a Boolean fashion, which identifies the red color, resulting in a binary image represented by the variable `BW`.

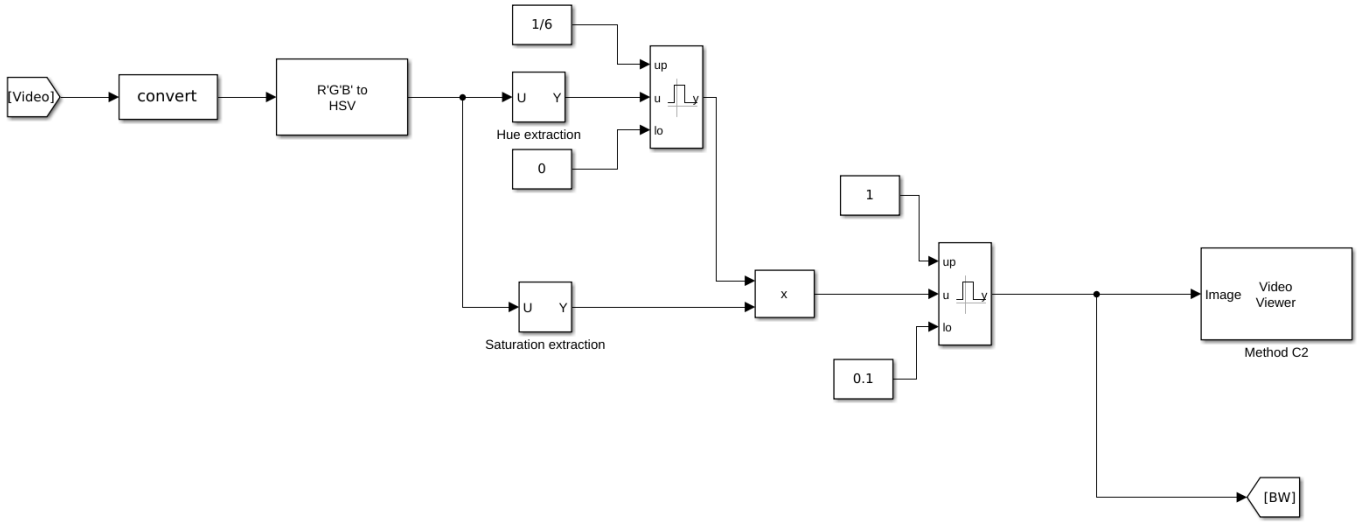

Figure 20: Red extraction and image binarisation.

The core of the proposed solution lies in the part of the Simulink model reported in Figure 21, where the publication of the command velocities and of the position of the centroid in the image frame, asked by the assignment, to the ROS network happen. The longitudinal and angular command velocities, respectively denoted as `CVL` and `CVA`, are fed through saturation blocks to limit their maximum and minimum values.

Figure 22 shows in detail the content of the gray subsystem present in Figure 21, responsible for the feedback calculation of the command velocities.

Instead, Figures 23 and 24 offer a perspective on the results of the image processing.

As Figures 25, 27 and 28 show, the wobbling movement is completely eliminated. From Figure 28, the delay of the longitudinal command velocity with respect to the angular one can be appreciated. Figure 26 fulfills the assignment's request to plot the trajectory of the centroid position in the image frame.

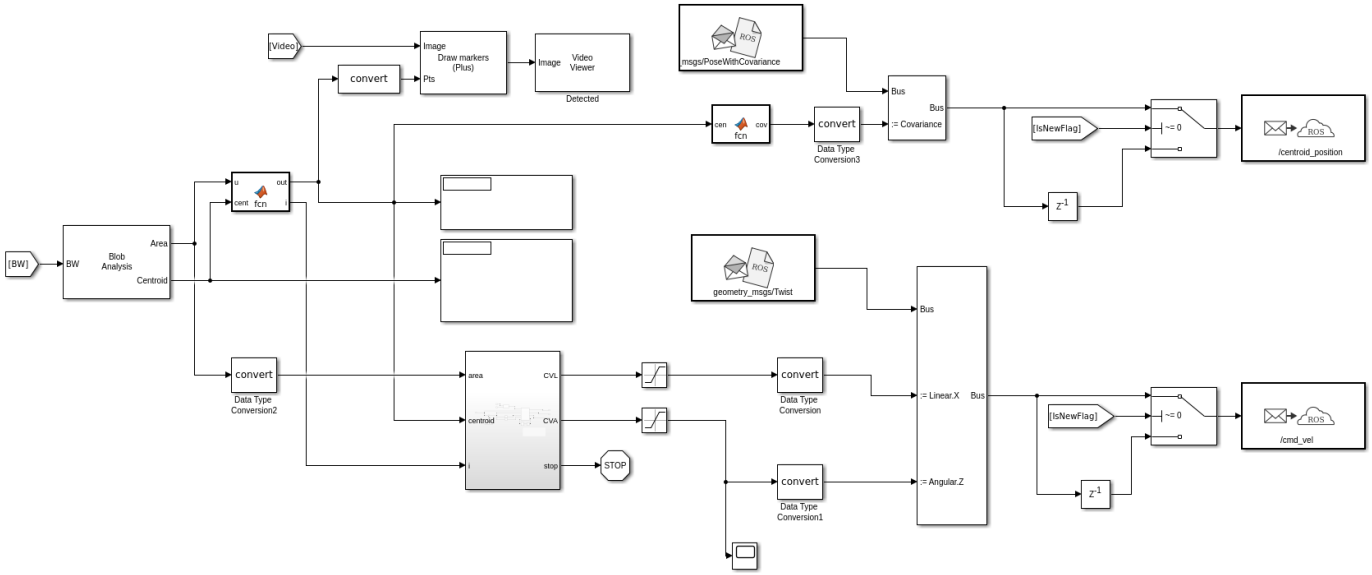

Figure 21: Publication block.

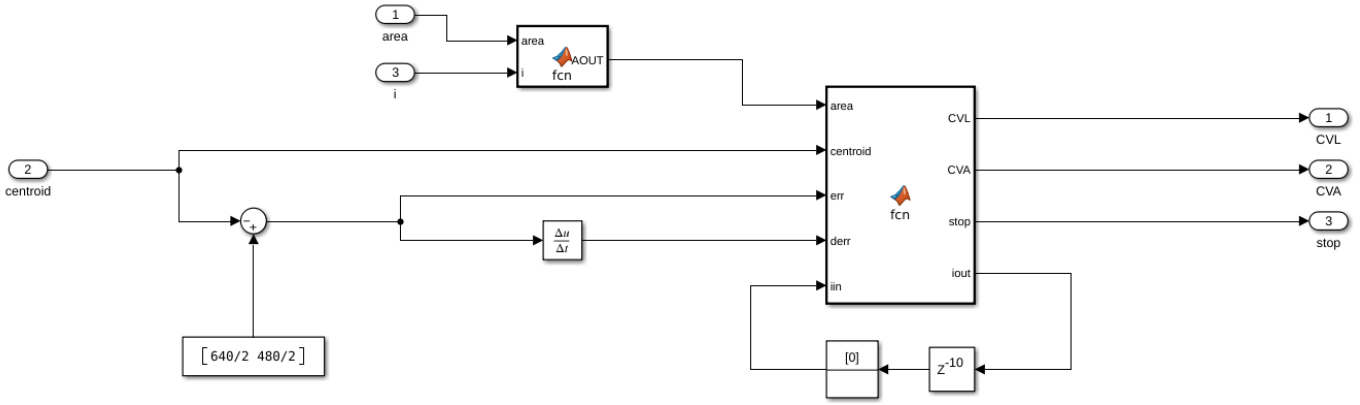

Figure 22: Feedback law's computation subsystem.

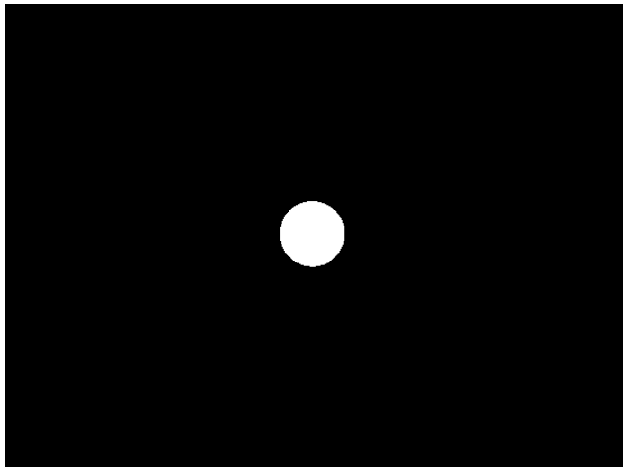

Figure 23: The recognised red ball represented as a BLOB.

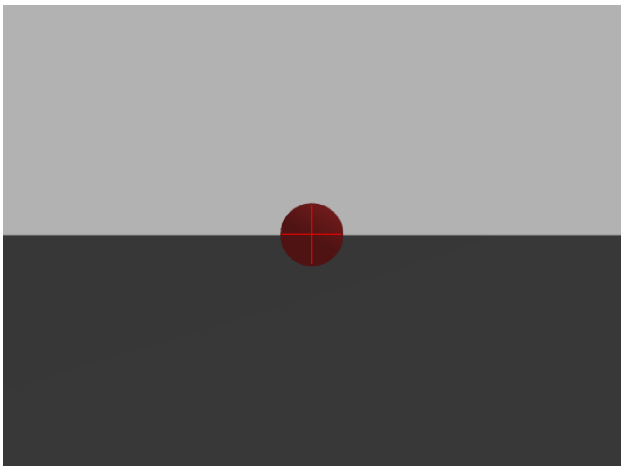

Figure 24: The centroid of the ball highlighted with a marker.

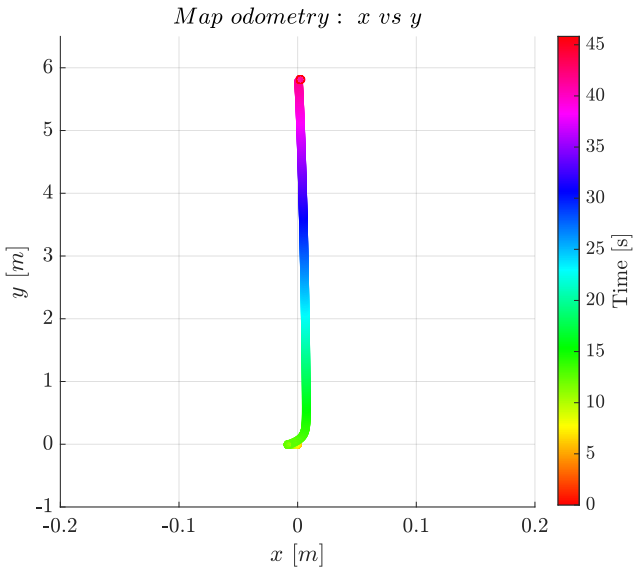

Figure 25: Cartesian trajectory of the Turtlebot.

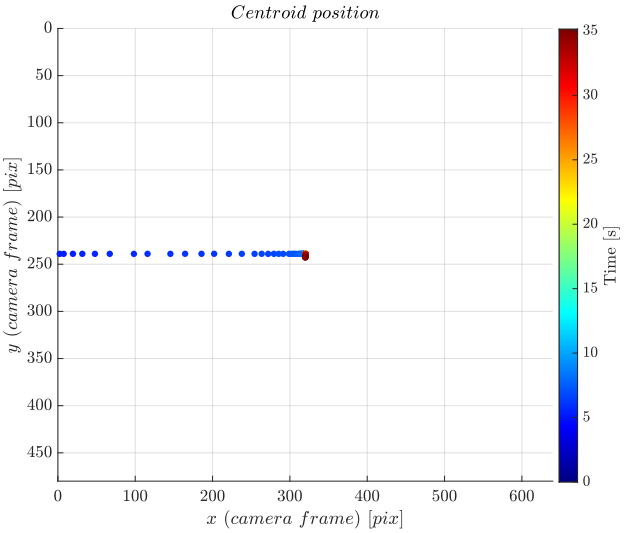

Figure 26: Trajectory of the centroid within the image frame.

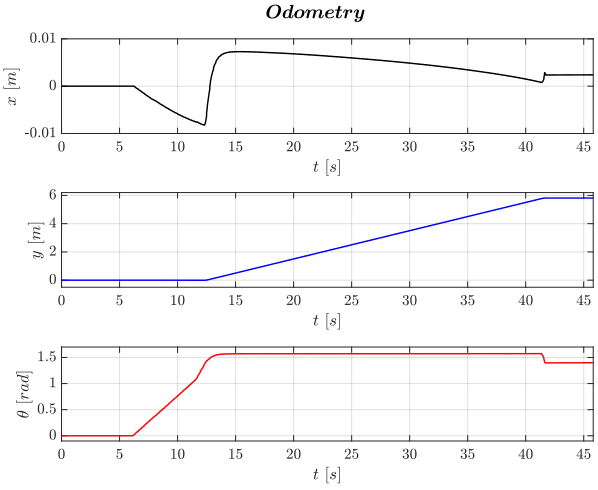

Figure 27: Trajectory odometry.

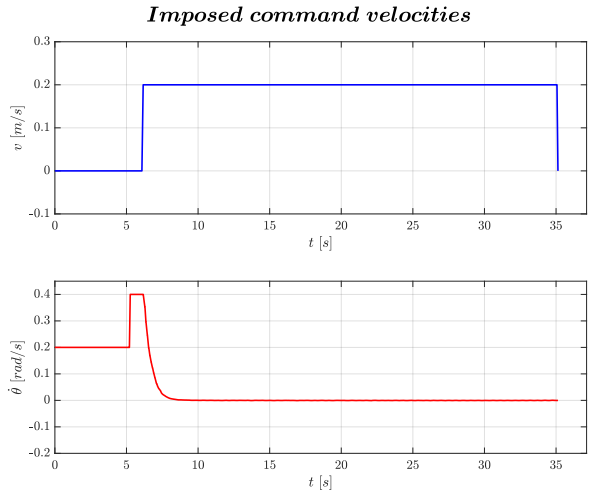

Figure 28: Evolution of the imposed command velocities.

D. Assignment 4

The fourth assignment tasks us with modifying a provided MATLAB code implementing a simplified Dijkstra algorithm to find the best route from a starting position to a final goal in a set of four provided `.png` images, which are imported and binarised to matrices within the code, with 0 representing free space, 1 representing occupied space. The provided code currently only considers vertical or horizontal movements but does not account for diagonal movements, which we are instructed to incorporate into our code. Additionally, we are tasked with further modifying the algorithm to transform it into an A* method.

Although acknowledged, certain inefficiencies of the code were not addressed within the scope of the current assignment. One such inefficiency pertains to the dependencies matrix G, which represents connectivity between nodes. The entry G_{ij} equals 1 if a path can go from the i^{th} node to the j^{th} node and 0 otherwise. For a square image of size $N \times N$, the graph contains N^2 nodes, resulting in a $N^2 \times N^2$ matrix G. This presents memory usage challenges, especially with images as small as 300 pixels per side. We propose a solution to this limitation in the context of the Final Project 2 (see Section III-A).

For the sake of brevity, we only present here the specific modifications made to the code, directing interested readers to the complete code on our GitHub repository[5]. In Listing 2, the code responsible for assembling the dependencies matrix G allowing diagonal movements is reported. New exploration conditions had to be implemented to determine if a node j is reachable from the node i, denoted by G_{ij} being non-zero. Additionally, the entry G_{ij} for a diagonal connection is the $\sqrt{(2)}$, distinguishing it from horizontal and vertical displacements, highlighted by 1.

```
G = - 1*ones(size(BW, 1)*size(BW, 2)); % tree dependencies
% create edge matrix
for ii = 1:size(BW, 1)
    for jj = 1:size(BW, 2)
        if BW(ii, jj) == 1
            %%% itself
            G( (ii - 1)*size(BW, 1) + jj,(ii - 1)*size(BW, 1) + jj)=0;
            %%% exploring verticals and horizontals (what we are provided)
            %       N
            %      \|/
            % W ---+--- E
            %      /|\
            %       S

            SCon = ii + 1 > 0 && ii + 1 <= size(BW, 1) && BW(ii + 1, jj) == 1;
            NCon = ii - 1 > 0 && ii - 1 <= size(BW, 1) && BW(ii - 1, jj) == 1;
            ECon = jj + 1 > 0 && jj + 1 <= size(BW, 2) && BW(ii, jj + 1) == 1;
            WCon = jj - 1 > 0 && jj - 1 <= size(BW, 2) && BW(ii, jj - 1) == 1;

            % South
            if SCon
                G((ii - 1)*size(BW, 1) + jj, (ii + 1 - 1)*size(BW, 1) + jj) = 1;
            end

            % North
            if NCon
                G((ii - 1)*size(BW, 1) + jj, (ii - 1 - 1)*size(BW, 1) + jj) = 1;
            end

            % East
            if ECon
                G((ii - 1)*size(BW, 1) + jj, (ii - 1)*size(BW, 1) + jj + 1) = 1;
            end

            % West
            if WCon
                G((ii - 1)*size(BW, 1) + jj, (ii - 1)*size(BW, 1) + jj - 1) = 1;
            end

            %%% exploring diagonals (novelty)
            %
            % NW \|/ NE
            %   ---+---
            % SW /|\ SE
```

[5]https://github.com/JacopoPorzio/AutonomousVehicles

```
            %
            SWCon = (ii + 1 > 0 && jj - 1 > 0) && (ii + 1 <= size(BW, 1) && jj - 1 <=
            ↪  size(BW, 2)) && (BW(ii + 1, jj - 1) == 1);
            NWCon = (ii - 1 > 0 && jj - 1 > 0) && (ii - 1 <= size(BW, 1) && jj - 1 <=
            ↪  size(BW, 2)) && (BW(ii - 1, jj - 1) == 1);
            SECon = (ii + 1 > 0 && jj + 1 > 0) && (ii + 1 <= size(BW, 1) && jj + 1 <=
            ↪  size(BW, 2)) && (BW(ii + 1, jj + 1) == 1);
            NECon = (ii - 1 > 0 && jj + 1 > 0) && (ii - 1 <= size(BW, 1) && jj + 1 <=
            ↪  size(BW, 2)) && (BW(ii - 1, jj + 1) == 1);


            % NE
            if NECon
                 G((ii - 1)*size(BW, 1) + jj, (ii - 1 - 1)*size(BW, 1) + jj + 1) =
                 ↪  sqrt(2);
            end

            % NW
            if NWCon
                 G((ii - 1)*size(BW, 1) + jj, (ii - 1 - 1)*size(BW, 1) + jj - 1) =
                 ↪  sqrt(2);
            end

            % SE
            if SECon
                 G((ii - 1)*size(BW, 1) + jj, (ii + 1 - 1)*size(BW, 1) + jj + 1) =
                 ↪  sqrt(2);
            end

            % SW
            if SWCon
                G((ii - 1)*size(BW, 1) + jj, (ii + 1 - 1)*size(BW, 1) + jj - 1) = sqrt(2);
            end

        end
    end

end
```

Listing 2: Dependencies matrix G allowing diagonal movements.

This modification reported in Listing 2 serves both to enable the original Dijkstra algorithm to consider diagonal movements and to introduce the A* algorithm. However, for completing the implementation of A* algorithm, the code responsible for calculating the search costs must also account for the so-called *cost-to-go*, which an underestimate of the cost to reach the goal from the considered node, which helps discourage exploration in regions of the graph far from the goal. In our case, we calculate the *cost-to-go* as the Euclidean distance between the considered node and the goal. In Listing 3, we present a snippet of code illustrating this modification, emphasising the variations in the calculation of search costs for the three different algorithms: the original Dijkstra, the Dijkstra algorithm modified to allow diagonal movements, and the A* algorithm.

```
switch AssignmentCase
            case 'DijkstraOriginal'
                if dist(con_nodes(i_con)) > dist(cur_node) + 1
                    dist(con_nodes(i_con)) = dist(cur_node) + 1;
                    prec(con_nodes(i_con)) = cur_node;
                end

            case 'DijkstraDiagonal'
                if ( dist(con_nodes(i_con)) > dist(cur_node) + 1 ) && G(cur_node,
                ↪  con_nodes(i_con)) == 1
                    dist(con_nodes(i_con)) = dist(cur_node) + 1;
                    prec(con_nodes(i_con)) = cur_node;
                elseif ( dist(con_nodes(i_con)) > dist(cur_node) + sqrt(2)) &&
                ↪  G(cur_node, con_nodes(i_con)) == sqrt(2)
                    dist(con_nodes(i_con)) = dist(cur_node) + sqrt(2);
```

```
                        prec(con_nodes(i_con)) = cur_node;
                    end

            case 'A*'
                    ny = fix(con_nodes(i_con)/size(BW, 1)) +  1*(mod(con_nodes(i_con),
                    ↪  size(BW, 1)) ~= 0);
                    nx = con_nodes(i_con) - (ny - 1)*size(BW, 1);
                    CostToGo = sqrt((nx - goal_pos(1))^2 + (ny - goal_pos(2))^2);
                    if ( dist(con_nodes(i_con)) > dist(cur_node) + 1 + CostToGo) &&
                    ↪  G(cur_node, con_nodes(i_con)) == 1
                        dist(con_nodes(i_con)) = dist(cur_node) + 1 + CostToGo;
                        prec(con_nodes(i_con)) = cur_node;
                    elseif ( dist(con_nodes(i_con)) > dist(cur_node) + sqrt(2) + CostToGo)
                    ↪  && G(cur_node, con_nodes(i_con)) == sqrt(2)
                        dist(con_nodes(i_con)) = dist(cur_node) + sqrt(2) + CostToGo;
                        prec(con_nodes(i_con)) = cur_node;
                    end
end
```

Listing 3: Code comparing the cost calculation for the originally provided Dijkstra algorithm, Dijkstra accounting for diagonal movements, and A*.

With these modifications in place, we can execute the algorithms on the maps and compare their performance. For convenience, we refer to the originally provided Dijkstra algorithm as "Base case Dijkstra" and the Dijkstra algorithm allowing diagonal movements as "Complete Dijkstra". In the following images, the light blue dots represent accessible positions on the map, while black dots are occupied by obstacles. The nodes that are analysed by the search algorithm are indicated by superimposed cross marks.

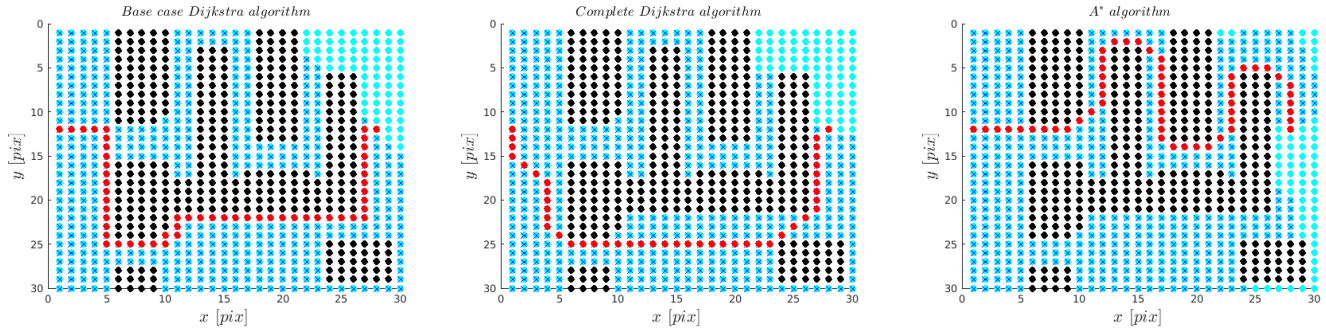

Figure 29: Algorithms comparison on the first map.

Algorithm	Number of analysed nodes	Path length in number of nodes
Base case Dijkstra	494	54
Complete Dijkstra	493	44
A*	518	54

Table I: Algorithms comparison on the first map.

Figure 29 and Table I show that A* performs worse than the other two algorithms, as highlighted by a larger number of analysed nodes and longer path length. However, this outcome is attributed to the specific characteristics of the map under consideration. In fact, along its path, A* selected points that are closer to the final goal compared to the other two algorithms: this strategy is not ideal with this specific map. When examining the second map (see Figure 30 and Table II), the true effectiveness of the A* algorithm becomes apparent.

Algorithm	Number of analysed nodes	Path length in number of nodes
Base case Dijkstra	703	53
Complete Dijkstra	707	42
A*	546	45

Table II: Algorithms comparison on the second map.

As evident from Figure 30 and Table II, with this type of map, the A* algorithm demonstrates superior performance. It is worth noting that while A* yields a longer path length, it manages to significantly reduce the number of required explorations,

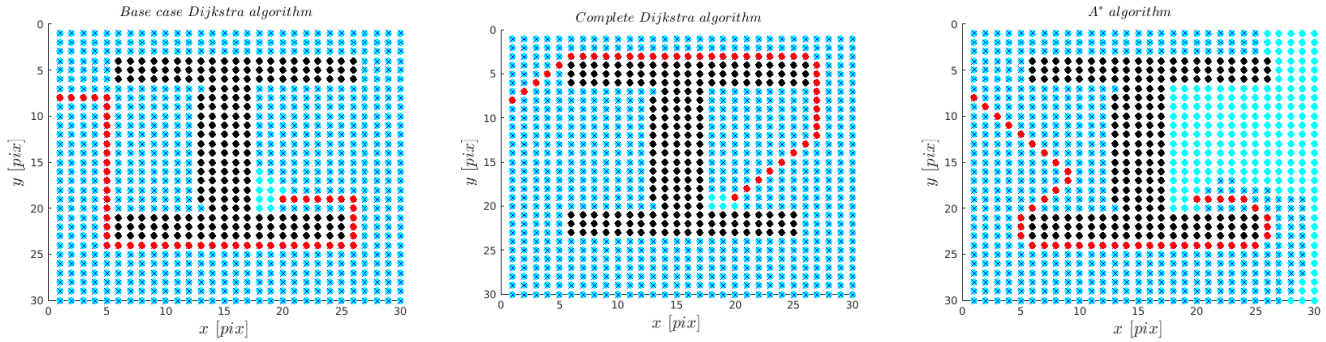

Figure 30: Algorithms comparison on the second map.

saving 23% exploration compared to Complete Dijkstra. However, it is essential to recognise that the path lengths between the Complete Dijkstra and A* algorithms are not entirely comparable. Once again, this disparity arises because A* considers the distance between the goal and the node when choosing the path to follow, leading to the inclusion of certain displacements that the Dijkstra algorithm would disregard.

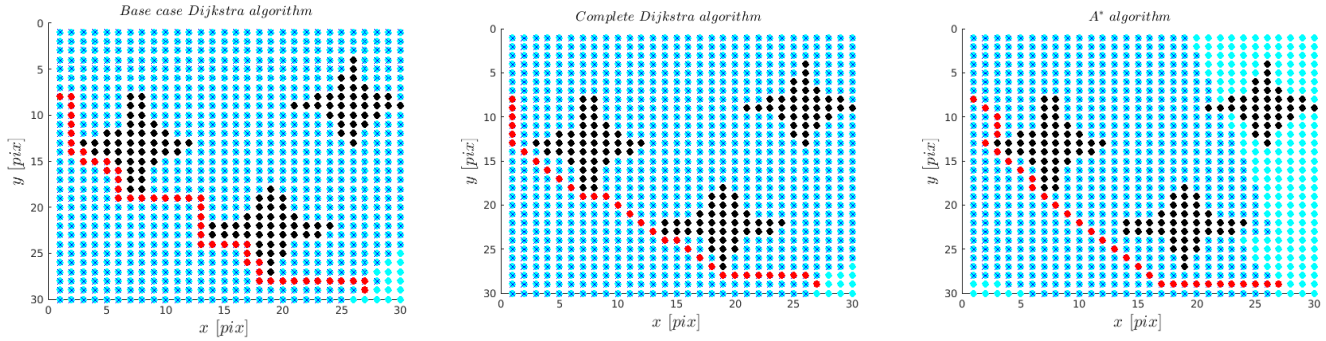

Figure 31: Algorithms comparison on the third map.

Algorithm	Number of analysed nodes	Path length in number of nodes
Base case Dijkstra	771	48
Complete Dijkstra	776	32
A*	646	32

Table III: Algorithms comparison on the third map.

Figure 31 and Table III offer another example of the computational advantage of A*: while achieving the same path length, the Complete Dijkstra algorithm needs 20% more exploration compared to A*.

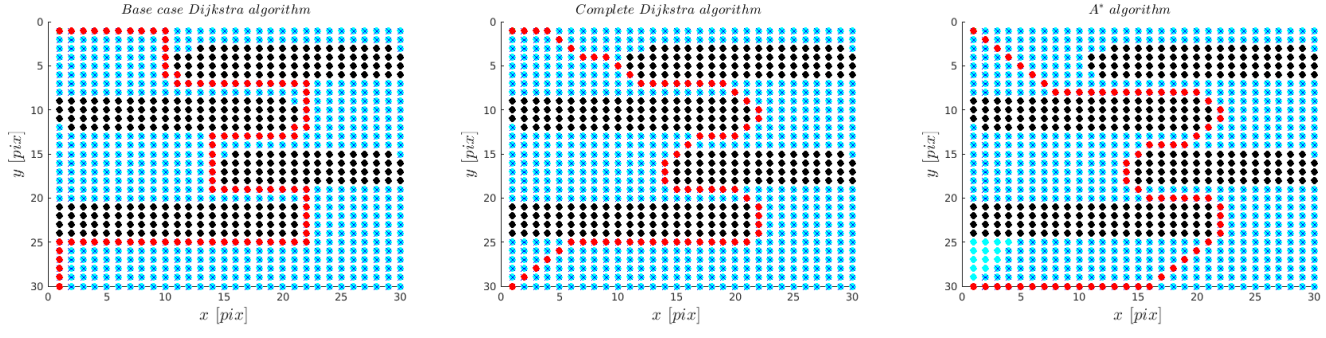

Figure 32: Algorithms comparison on the fourth map.

Finally, from Figure 32 and Table IV we can conclude that the performances of the two algorithms are strongly map-dependent. However, even if some maps make the performances of the algorithms almost indistinguishable, A* one proves to

Algorithm	Number of analysed nodes	Path length in number of nodes
Base case Dijkstra	597	88
Complete Dijkstra	597	65
A*	592	65

Table IV: Algorithms comparison on the fourth map.

be the best trade-off between exploration and path length, thanks to its capability of discarding paths that are too far from the goal.

E. Assignment 5

The assignment requires implementing a Finite State Machine model of a parking gate. The gate should raise when a car is waiting in front of it, remain raised while the car is transiting and lower once the car has passed. The movement of the car has been modeled using a Finite State Machine, modified from a provided one.

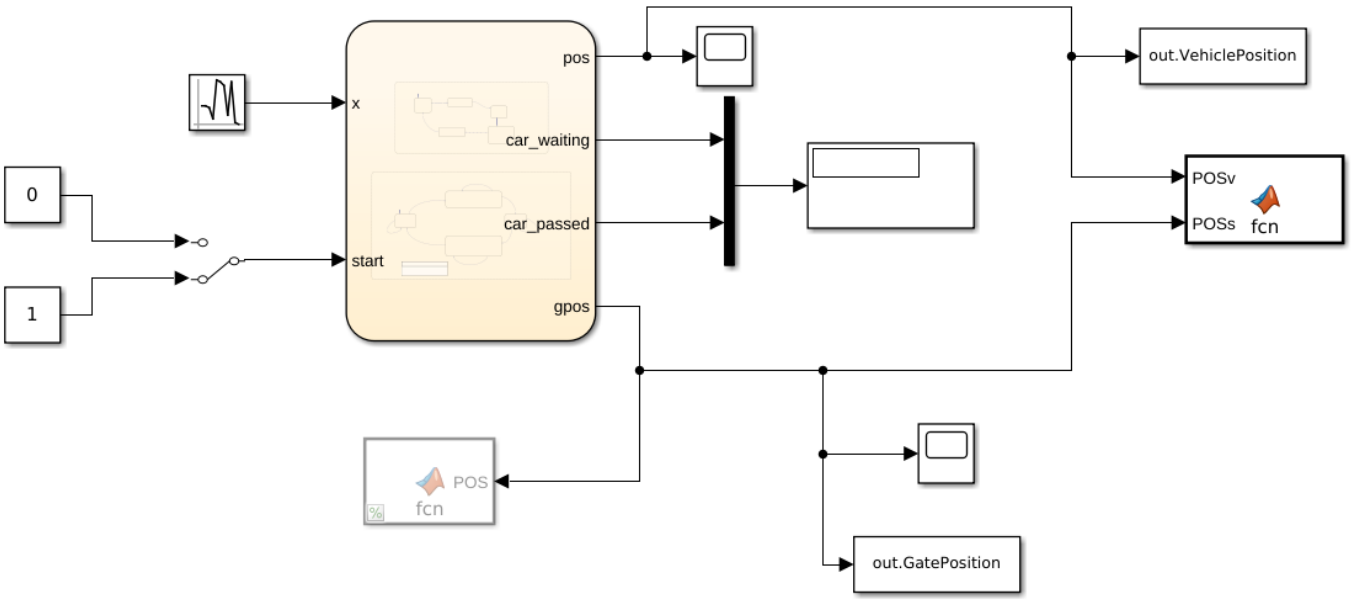

Figure 33: Simulink model of the whole system.

Figure 33 displays the complete model of the system. The simulation takes place within the yellow chart, which houses two Finite State Machines running in parallel: the one shown in Figure 34 simulates the movement of the car, while the other reported in Figure 35 is responsible for simulating of the motion of the parking gate.

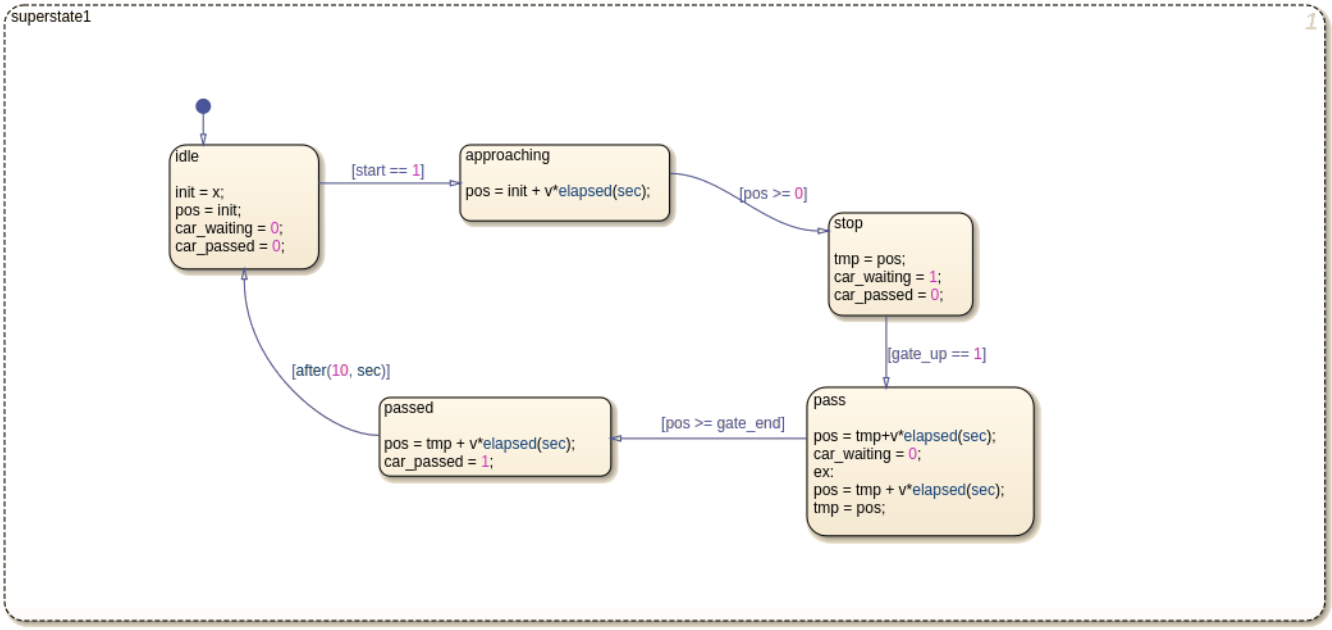

Figure 34: Finite State Machine simulating the car movement.

Taking as reference Figure 33, the initial position of the car `pos` is determined by a random number input to the system. Until the switch is manually triggered, indicated by the `start` variable being set to 1, the car remains stationary. Initially,

the car's position `pos` is randomly set as negative, progressing towards 0 in front of the parking gate. When it reaches 0, the variable `car_waiting` is set to 1, indicating it is in the waiting state. At this point, the gate begins to raise. Once the gate has reached the upward position (indicated by `gate_up` being set to 1 by the gate's Finite State Machine), the car can proceed at a constant velocity. After the car has traveled a fixed distance from the gate, it enters the `passed` state, and the gate can lower. Ten seconds after the car has passed, another car is spawned.

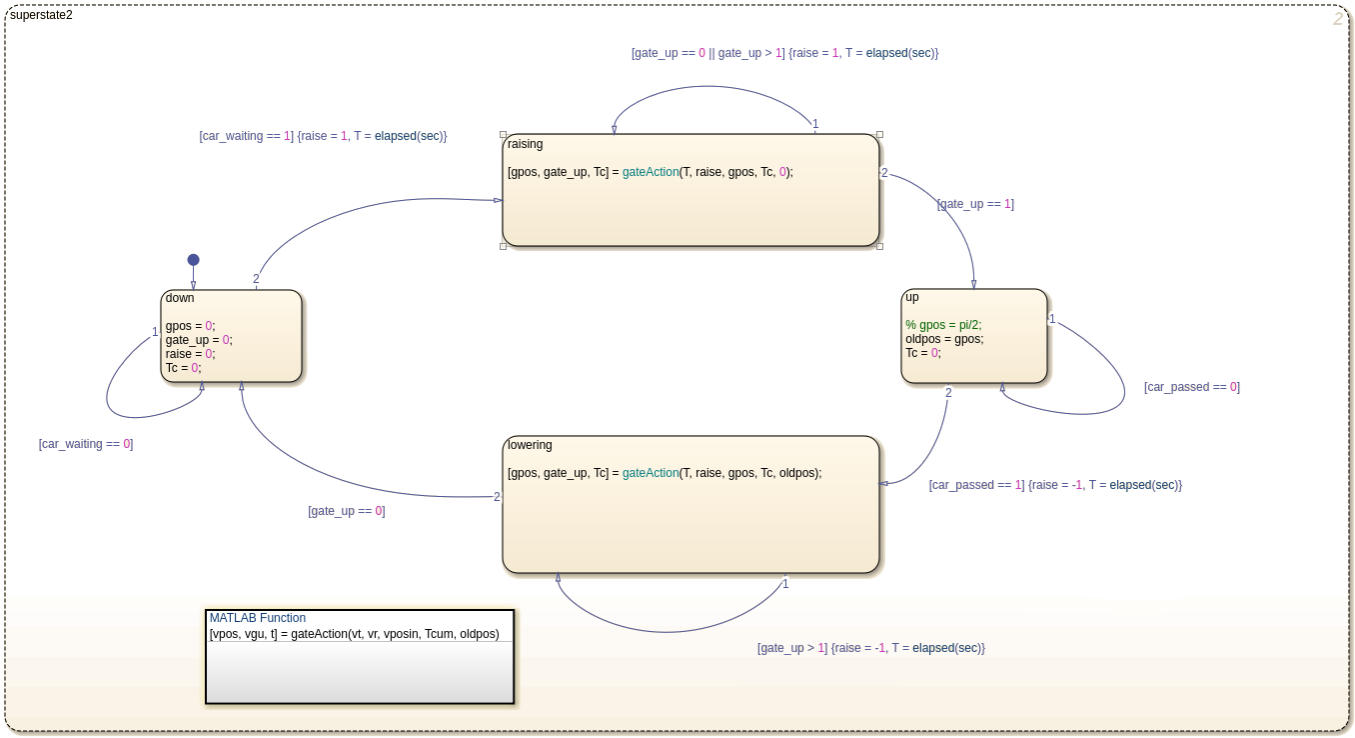

Figure 35: Finite State Machine simulating the gate displacement.

As anticipated, Figure 35 shows the simulation strategy behind the parking gate. When no car is waiting, the gate remains in the lowered position. However, as soon as a car approaches the gate, it enters the `raising` state. During this state, the MATLAB function `gateAction` (see Listing 4) handles the movement of the gate while tracking the elapsed time to ensure the gate follows a specific velocity profile. Once the gate reaches a position of 90°, indicating that it is fully raised, the variable `gate_up` is set to 1, allowing the car to pass. The gate maintains this position until the car has passed to the other side. Subsequently, the gate enters the `lowering` state, where the MATLAB function behaves similarly to the raising case, bringing the gate back to its resting position. Once the gate reaches the resting position, the cycle can restart from the beginning.

```
function [gpos, gate_up, Tc] = gateAction(T, raise, gposin, Tc, oldpos)

    wmax = 4*pi/180;  % rad/s, maximum angular velocity of gate
    dwmax = 1*pi/180; % rad/s, maximum angular acceleration of gate
    DC = 0;           % rad,   down condition
    UC = pi/2;        % rad,   up condition

    Tc = Tc + T;

    if raise == 1
        gpos = 1/2*dwmax*Tc^2;
    elseif raise == - 1
        gpos = oldpos - 1/2*dwmax*Tc^2;
    else
        gpos = gposin;
    end

    if abs(gpos - DC) < 1e-2
        gate_up = 0;
```

```
    elseif abs(vpos - UC) < 1e-2
        gate_up = 1;
    else
        gate_up = 2;
    end

end
```

Listing 4: MATLAB function regulating the movement of the gate.

The MATLAB function `gateAction` is responsible for simulating the motion of the gate, and its behaviour is straightforward. It implements a simple maximum acceleration profile, with positive acceleration for raising the gate and negative acceleration for lowering it. An offline check ensures that the maximum velocity reached is within the specified limit. When the gate is in the vertical position, the variable `gate_up` is set to 1, indicating that the car can pass. A tolerance is used to determine the position of the gate relative to the resting and upward references, as strict equality checks proved to be unreliable.

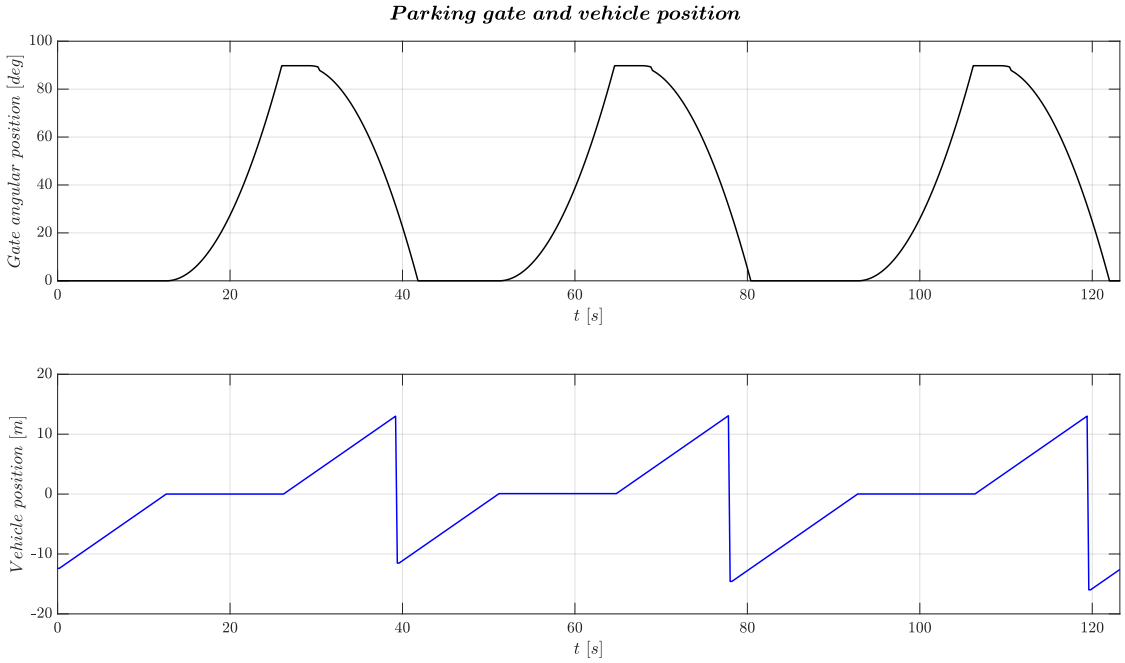

Figure 36: Comparison between the positions of the gate and the car over time.

Figure 36 provides a comparison between the positions of the gate and the car over time. The plot of the angular position of the gate exhibits a typical maximum acceleration/maximum deceleration profile, indicating that the MATLAB function is functioning correctly. The discontinuity in the car's position is a result of the spawning of a new car, as the previous one has moved a sufficient distance away from the gate. For further visualisations, the interested reader is encouraged to refer to the car and gate animations available on the GitHub repository[6].

[6]https://github.com/JacopoPorzio/AutonomousVehicles

III. Final Projects

A. Final Project 2

The Final Project 2 encompasses three objectives:

1) simulate the Turtlebot within the `Gazebo House` scenario and utilise the `gmapping` tool to generate a map of the environment;
2) modify the A* algorithm from Assignment 4 (see Section II-D) to enable analysis of maps larger than 300x300 pixels, thereby overcoming limitations associated with the dependencies matrix G;
3) apply feedback control to make the Turtlebot track a trajectory calculated through the modified A* algorithm on the map generated with gmapping, while localising the Turtlebot on the same map.

The Turtlebot utilised for this task is the Waffle Pi. Addressing the first objective is relatively straightforward following a simple strategy, albeit it is time-consuming to map such a sizable environment manually with the Turtlebot. Utilising standard SLAM routines provided by `gmapping` without customization yielded a slightly imperfect but acceptable result, as depicted in Figure 37. Before feeding the map into the algorithm, the corresponding image needs to be binarised into a matrix once again: unknown space (gray) and free space (white) will be set to 0, while occupied space (black) will be represented by 1.

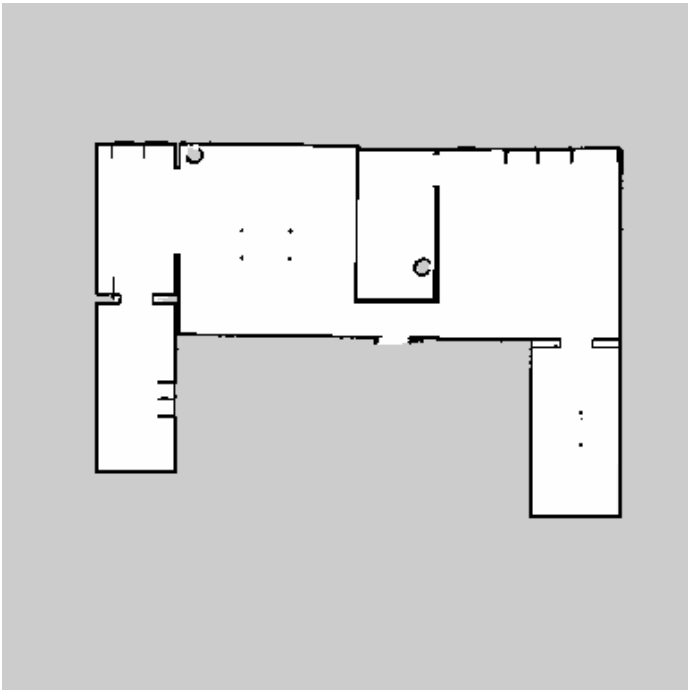

Figure 37: Map obtained through the SLAM method offered by `gmapping`.

Regarding the second point, the modification of the A* algorithm, as anticipated in Assignment 4 (see Section II-D), the provided version needs to be adapted due to its highly inefficient method of calculating dependencies, i.e., the nodes that can be reached from a given one. The dependencies matrix G is interpreted such that the entry G_{ij} is equal to 1 if a path can go from the i^{th} node to the j^{th} node and equal to 0 otherwise. With an image size of N×N, containing N^2 nodes, the matrix G has a size of N^2×N^2, leading to excessive memory usage even for relatively small images. To address this problem, the G matrix is eliminated because storing information about the nodes reachable from a given node is unnecessary when focusing the calculation of the path on a specific node very distant from it. Instead, dependency calculation is performed dynamically at each step of solving, considering only the dependencies of the current node; when a new node is taken into consideration, the previous dependency information is substituted by the current one. This means that the algorithm dynamically updates the dependency information as it progresses, rather than storing the entire dependency matrix G. The same algorithm used to assemble the G matrix in Assignment 4 is modified to accommodate these novelties, implemented in the MATLAB function `conNodescalc` (see Listing 5). This function takes a free node within the image as input and examines the vertical, horizontal, and diagonal directions around the node to determine its dependencies. This approach streamlines the algorithm and significantly reduces memory usage, at the cost of slightly slowing down the process. However, we are not interested in real-time trajectory calculation, and we are satisfied with the result.

```
function [conNodes, costNodes] = conNodescalc(BW, q)
    ny = fix(q/size(BW, 1)) +  1*(mod(q, size(BW, 1)) ~= 0);
    nx = q - (ny - 1)*size(BW, 1);
    % find all the connected nodes (i.e. also the inaccessible ones)
    conNodesMax = -1*ones(1, 8);
    costNodesMax = -1*ones(1, 8);

    if BW(ny, nx) == 1
        % Horizontal and vertical directions
        SCon = ny + 1 > 0 && ny + 1 <= size(BW, 1) && BW(ny + 1, nx) == 1;
        NCon = ny - 1 > 0 && ny - 1 <= size(BW, 1) && BW(ny - 1, nx) == 1;
        ECon = nx + 1 > 0 && nx + 1 <= size(BW, 2) && BW(ny, nx + 1) == 1;
        WCon = nx - 1 > 0 && nx - 1 <= size(BW, 2) && BW(ny, nx - 1) == 1;

        % North
        if SCon
            conNodesMax(1, 7) = (ny + 1 - 1)*size(BW, 1) + nx;
            costNodesMax(1, 7) = 1;
        end

        % South
        if NCon
            conNodesMax(1, 2) = (ny - 1 - 1)*size(BW, 1) + nx;
            costNodesMax(1, 2) = 1;
        end

        % East
        if ECon
            conNodesMax(1, 5) = (ny - 1)*size(BW, 1) + (nx + 1);
            costNodesMax(1, 5) = 1;
        end

        % West
        if WCon
            conNodesMax(1, 4) = (ny - 1)*size(BW, 1) + (nx - 1);
            costNodesMax(1, 4) = 1;
        end

        % exploring diagonals
        %
        % NW \|/ NE
        %  ---+---
        % SW /|\ SE
        %
        SWCon = (ny + 1 > 0 && nx - 1 > 0) && (ny + 1 <= size(BW, 1) && nx - 1 <= size(BW,
        ↪  2)) && (BW(ny + 1, nx - 1) == 1);
        NWCon = (ny - 1 > 0 && nx - 1 > 0) && (ny - 1 <= size(BW, 1) && nx - 1 <= size(BW,
        ↪  2)) && (BW(ny - 1, nx - 1) == 1);
        SECon = (ny + 1 > 0 && nx + 1 > 0) && (ny + 1 <= size(BW, 1) && nx + 1 <= size(BW,
        ↪  2)) && (BW(ny + 1, nx + 1) == 1);
        NECon = (ny - 1 > 0 && nx + 1 > 0) && (ny - 1 <= size(BW, 1) && nx + 1 <= size(BW,
        ↪  2)) && (BW(ny - 1, nx + 1) == 1);

        % NE
        if NECon
            conNodesMax(1, 3) = (ny - 1 - 1)*size(BW, 1) + (nx + 1);
            costNodesMax(1, 3) = sqrt(2);
        end

        % NW
        if NWCon
            conNodesMax(1, 1) = (ny - 1 - 1)*size(BW, 1) + (nx - 1);
            costNodesMax(1, 1) = sqrt(2);
        end

        % SE
```

```
        if SECon
            conNodesMax(1, 8) = (ny + 1 - 1)*size(BW, 1) + (nx + 1);
            costNodesMax(1, 8) = sqrt(2);
        end

        % SW
        if SWCon
            conNodesMax(1, 6) = (ny + 1 - 1)*size(BW, 1) + (nx - 1);
            costNodesMax(1, 6) = sqrt(2);
        end

    end

    conNodes = conNodesMax(conNodesMax > 0);
    costNodes = costNodesMax(costNodesMax > 0);

end
```

Listing 5: `conNodescalc`: dynamically calculating dependencies.

After applying this crucial modification to analyse large-size images, four different starting and goal positions were provided to the algorithm to determine the optimal paths to traverse the house (see Figures 38 to 42). However, due to its inherent definition, the algorithm lacks knowledge of the Turtlebot's dimensions. Consequently, the defined paths are dangerously close to the walls, rendering them impractical for the Turtlebot to follow without risking collision. To address this issue, an obstacle enlargement technique was employed, leveraging the built-in MATLAB function `imdilate`, the symmetrical geometry of the Turtlebot, and a safety coefficient: for the purpose of path calculation the walls of each binarised image are thickened. (However, Figures 38 to 42 show the walls with their actual thickness.) The beneficial impact of this wall enlargement technique is evident from the comparison between Figures 38 and 39, where the paths are suitably adjusted to provide safe clearance for the Turtlebot. Notably, this technique also significantly reduces computation time, with wall-enlarged calculations taking only 11.12 seconds compared to 18.09 seconds without. For the sake of brevity, the other three paths that are presented are calculated from the wall-enlarged maps (Figures 40 to 42).

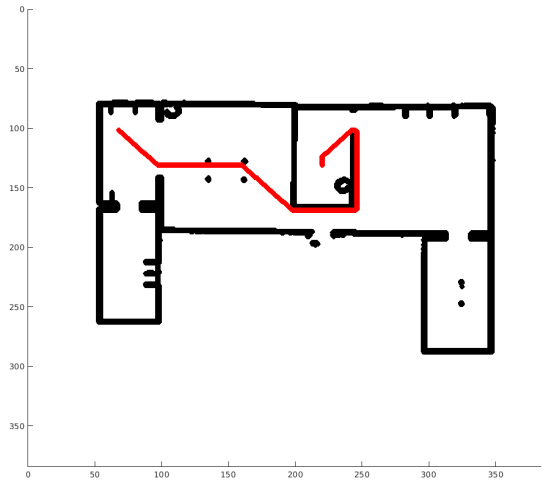

Figure 38: First path analysed, no wall enlargement.

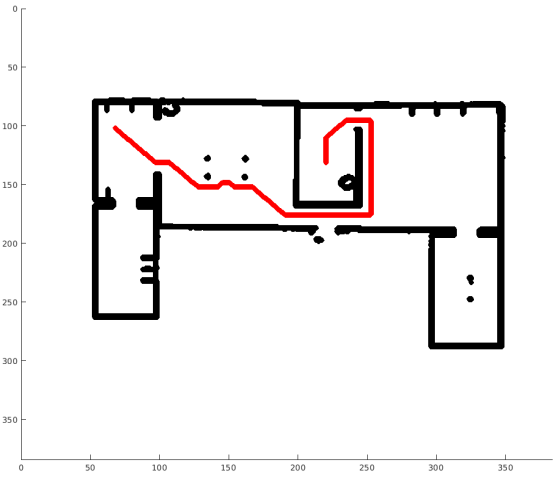

Figure 39: First path analysed, with wall enlargement.

Before delving into the Simulink model responsible for publishing velocity commands to the Turtlebot, two crucial issues need addressing. Firstly, the actor overseeing the localization of the Turtlebot within the provided map to allow feedback tracking of the trajectory: the ROS Navigation Stack. Utilising the Augmented Monte Carlo Localization from the Navigation Stack provides valuable data, including the position and orientation of the odometry frame relative to the map frame, acting as a fixed reference. This information is published as a message on the `/tf` ROS topic. However, due to an unidentified

Figure 40: Second path.

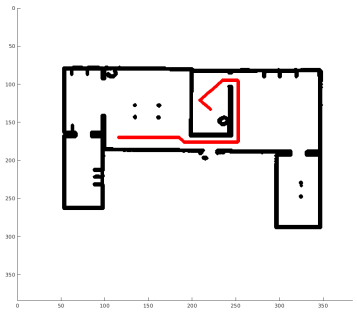

Figure 41: Third path.

Figure 42: Fourth path.

issue in Simulink[7], a Python listener was developed to extract this data and publish it within the `Covariance` field of a `PoseWithCovariance` message, whose code is reported in Listing 6.

```python
import roslib
import rospy
import math
import tf
from geometry_msgs.msg import PoseWithCovariance
from std_msgs.msg import Float64

def tf_list():
    rospy.init_node('tf_listener', anonymous=False)
    pub = rospy.Publisher('tf_listener_pub', PoseWithCovariance, queue_size = 10)
    listener = tf.TransformListener()
    rate = rospy.Rate(30)

    while not rospy.is_shutdown():
        try:
            (trans, rot) = listener.lookupTransform('/odom', '/map', rospy.Time(0))
        except (tf.LookupException, tf.ConnectivityException, tf.ExtrapolationException):
                    continue
        msg = PoseWithCovariance()
        msg.covariance = [trans[0], trans[1], trans[2], rot[0], rot[1], rot[2], rot[3]] +
        ↪  [0]*29
        pub.publish(msg)
        rate.sleep()

if __name__ == '__main__':
        try:
                tf_list()
        except rospy.ROSInterruptException:
                pass
```

Listing 6: Python listener for accessing localisation data.

The node running `tf_list` will operate alongside the other ROS services, the Gazebo world, the Rviz instance, and the Simulink model. It will read from the published topic containing the relationship between the child frame (`/odom`) and the parent frame (`/map`), providing to the Turtlebot the information about its position inside the map. Another issue to address involves the trajectory tracking for the Turtlebot. To guide the Turtlebot along the path using a simple policy, a feedback control law similar to that used in Assignment 2 (see Section II-B) was employed, albeit with significant modifications. However, tracking a trajectory consisting of a large number of points poses challenges: conflicts may arise with crossed points, and it is not necessary to follow a dense set of points on a perfectly straight line. To extract meaningful waypoints from the A*-generated trajectory, a semi-automatic method was implemented. This method involves numerically calculating the second

[7]Despite attempts to incorporate a subscriber block into the Simulink model for computing and publishing command velocities, it consistently failed for reasons that remained unknown. Surprisingly, the only `/tf` messages accessible were those related to the links between the base and the wheels, rather than the desired localization data.

derivative of the trajectory as a function of both x and y directions. Only points where both derivatives exceed a threshold are considered meaningful, indicating a turn. While the code is not provided here for brevity, interested readers can find it in the GitHub repository[8]. The result of this procedure is illustrated in Figure 43, where the red trajectory to be tracked is converted into a set of meaningful waypoints to reach in terms of x and y position and orientation: the orientation requirement associated to each waypoint is the angle of the distance vector from the waypoint to the next one.

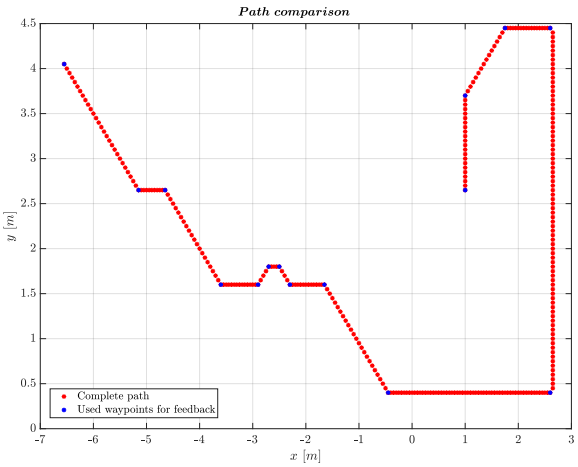

Figure 43: Trajectory to waypoints.

The Simulink model encapsulating all the pipeline for controlling the Turtlebot is reported in Figure 44.

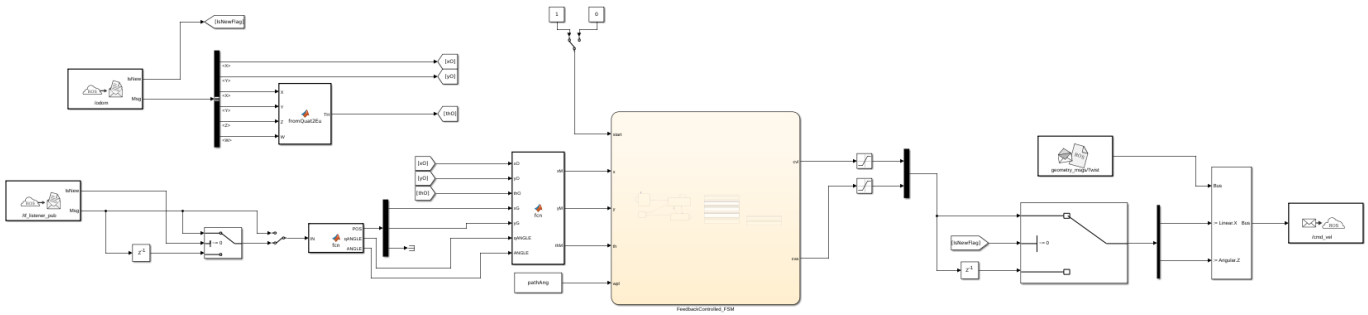

Figure 44: Complete Simulink model.

A Finite State Machine (FSM) (the yellow block in Figure 44) has been employed to implement the feedback control. The components outside the FSM closely resemble those in the Simulink model for Assignment 2, with the exception of the `/tf_listener_pub` subscriber, which reads the coordinate changes from the `/map` frame to the `/odom` one. This information is utilised in the final MATLAB function before the FSM to convert the coordinates from the `/odom` frame to the `/map` frame. This conversion allows to understand the Turtlebot's position in the map, in order to compare it with the position of the trajectory's waypoints, which in turn allows their tracking. The FSM's operation, illustrated in Figure 45, is straightforward in terms of state transitions. `cvlCalc` is the crucial function that is responsible for computing the longitudinal command velocity. Two additional functions are employed in the feedback control system: `disterrfun` and `cvaCalc`. The `disterrfun` function calculates the waypoints errors in both Cartesian distance (`Dist`) and angular position (`ErrTh`). Similar to the approach in Assignment 2, the angular reference is the direction of the distance vector when the Turtlebot is far from the waypoint and coincides with the waypoint's orientation when it is close. The `cvaCalc` simply implements a proportional feedback controller for calculating the angular command velocity.

Given their simplicity and their partial discussion in Assignment 2, we do not report the description of `disterrfun` and `cvaCalc`: the interested reader is referred to the GitHub repository[9]. Instead, we report the feedback policy `cvlCalc`.

[8]https://github.com/JacopoPorzio/AutonomousVehicles
[9]https://github.com/JacopoPorzio/AutonomousVehicles

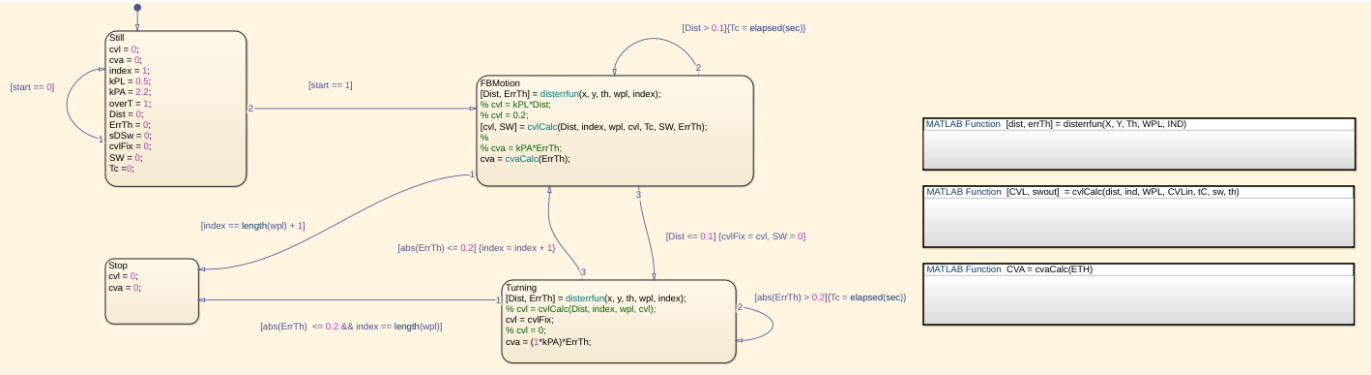

Figure 45: Finite State Machine controlling the velocity output.

```
function [CVL, swout]  = cvlCalc(dist, ind, WPL, CVLin, tC, sw, th)

    persistent a
    C = 1.25;
    kR1 = 0.1;
    kR2 = 0.5;
    a3 = 0.005;
    TH = WPL(ind, 3);
    gamma = 5;

    if dist <= 0.5 && sw == 0
        cvlin0 = CVLin;
        if 1e-2 - pi/2 < abs(TH - th) && abs(TH - th) < pi + 1e-2
            a = gamma*((kR1*cvlin0)^2 - cvlin0^2)*C;
        else
            a = ((kR2*cvlin0)^2 - cvlin0^2)*C;
        end
        swout = 1;
    else
        a = 0;
        swout = 0;
    end

    if dist <= 0.5
        CVL = CVLin + a*tC;
    else
        cvlTmp = CVLin + a3;
        CVL = (cvlTmp)*(cvlTmp <= 0.2) + (0.2)*(cvlTmp > 0.2);
    end

    CVL = CVL*(CVL >= cvlTh) + cvlTh*(CVL < cvlTh);

end
```

Listing 7: Feedback policy.

The first `if-else` statement deals with the situation in which the Turtlebot is close to the waypoint and a switch variable `sw` is 0 (see Figure 45: the variable `sw` resets to 0 upon entering the `Turning` state, preventing negative acceleration between closely spaced waypoints). Under these circumstances, a constant deceleration law is computed, with the deceleration rate being stronger for tighter curves. The second `if-else` statement enforces the deceleration law when the Turtlebot nears the waypoint, applying a simple constant positive acceleration velocity law with saturation considered. The saturation is considered here, because otherwise the FSM would be agnostic of the saturation that is imposed to real system, which is applied outside the FSM itself for convenience. The final statement before the function's end prevents excessive slowing down of the Turtlebot.

To streamline the discussion, simulations are presented for only two out of the four identified paths. The results of the first path simulation are shown in Figures 46 to 49. Figure 46 shows that the discrepancies between the theoretical A*-calculated trajectory with the simulated one are minimal. The differences are attributable to imperfections in the map, which may not

represent the real-world environment with perfect accuracy, and the employed feedback policy, according to which the Turtlebot does not come to a complete stop even during tight turns.

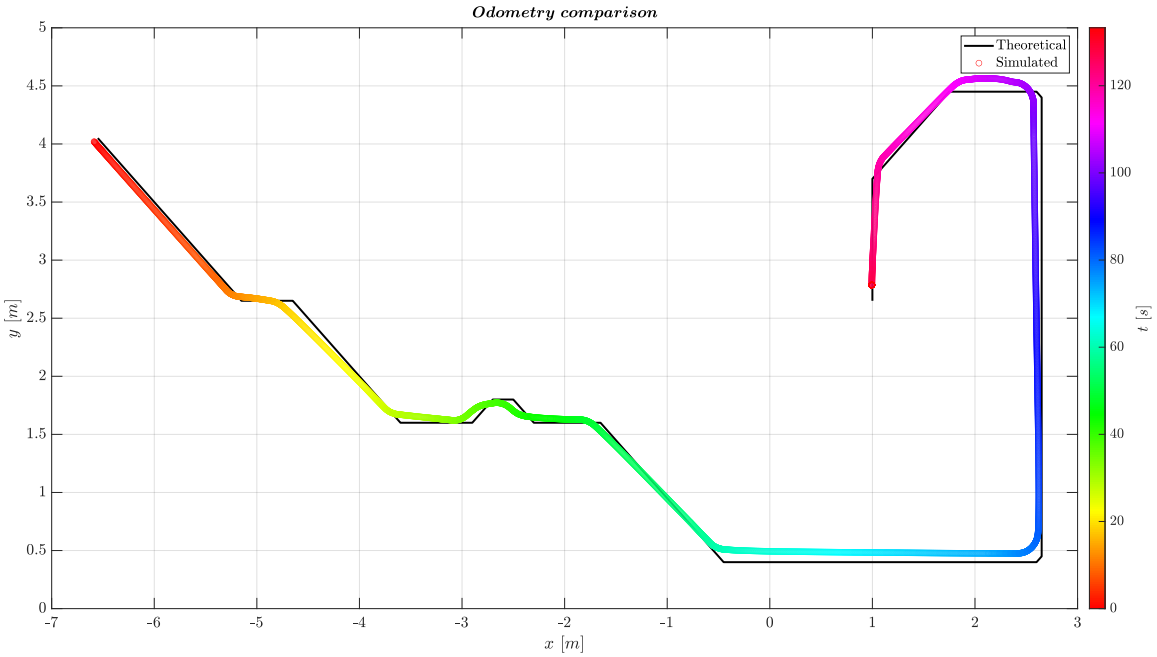

Figure 46: First experiment. Comparison between the theoretical trajectory versus the actual one.

The imperfections in the map are reflected in Figure 49. Ideally, if the map accurately represented the real-world environment, the curves on this graph would appear as horizontal straight lines (not necessarily zero-valued). However, the observed variations, though present, are relatively minor, resulting in a slight deviation between the theoretical and actual trajectories. An interesting observation is the tooth-shaped irregularity shortly after the 20-second mark, where the Turtlebot makes its first series of turns (as shown in Figure 46). This results in the localization updating twice, causing a slight deviation between the actual and theoretical trajectories.

Examining the results of the experiment where the Turtlebot tracks the second set of waypoints (see Figures 50 to 53), we observe a similar outcome. Once again, the differences between the theoretical trajectory and the actual one (Figure 50) are relatively small and can be attributed to the same factors, namely the non-zero velocity during turns and the imperfect localization. Notably, from Figure 53, we can see that the sequence of turns in the initial part of the trajectory puts a considerable demand on the localization services, resulting in high-frequency disturbances. Nonetheless, as depicted in Figure 50, the tracking is well-maintained. In conclusion, to achieve better results, it is crucial to properly tune the `gmapping` tool to produce a more consistent map, facilitating faster and less-disturbed localization, and to implement more refined control policies, increasing the quality of the tracking.

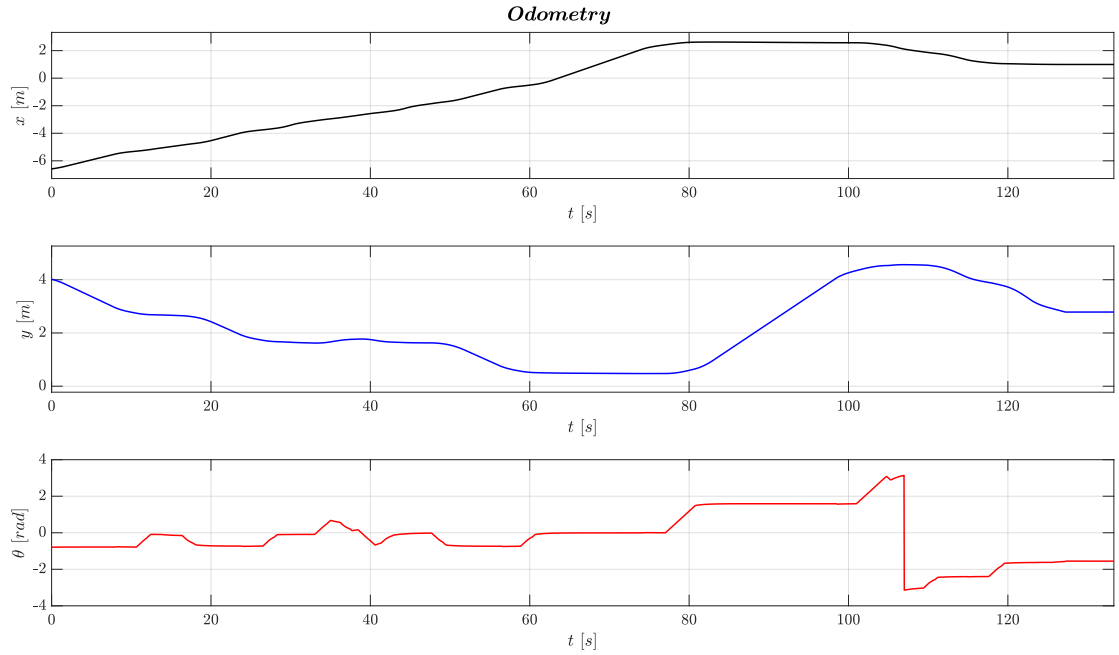

Figure 47: First experiment. Odometry evolution in time.

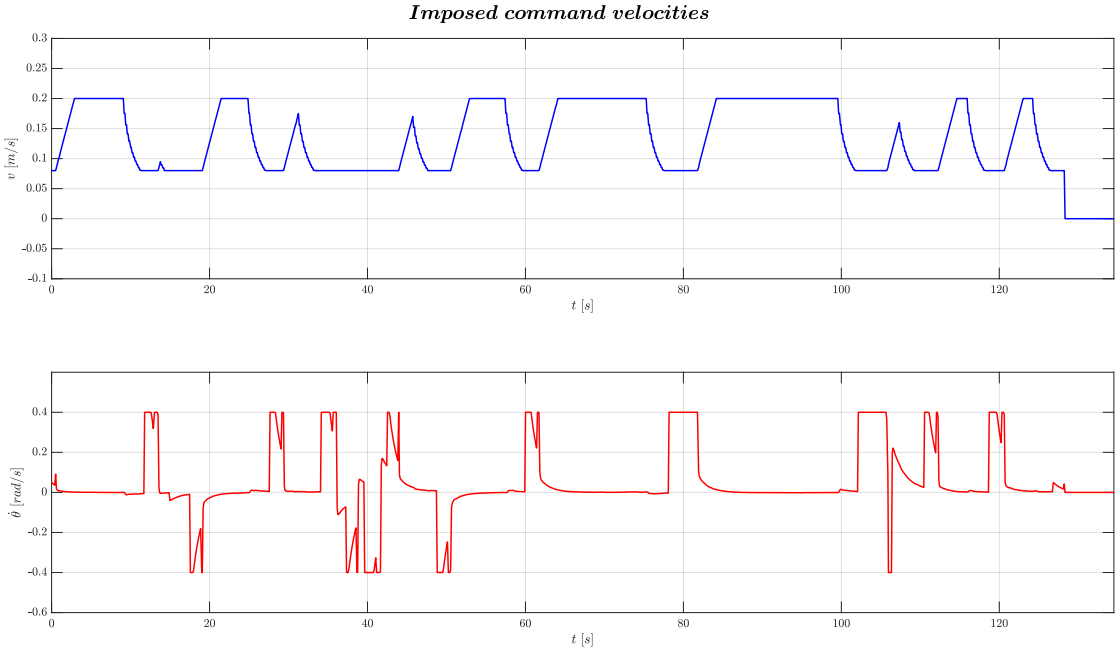

Figure 48: First experiment. Imposed command velocities.

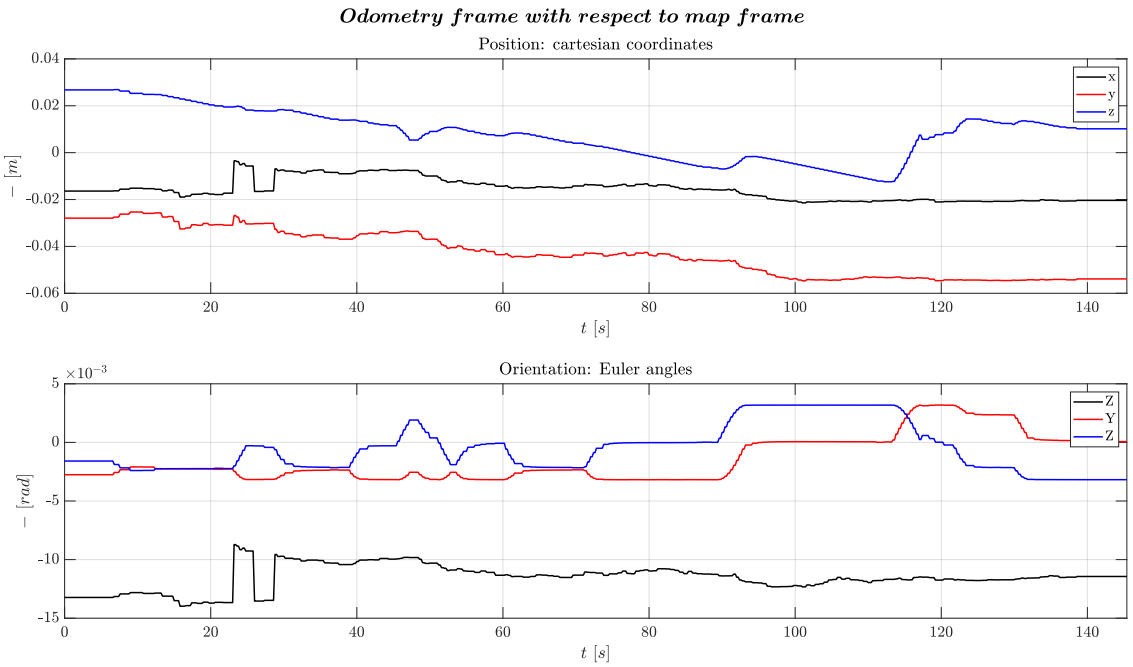

Figure 49: First experiment. Transformation from map frame to odometry frame.

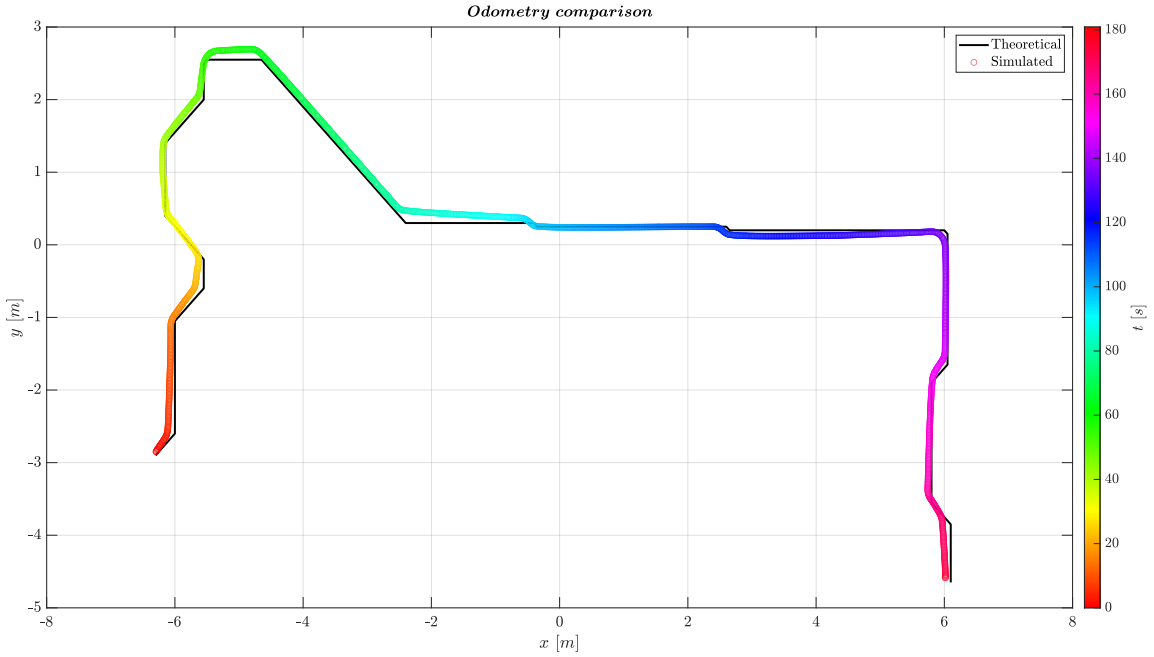

Figure 50: Second experiment. Comparison between the theoretical trajectory versus the actual one.

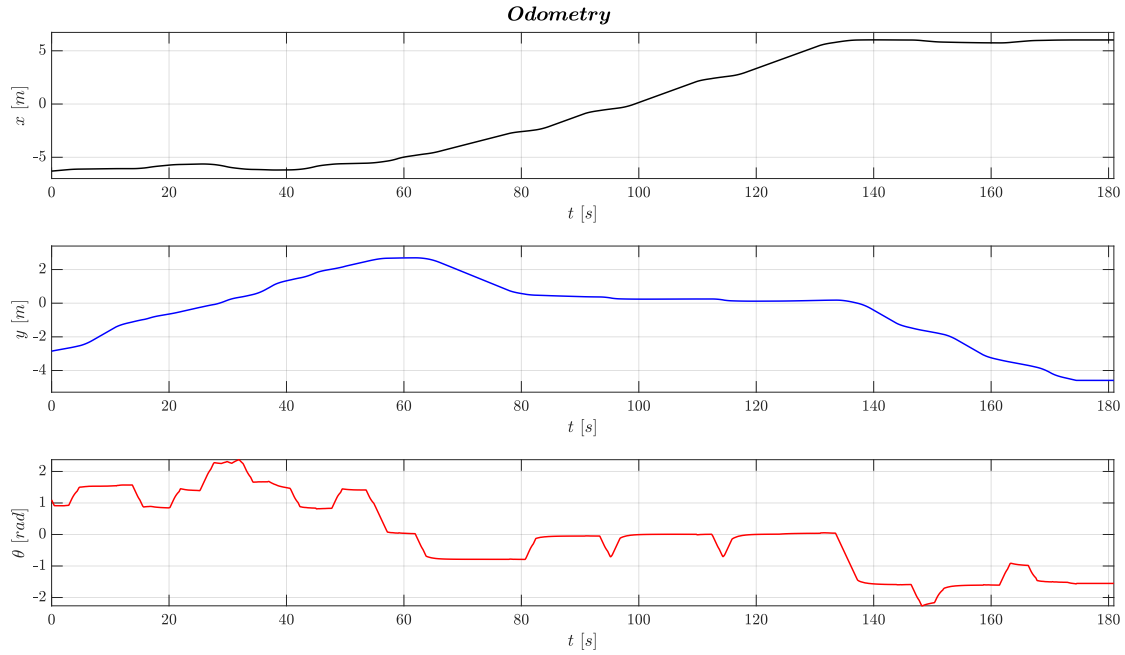

Figure 51: Second experiment. Odometry evolution in time.

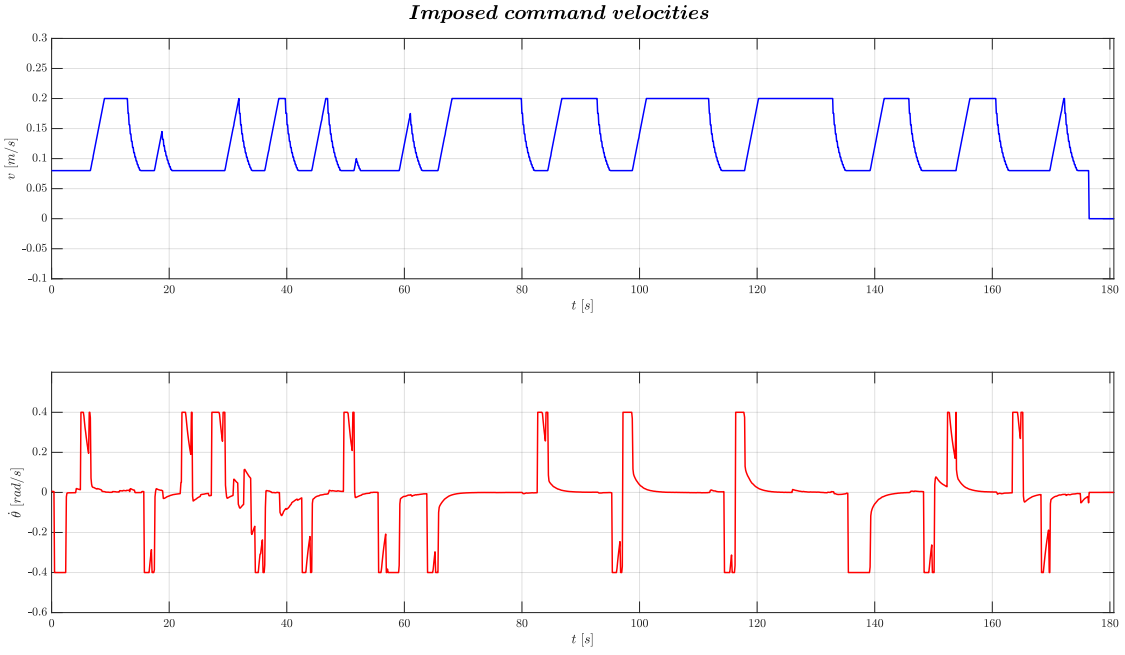

Figure 52: Second experiment. Imposed command velocities.

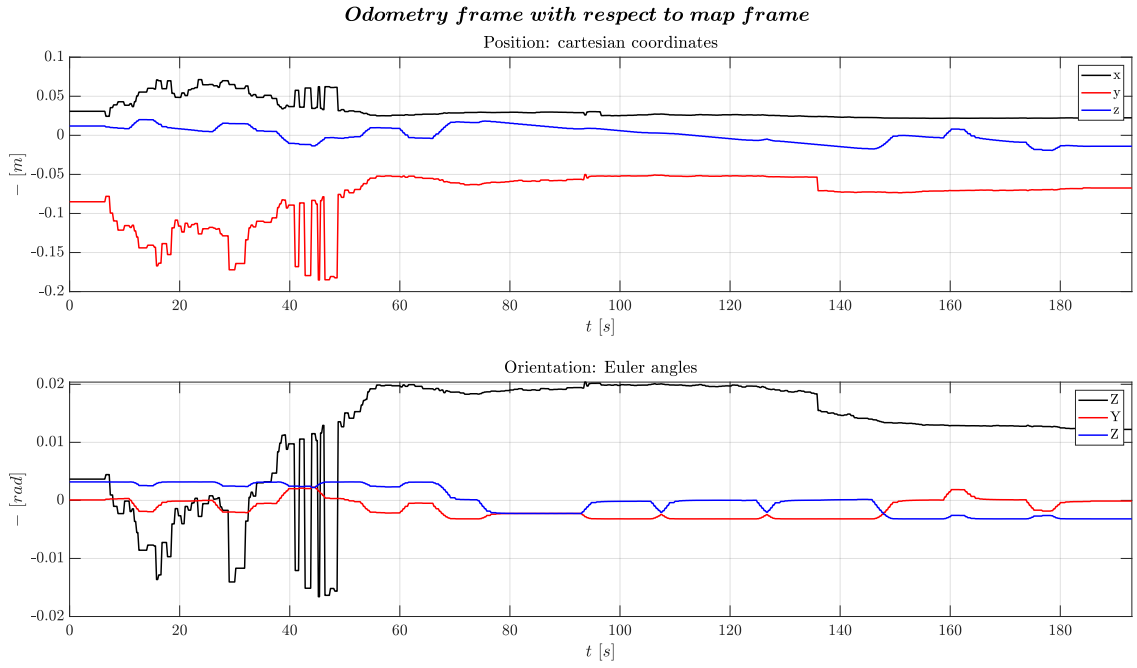

Figure 53: Second experiment. Transformation from map frame to odometry frame.

B. Final Project 3

This project asks to define a path of waypoints: in between these waypoints, red and green spheres have to be positioned. The Turtlebot must navigate to each of the waypoints, avoiding the spheres with a conditional logic: if the sphere is red, it must overtake it on the left; otherwise, it should pass on the right. The robot utilised for this task is the Turtlebot Waffle Pi.

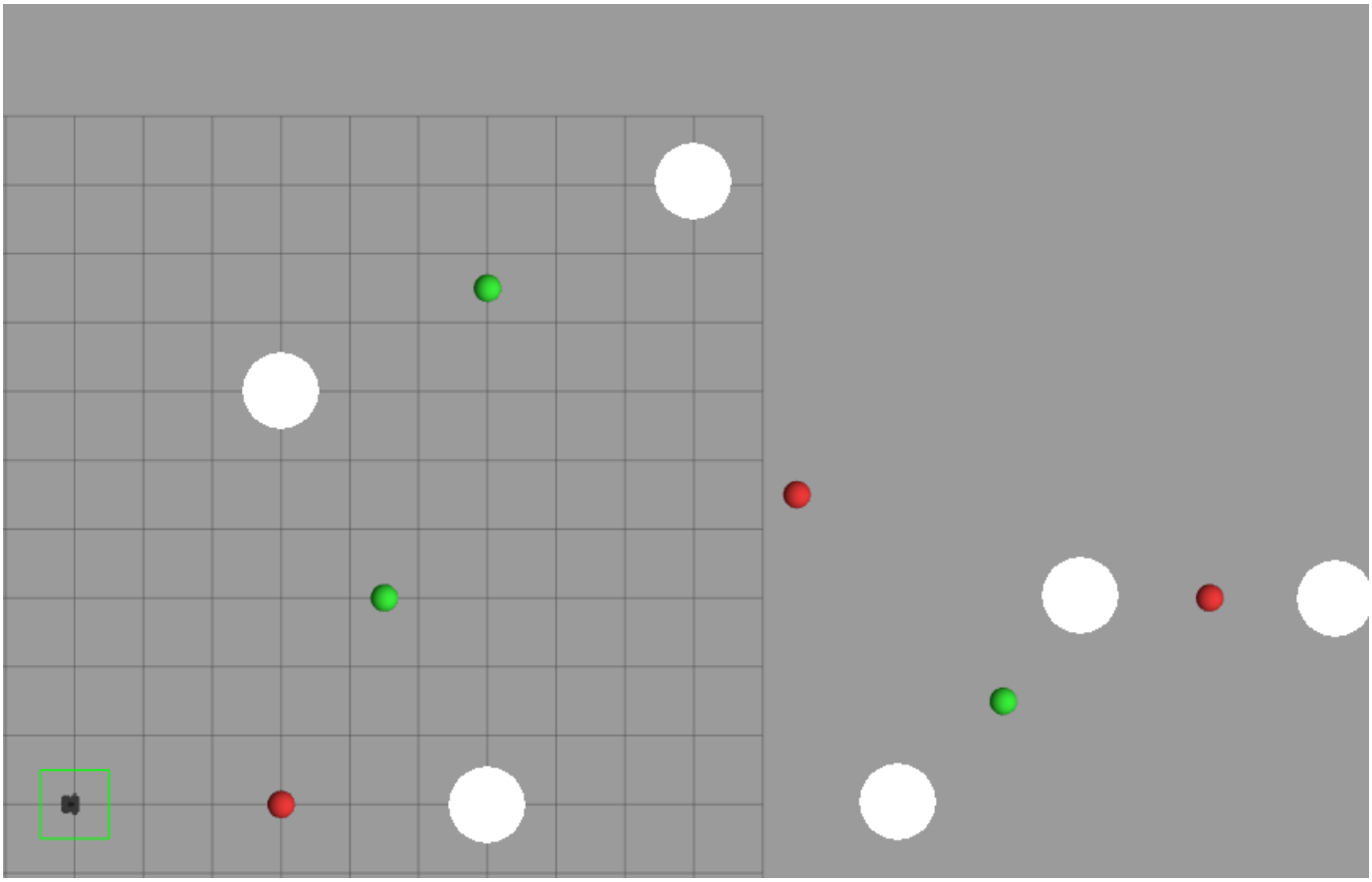

Figure 54: The path with white waypoints and coloured spheres.

The Simulink model for this task (which is not shown in its entirety for the sake of brevity) comprises two main subsystems. The first subsystem (see Figure 55) processes camera images to detect and identify colored balls. The second subsystem (see Figure 56) reads odometry data and scans the Turtlebot's surroundings: its goal is to compute a feedback law using a Finite State Machine (FSM) approach (see Figure 57), which implements a control law that is strongly similar to the one developed for Assignment 2 Section II-B, albeit in an FSM fashion.

The implementation difference between the FSM and the control law defined in Assignment 2 lies in the presence of the `Overtaking` state. While the `FBMotion` and `Turning` states allow the Turtlebot to perform the same actions as before, respectively calculating feedback controls for navigating towards the waypoint and stopping at turns, the `Overtaking` state has been introduced to navigate around obstacles.

During overtaking, the Turtlebot executes a manoeuvre around the obstacle using a feedback control system. This system guides the Turtlebot along a path defined by a set of waypoints, which are calculated before entering the `Overtaking` state. These waypoints are strategically positioned around the sphere based on its radius, which is estimated from the `Area` variable outputted by the image processing pipeline defined in Figure 55, and the sphere's distance from the Turtlebot, calculated from the LiDAR scans. Throughout the overtaking process, the Turtlebot maintains a constant longitudinal velocity, preserving the speed set before entering the overtaking state. The MATLAB function `OTdisterrfun` incorporates the right swerving direction according to the color of the sphere into the error calculation for the angular command velocity. In order to initiate the overtaking manoeuvre, certain conditions must be met, including a minimum distance between the Turtlebot and the obstacle; additionally, the controller verifies that the obstacle closest to the Turtlebot is actually in front of it by comparing the angle corresponding to the minimum range detected by the LIDAR and the Turtlebot's yaw. Figures 58 to 60 illustrate the results of the experiment. The observed behaviours are effectively smooth and spike-less.

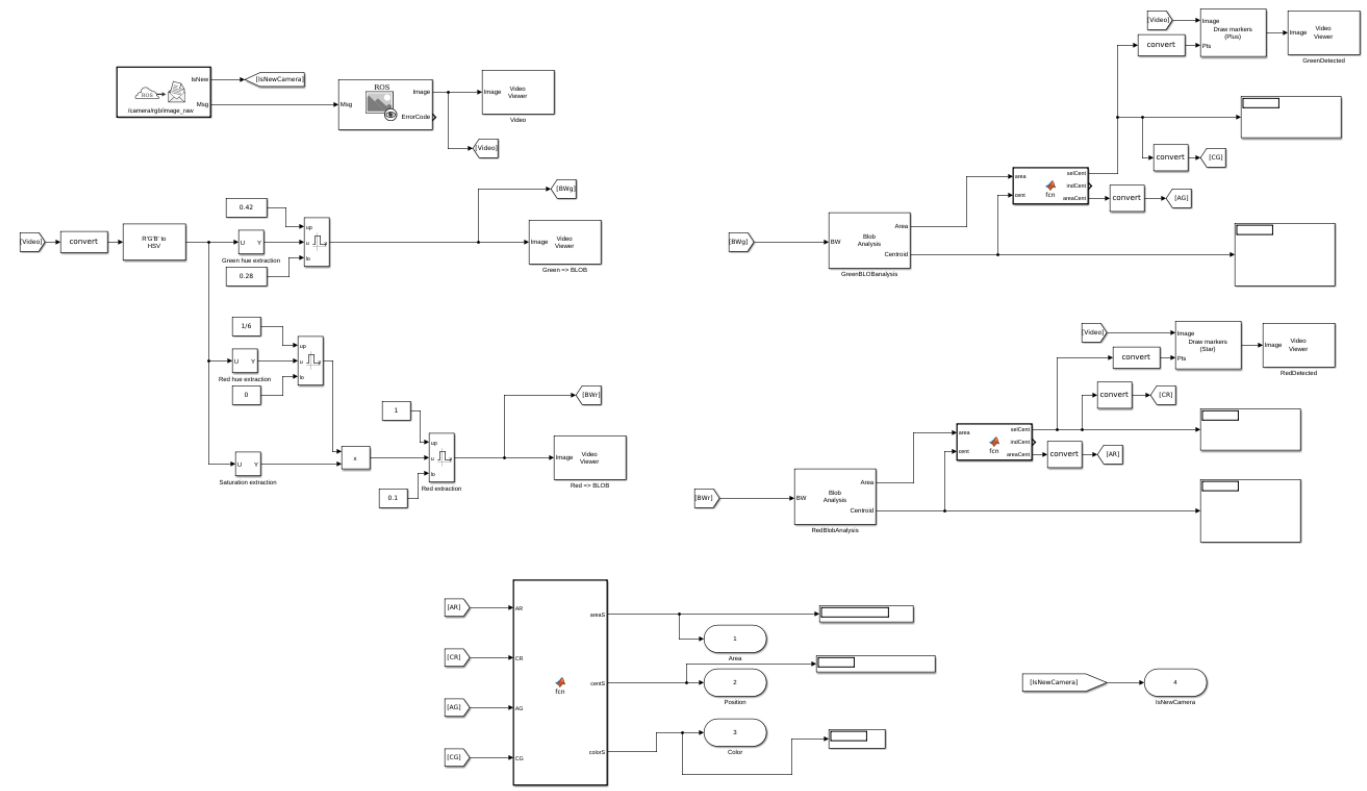

Figure 55: Camera reading subsystem.

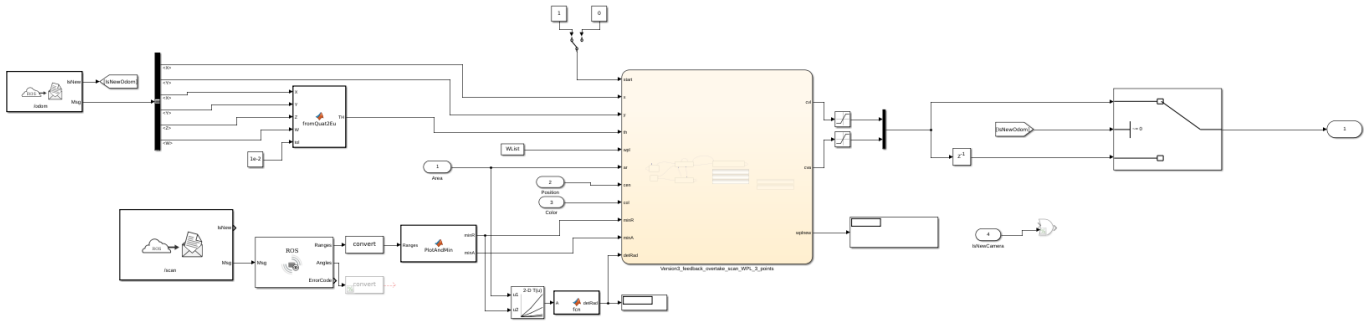

Figure 56: Feedback controller subsystem.

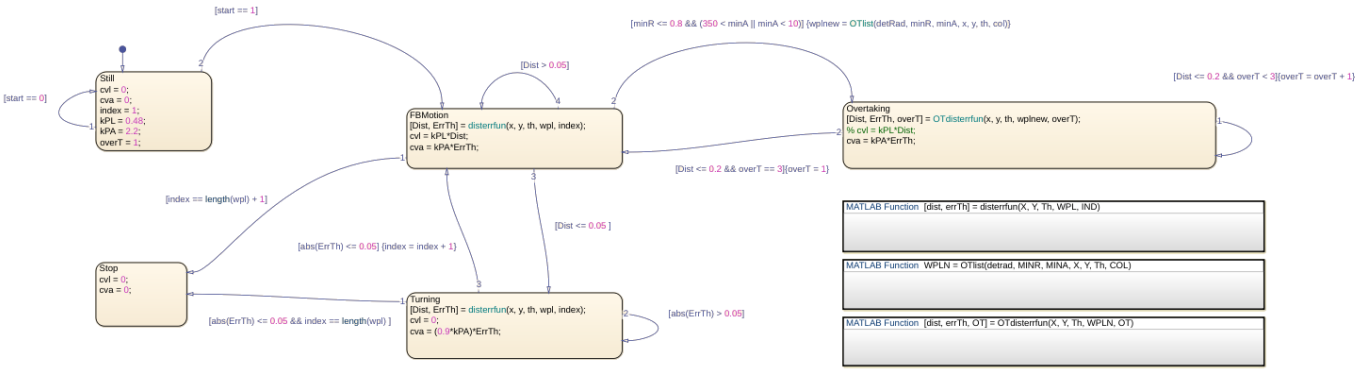

Figure 57: Finite State Machine implementing the feedback control.

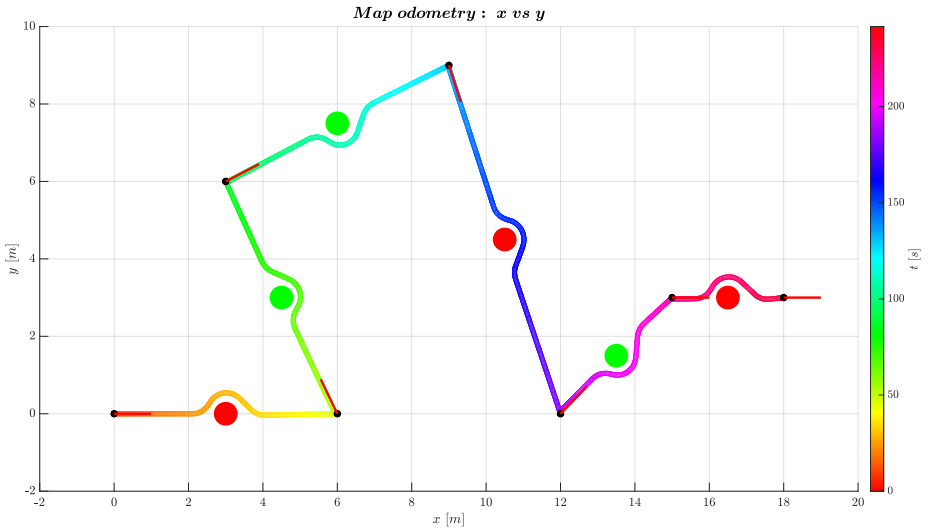

Figure 58: Turtlebot's trajectory with black-dotted waypoints, with red bars being their orientation.

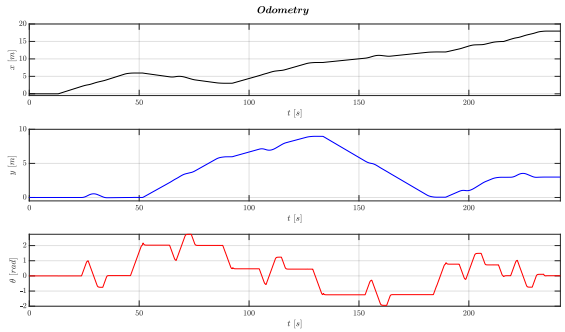

Figure 59: Odometry plots.

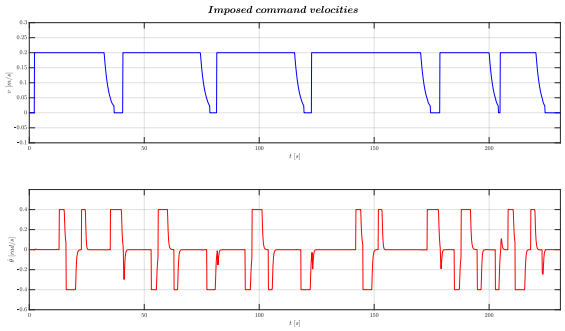

Figure 60: Imposed command velocities.