A.Y. 2022 - 2023

Autonomous Vehicles course

Report on assignments and final projects

Professors

Francesco Braghin

Stefano Arrigoni

Student

Jacopo Porzio

10683989

988718

Link to the github repository on gitfront:
https://gitfront.io/r/Heisenasd/SaS6Rt6oP3V6/AutonomousVehicles/

Index

First part: assignments

1. Assignment 1

The first assignments consists in the analysis of a previously recorded ROS bag to extract from it some relevant information. The bag contains the odometry and the scanned space around the Turtlebot Waffle Pi, which traveled following a user-defined trajectory in the default Gazebo Turtlebot world.

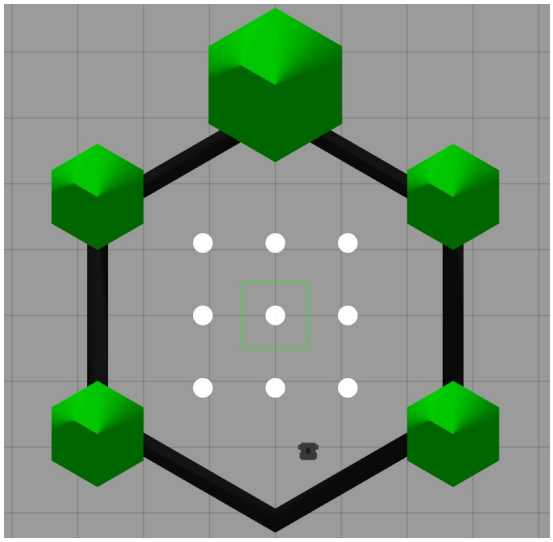

Figure 1: the Waffle Pi inside the reference world.

The first stage of the analysis starts with the estimation of the command velocity imposed to the Turtlebot. Thus, we need to first identify the linear and the angular displacement; of the latter, for obvious reasons only the yaw is relevant, thus, the pitch and roll behaviors will be neglected.
No particular issues were encountered for extracting the linear odometry from the bag, while a little effort had to be spent on translating the angular coordinates, given in quaternions, to Euler angles.

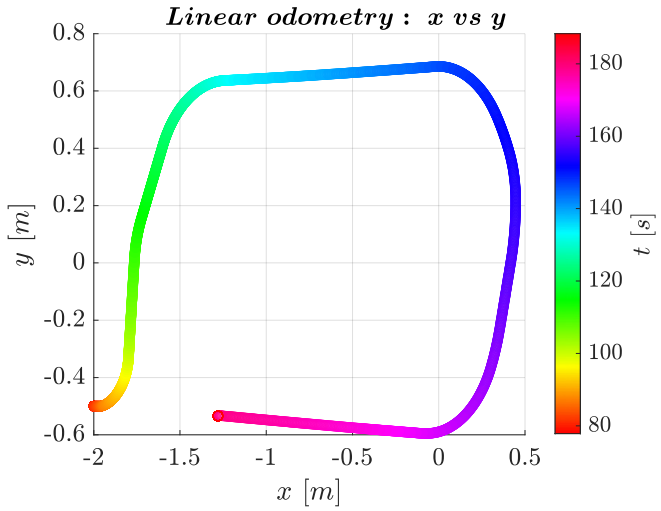

Figure 2: linear odometry.

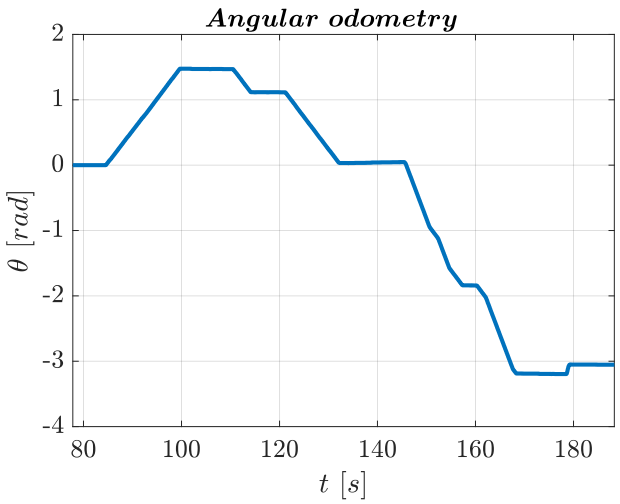

Figure 3: angular odometry.

Once the position information has been extracted, the analysis can further proceed and the imposed command velocity can be estimated. This has been done by approximating the velocity with the forward finite difference scheme and then by smoothing this preliminary velocity estimation with a moving average, in order to decrease the noise produced by numerical differentiation.

Forward finite difference approximation: $\left.\frac{dq}{dt}\right|_k \approx \frac{q(k+1)-q(k)}{\Delta t}$

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

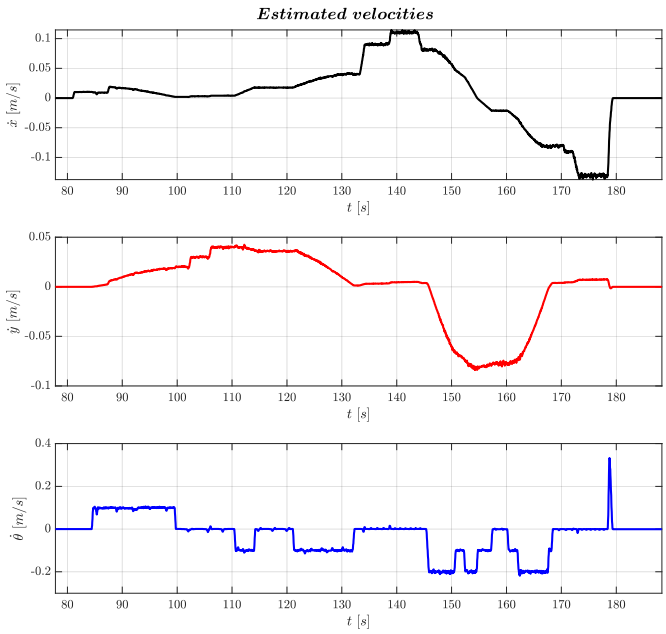

Figure 4: estimated linear and angular velocities.

The Turtlebot takes as input a longitudinal velocity and an angular one: therefore, for what it concerns the latter one, it's already what we need, but we have to calculate the Pythagorean addition of the two velocities in *x* and *y* direction in order to obtain the longitudinal one.

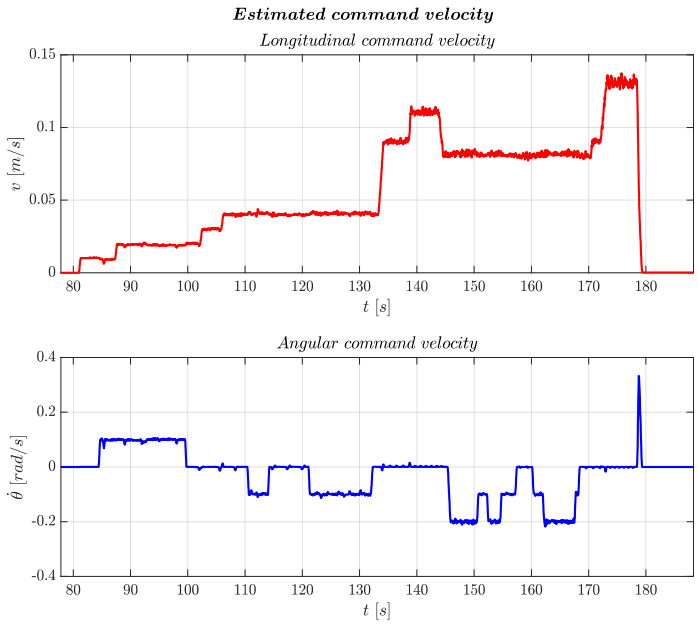

Figure 5: estimated linear and angular command velocities.

Now that we have estimated the imposed command velocities, we can try to publish them to the robot in a feedforward fashion, in order to draw some conclusions on the accuracy in the estimation process.
The estimated commands were published to the ROS topic */cmd_vel* by means of a Simulink scheme, which is displayed in figure 6.

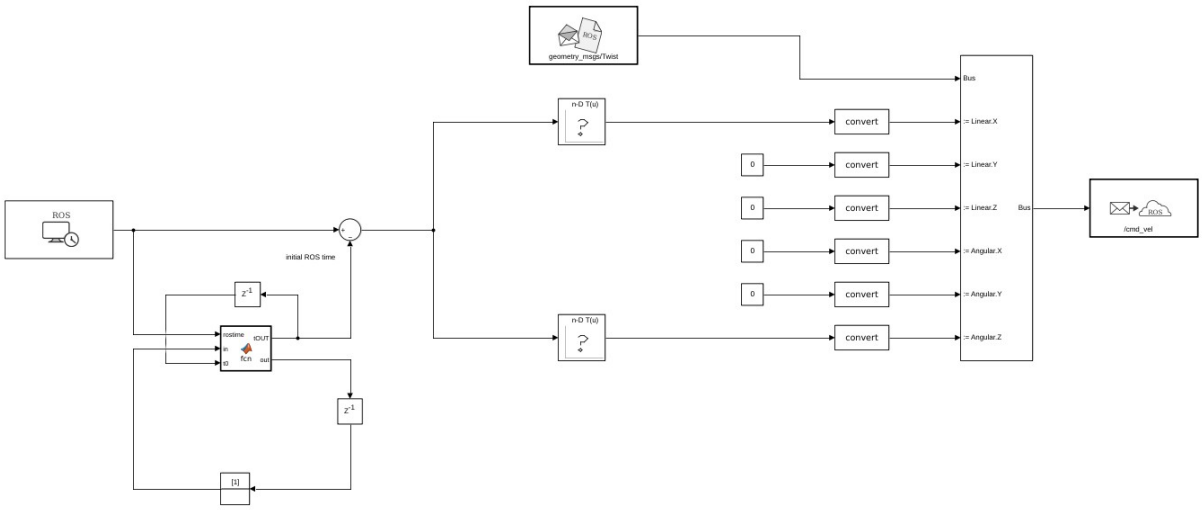

Figure 6: Simulink publication model.

The blocks on the left of the model are used in order to enter a look-up table with the correct time instant in the Simulink reference frame, which is obtained by means of a difference between the the actual time instant and the time when the simulation started in the ROS reference frame.
After having published the command velocities and recorded a bag, we can advance the first part of the present analysis to its final stage: we just have to repeat the operations that have been already carried out for the analysis of the provided bag, but we will now focus on drawing comparisons between the two cases.

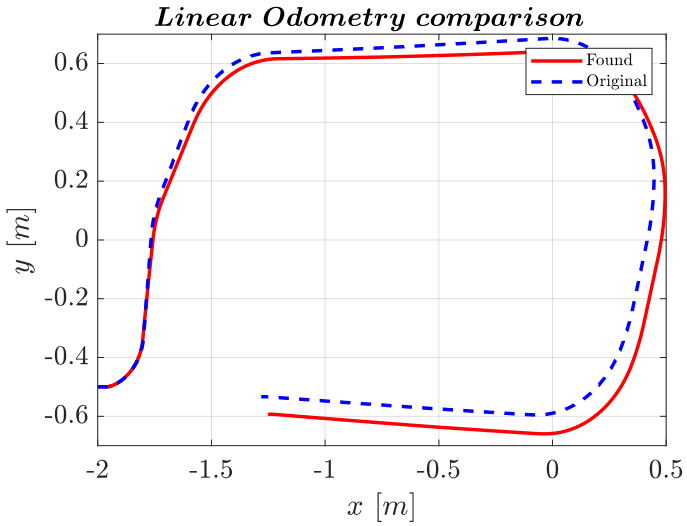

Figure 7: comparison between the found and original linear odometry.

We see a slight difference between the original trajectory and the one obtained by the imposition of the estimated command velocities. We can imagine that the distinction arose from the fact that the movement has been imposed in a feedforward fashion with a set of commands that have been defined by numerical differentiation and a simple moving average filtering that cannot yield unitary effectiveness.

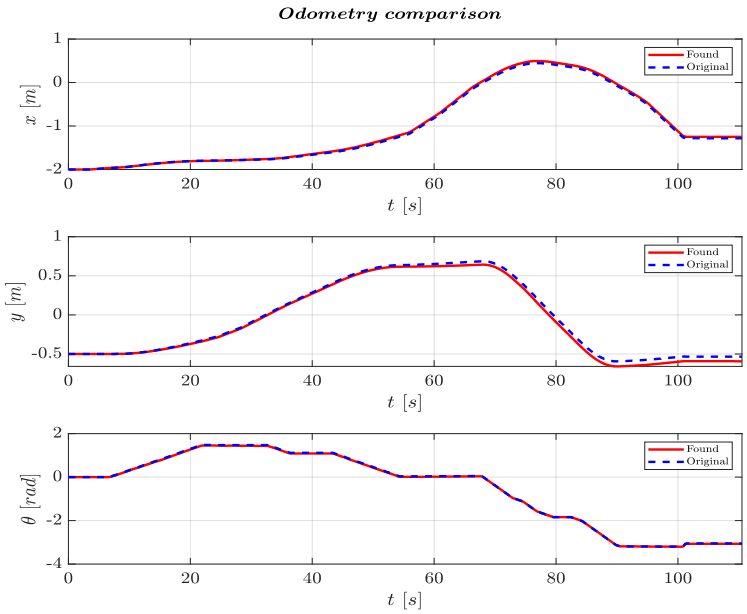

Figure 8: comparison between the found and original odometry.

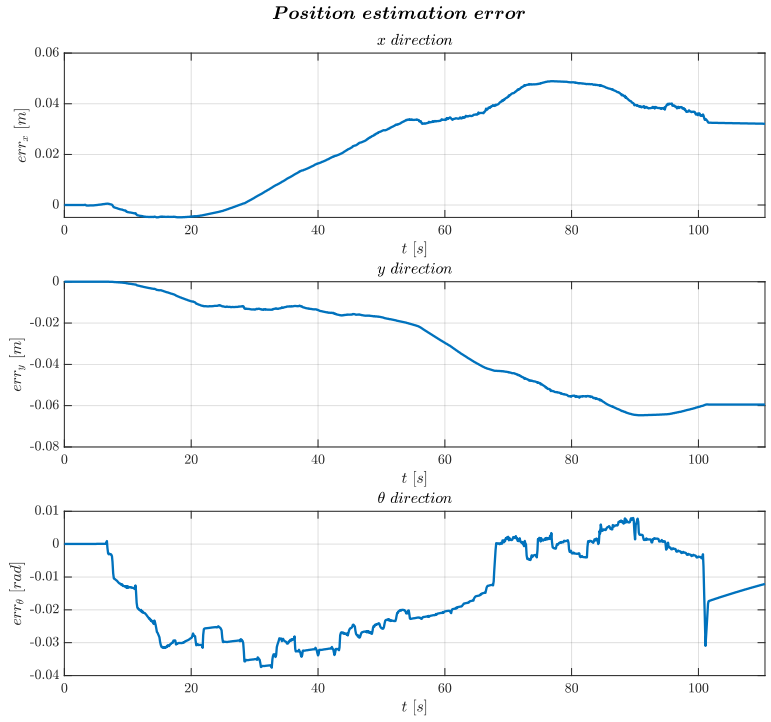

Figure 9: error between the trajectories.

In figure 8 we see that the plots are more or less coincident, and the errors in figure 9 are rather small: thus, the responsible of the deviation from the original is, indeed, to be found in the noisy command velocity which has been imposed, to which is superimposed an inevitable noise coming from software's side.
Even trying to lowpass this command velocity in order to have it smoother doesn't make the situation better, because, as we can see, the noise aren't that big in entity.
In the attempt to have a perfect following of the original trajectory, a procedure using the peaks in the acceleration to smooth even more the velocity estimate has been tried, but no significant results have been found[1].

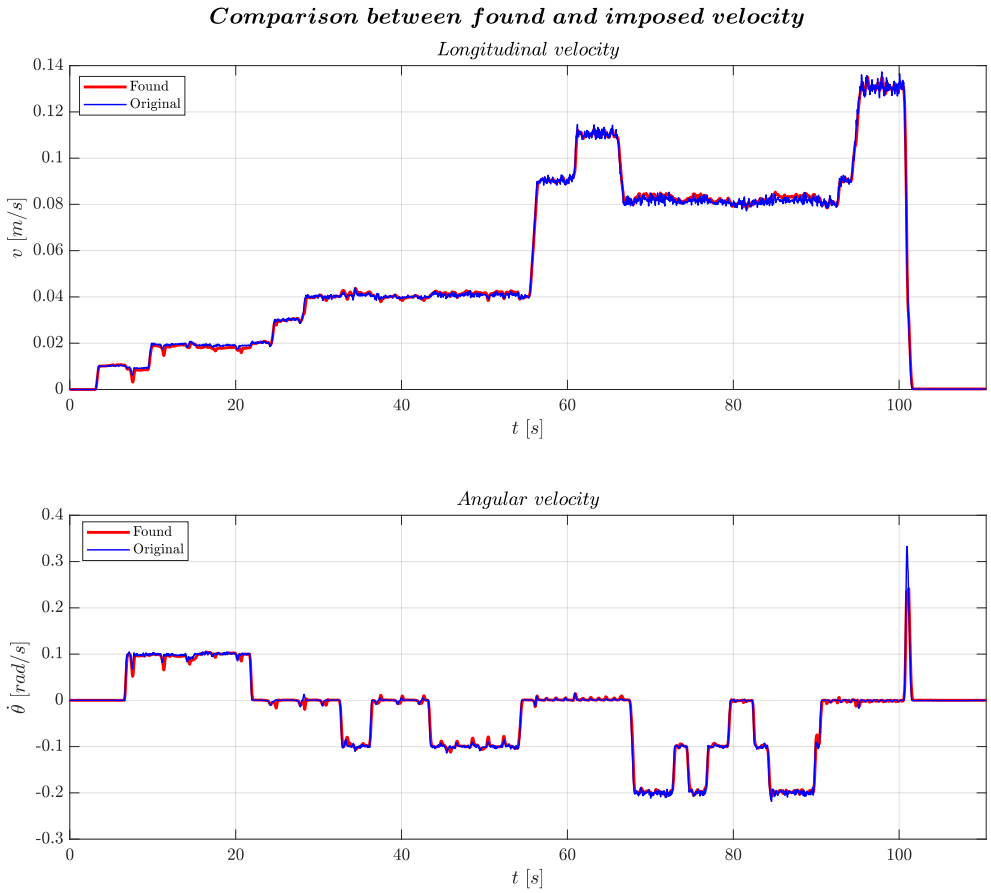

Figure 10: comparison between the imposed (found as estimation) and the original command velocities.

In figure 10 we have the comparison between the two estimations of command velocities imposed: from the provided bag and from the recording.

1 This procedure took advantage of the specific situation under analysis. The traits with almost constant velocity are indicated in the acceleration array by two spikes, one at the start and one at the end of the aforementioned trait. This fact was used to assign an averaged and constant value to these traits, but this process introduces an error, too. Check on the github repository.

The second part of the assignment asks to estimate the minimum distance between the Turtlebot and obstacles during its trajectory. This task is particularly simple as the scanned space coming from the LIDAR sensor can be found in the bag. The scanned points are given in the local Turtlebot reference frame, but they are given as ranges in a 360-cell long array, i.e. one point per degree. Hence, we first have to associate this "polar" information to a Cartesian one, simply doing at each time instant *k*:

$$\begin{aligned} \boldsymbol{\theta} &= (0,\ 1,\ \dots\ 360)^T \\ \boldsymbol{x}(k) &= \boldsymbol{Range}(k) \cdot \cos(\boldsymbol{\theta}) \\ \boldsymbol{y}(k) &= \boldsymbol{Range}(k) \cdot \sin(\boldsymbol{\theta}) \end{aligned}$$

The distance between an obstacle and the Turtlebot at the *k*-th instant is:

$$\boldsymbol{d}(k) = \sqrt{\boldsymbol{x}^2(k) + \boldsymbol{y}^2(k)}$$

Having this done, the only operation needed is finding the minimum values of $\boldsymbol{d}$ at each time instant.

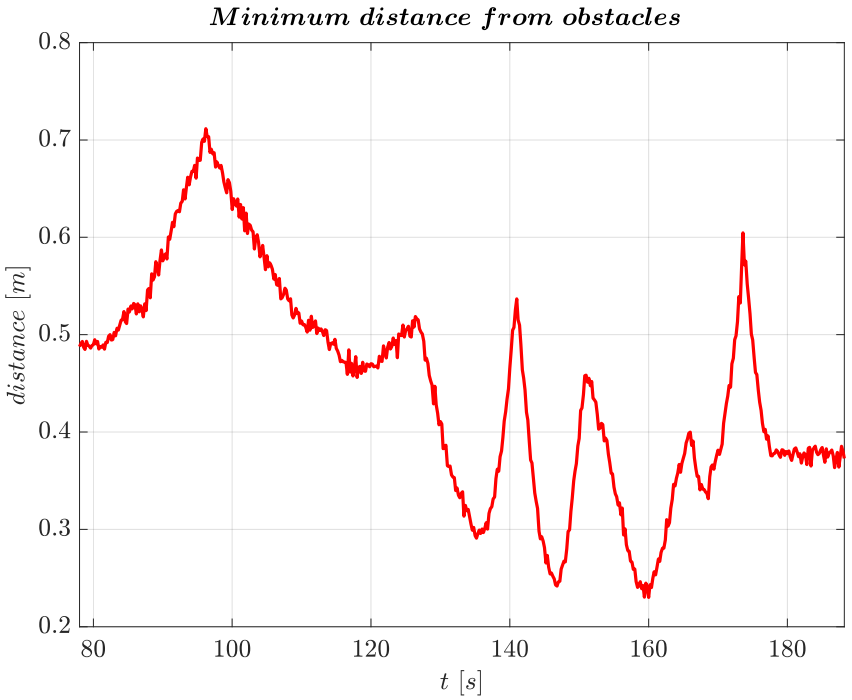

Figure 11: minimum Turtlebot - obstacles distance during the trajectory.

It is suggested to give a look to the github repository for secondary considerations that have been carried out, but that are out of the scope of the present analysis, such as an animated *.gif* that shows the motion of the Turtlebot in the scanned space, highlighting with a black line the distance between it and the nearest obstacle.

2. Assignment 2

For the second assignment we are provided a list of 10 waypoints through which the Turtlebot Burger has to pass. The easiest and safest way to make sure the Turtlebot will reach these waypoints is to implement a feedback control policy. Computing an open-loop control law would be needlessly difficult and we would be given no warranty that the Turtlebot will actually arrive at those waypoints with the wanted effectiveness. Therefore, a simple proportional feedback control action will be implemented, namely the longitudinal and angular command velocities to be imposed will be calculated as proportional to the difference between the pose of the Turtlebot and the goal's one.
As it is obvious, in order to obtain some given performances, the scheme has to deviate a little from the simplicity of a trivial proportional feedback scheme.

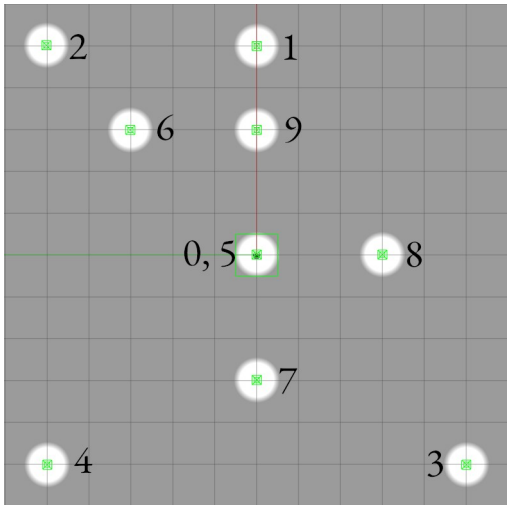

Figure 12: waypoints to reach with the numeration.

The waypoints are provided not only with a position requirement, but also with an orientation one.
The strategy that has been implemented in order to reach these two requisites has been the following: start from the *(i-1)*-th waypoint ,considering the magnitude of the distance vector $\boldsymbol{d}$ between the Turtlebot and the *i*-th waypoint to define a proportional law for the longitudinal velocity, while using its angle for calculating a proportional control policy for the angular velocity.
If θ is the orientation of Turtlebot and $\boldsymbol{d}$ the distance vector:

$$\textit{Feedback policy}:$$
$$\boldsymbol{d} = d\,e^{j\phi}$$
$$v = k_{P,LON}\,d$$
$$\omega = k_{P,ANG}(\phi - \theta)$$

The above formula is used until the Turtlebot is close enough to the waypoint: there, it is stopped (thus, $v = 0$) and the new reference angle is the waypoint's orientation. When the difference between the Turtlebot's orientation and the waypoint's one is under a tolerance, the Turtlebot starts again in order to reach the new waypoint, using the aforementioned feedback policy centered in it.
A challenge that was encountered is making realistic the Turtlebot movement in the curves. The odometry topic outputs the orientation in quaternions, as previously said, which is converted in Euler angles through a built-in Matlab function. The obtained orientation of the Turtlebot spans from $-\pi$ to π, while the waypoints' orientation lives in the interval from 0 to 2π: making the conversion of the latter to match the orientation of the Turtlebot isn't enough to have a realistic transitioning in the curves.

In fact, considering the case when the Turtlebot is oriented as $\theta = 0.75\pi$ and we want it to rotate to reach a heading of $\theta_R = 1.25\pi$, let's say we implement the following feedback policy to make the Turtlebot rotate:

$$\omega = k_{p,ANG}(\theta_R - \theta)$$

We can experience some problem if we blindly apply this law: on one hand, if we leave θ_R as such, the Turtlebot will forever spin back and forth without never reaching the reference heading, as θ cannot be larger than π nor smaller than π[2]; on the other hand, if we convert θ_R to stay inside the interval $[-\pi; \pi]$, namely $\theta_R = -0.75\pi$, in other to get there, the Turtlebot will not do a "natural" movement: instead of rotating to its left, it will make a whole and not "realistic" turn to its right, rotating more than necessary. In order to tackle this problem, a conversion logic has been implemented inside the MATLAB function that is responsible for computing the feedback law in the context of the Simulink model used to run the simulation: when the heading of the Turtlebot and the reference orientation take value inside a given interval, they are simultaneously converted to get a smooth and more plausible turn.

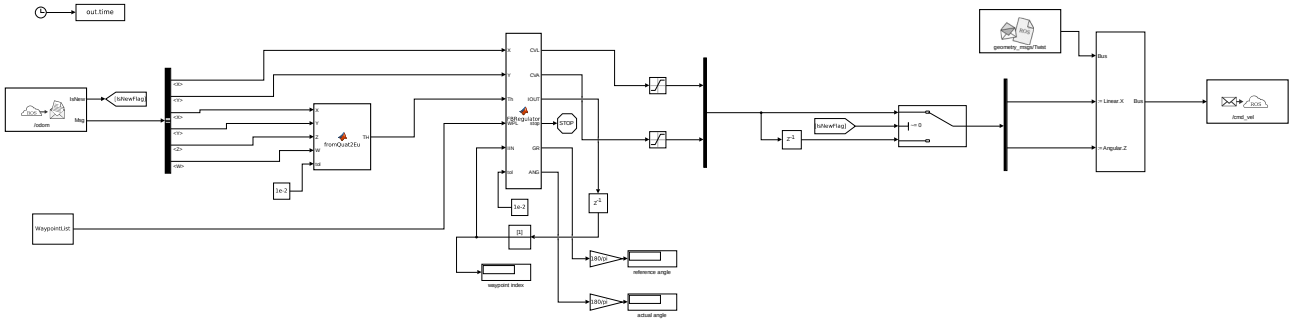

Figure 13: Simulink model used to publish /cmd_vel.

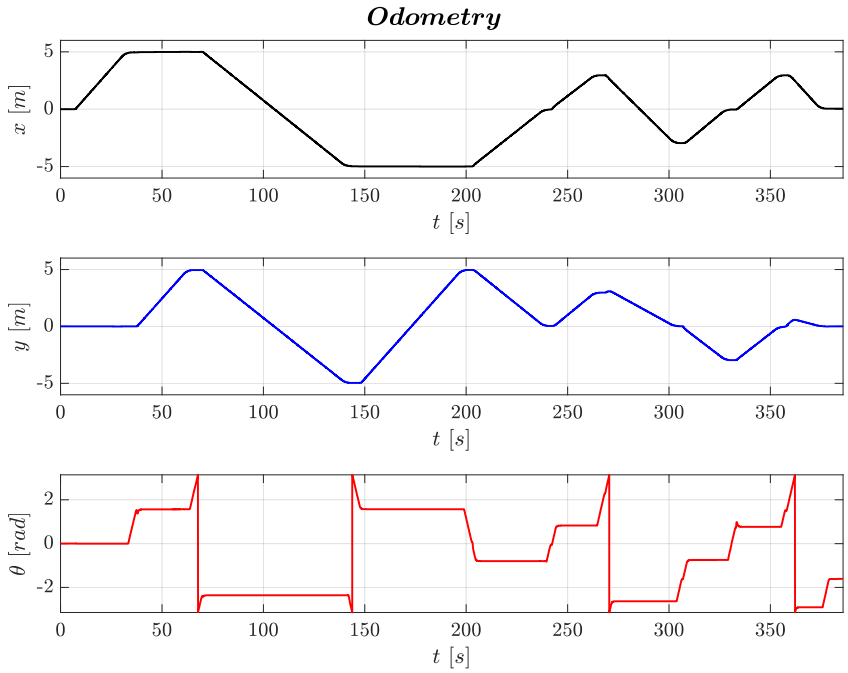

Figure 14: recorded odometry. The "jumps" in the angle are due to $\theta \in [-\pi, \pi]$

2 This happens because when the heading of the Turtlebot crosses π and it becomes negative, because it contemporaneously crosses $-\pi$. Thus, in order to reach 1.25π, the Turtlebot will revolve until π is reached and the process starts again, lasting forever.

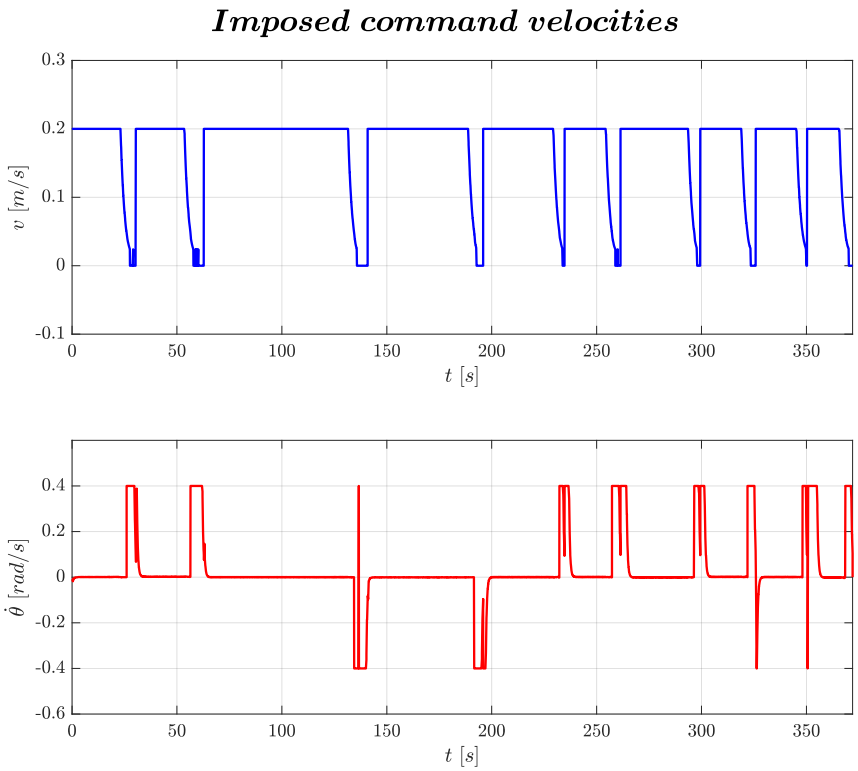

Figure 15: imposed command velocities.

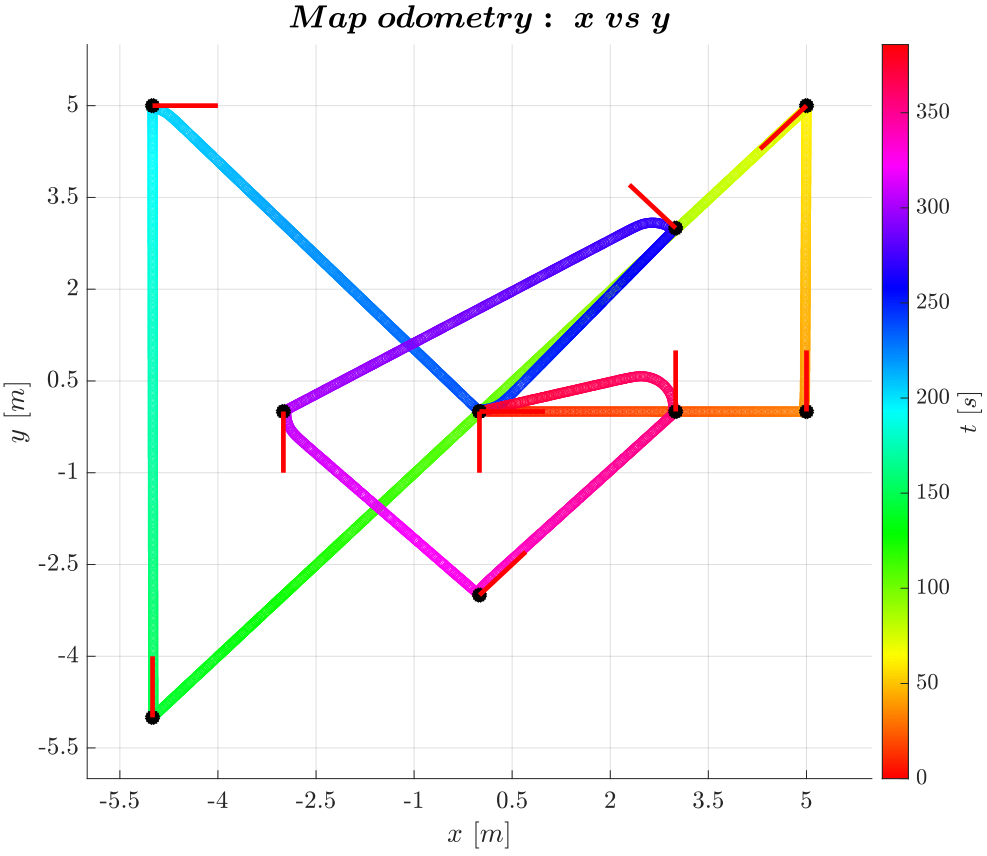

Figure 16: map plot of the trajectory. In black the waypoints, while the red bars indicate the waypoint's orientation.

As it can be inferred from figure 16, the waypoints have been reached with sufficient accuracy and Turtlebot's behavior in the curves is acceptable. From figure 14 we can highlight that probably there are some points where the Turtlebot slows down unnecessarily, but this is due to the fact that a general feedback control law has been selected, which isn't altered to consider specific transitioning just for the sake of brevity. In order to reduce the time losses in these passes, modifications can be applied to the piece of code which is responsible for the calculation of the control law, such as a flexible policy on the stopping criterion, i.e. just slow down when the curve is not very tight, and even implementing a "forecasting" approach, namely looking at the next waypoint when approaching to the current one, in order to have a better transition. These aspects are of relevant interest, but are out of the scope of the present assignment and they were just considered as possible improvements.

An interesting attempt that was made consisted of eliminating the stopping criterion at the waypoint.

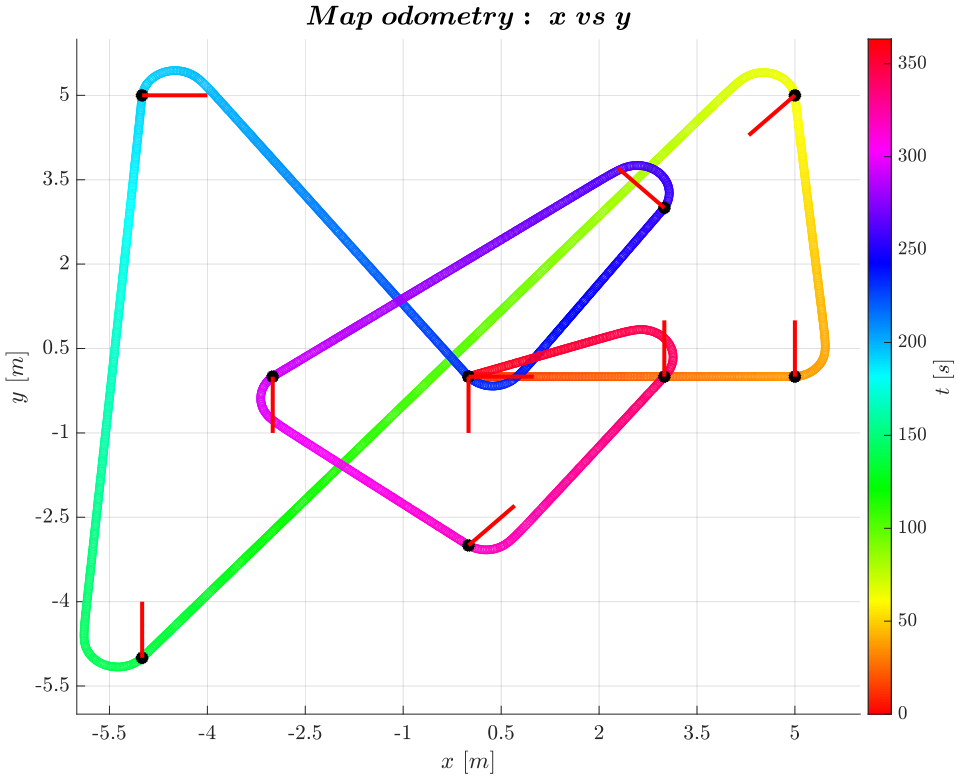

Figure 17: "fast" attempt. No stopping criterion at waypoints.

As we can see from the colorbar, this attempt is slightly faster, at the cost of missing the orientation requirement of the waypoint.

For details about the content of the Matlab function handling the calculation of the feedback law, the reader is suggested to check the github repository.

The bonus request of the present assignment asks to create a publisher that outputs the waypoints' list in the ROS network inside a *PoseWithCovariance* message. It was decided to write a Python script that, executed within the ROS frame, can handle the request.

```
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()

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

In order to make the Simulink model read this message, we just have to add a subscriber and write a MATLAB function that can extract the waypoints' list from the covariance array.

3. Assignment 3

The third assignment requests us to use the Turtlebot Waffle Pi, because of the presence of the camera. In fact, we are asked to place one red ball and one pink ball in the Gazebo empty world and to leverage the camera view to implement a feedback law that lets the Turtlebot drive towards the red ball and crash against it.

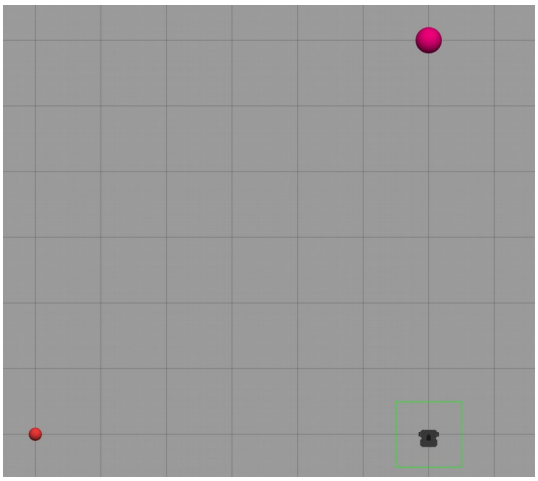

Figure 18: the Gazebo empty world with the two balls deployed on it.

The implemented strategy has been the following: the Turtlebot starts rotating in the look for the ball, when it sees the red ball, it fixes its heading in order to have it centered in the image and, with some delay it starts moving longitudinally. The implemented feedback policies, with A the area of the BLOB representing the red ball in the processed image (more on that later) and CP the position of the centroid in the image frame:

$$v = k_{P,LON} A$$

$$\omega = k_{P,ANG}\left(\frac{640}{2} - CP_x\right)$$

The law for the longitudinal velocity could have been "inverted", namely making it depend on the inverse of the area, to start faster and then slow down, but this could have lead to some wobbling phenomenon. In fact, a proper tuning of the feedback law has been performed in terms of gains and the initial delay for starting the longitudinal motion when the red ball is first spotted.
A synergistic approach was followed in order to reduce the wobbling behavior: if the Turtlebot starts immediately as the ball is first seen, the ball will no longer be in the center of the image, but it will shift to its left; thus, the error associated to ω will increase and the Turtlebot will push with the angular velocity to recenter the ball in the middle of the image. If the effects of the two agents are too large, namely if the gains are too big and the resulting actions in terms of v and ω are too strong, the Turtlebot will start wobbling because when it will actuate for the first time to reposition the ball in the center of its view, it will push too much, thus the ball will be shifted too much now to the right of the image and the process will last until the final crash. Thus, the gains were reduced a lot to avoid this phenomenon, and the reason behind the peculiar choice for the longitudinal velocity feedback policy is the same.
As previously mentioned, another countermeasure was leveraged: when the Turtlebot sees the ball for the first time, it doesn't immediately starts moving longitudinally, but it first fixes its heading to have the ball in the center of the image, in order to eliminate the wobbling problem. From the implementation

perspective, this is possible thanks to a variable that is outputted when the Turtlebot first sees the ball, which is seen by the controller of the longitudinal velocity only after a 10 steps delay.

Looking at the Simulink model, we first encounter the block responsible for the extraction of the image from the ROS network. The variable *IsNewFlag* will be used to avoid inputting a new signal when no new image is received.

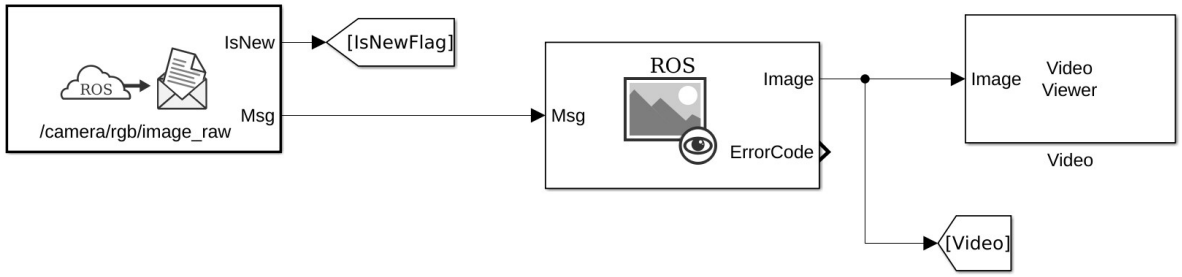

Figure 19: image extraction from ROS network.

The variable *Video* goes straight to the block which accounts for the translation of the RGB image to a binary representation, passing through some processing procedure.

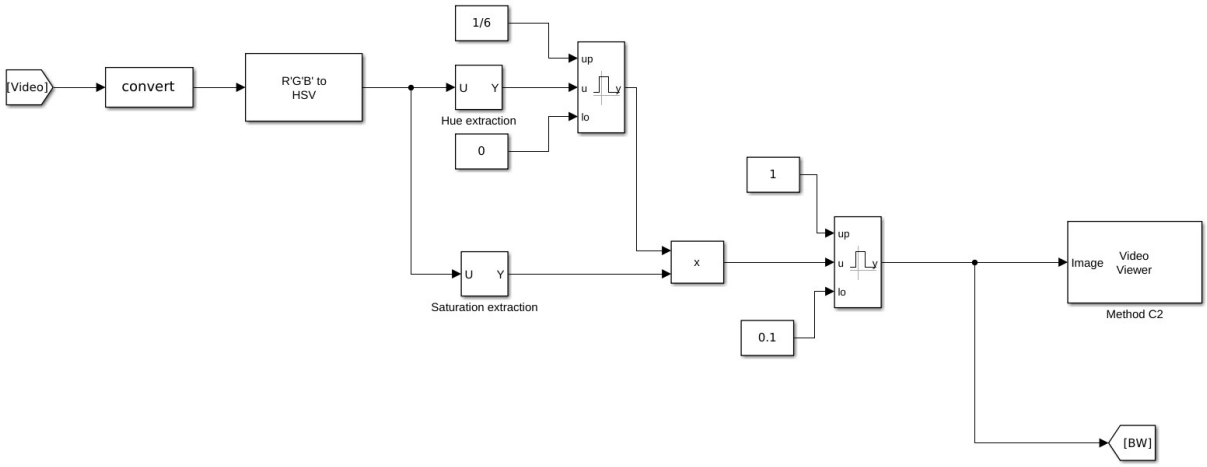

Figure 20: image binarization.

A very good aid for color recognition of images is the HSV representation, which is closer to how the human eye works with respect to the RGB one.
From color theory we know that red lies in the hue interval *[0; 60°]*, which after normalization becomes *[1; 1/6]*.
The ball we look for is red, while the floor and the sky of the Gazebo world are two different shades of gray. Unfortunately, the conventions used for this color representation assigns a hue equal to zero to gray and not only to red.
If we were seeking whichever color different from red, we could've just relied on hue, but here we are are mandated to leverage the saturation, because the saturation for red is way higher than the one for gray.
Thus, we impose a matrix multiplication to get a "saturated" hue image, from which only the high values are exctracted in a Boolean fashion, that identifies the red color, resulting in a binary image, represented by the variable *BW*.

For what it concerns the command velocity publication, the scheme is the following.

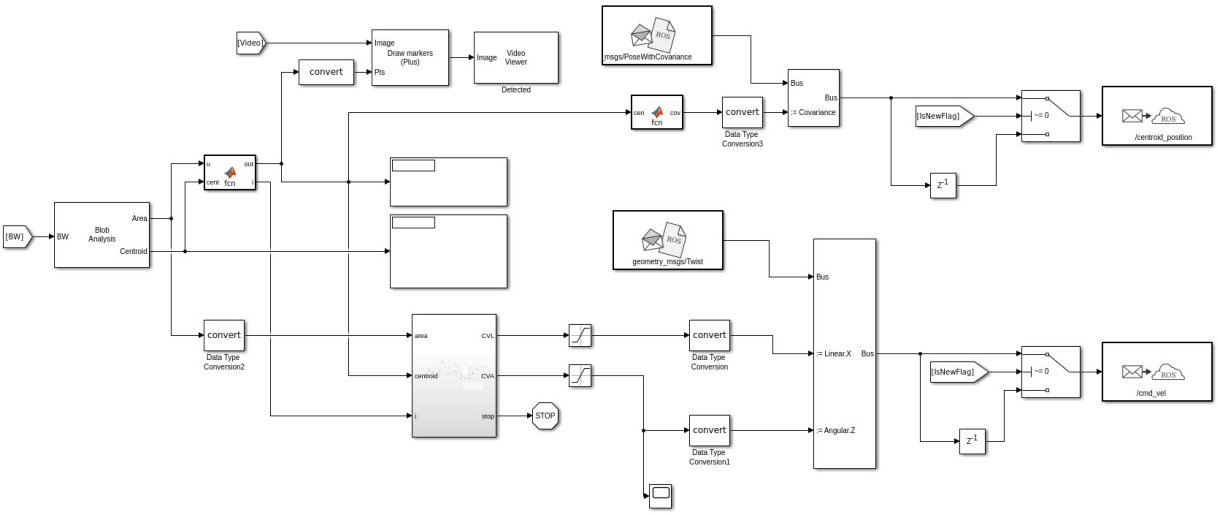

Figure 21: command velocity and BLOB centroid position publication block.

In the following figure what's inside the gray subsystem.

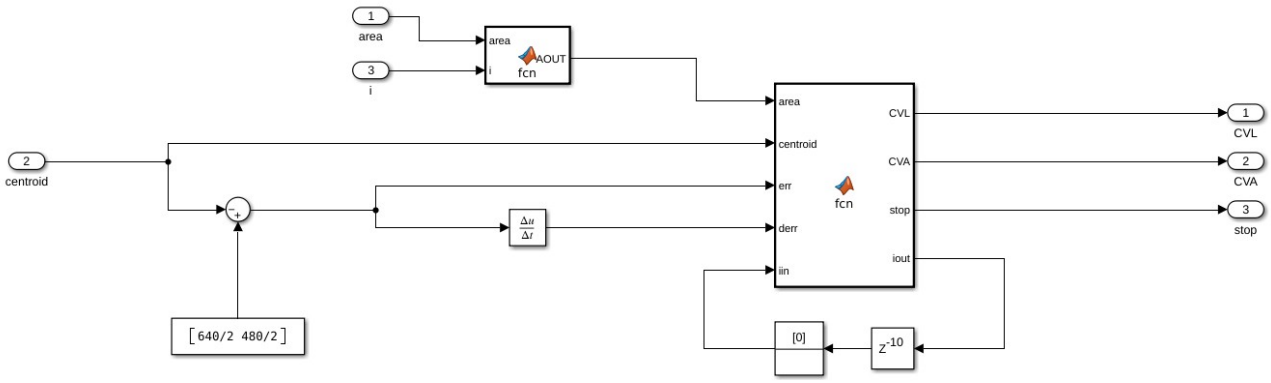

Figure 22: feedback law's computation subsystem.

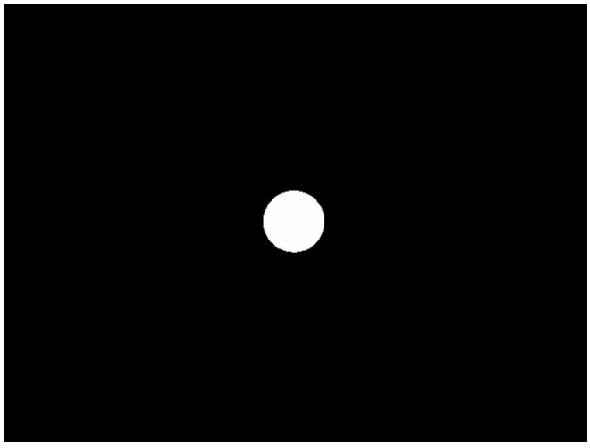

Figure 23: the red ball represented as a BLOB.

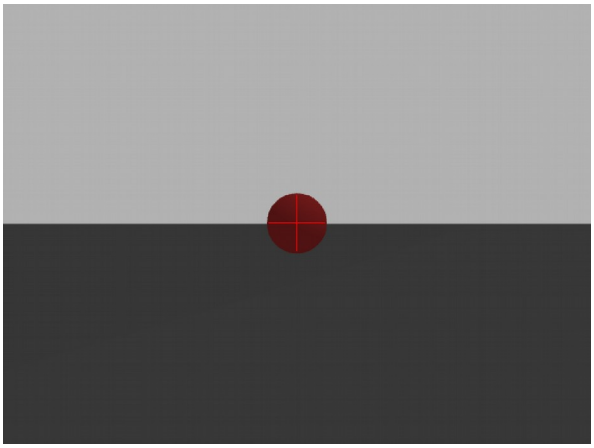

Figure 24: the recognized red ball, highlighted with a marker.

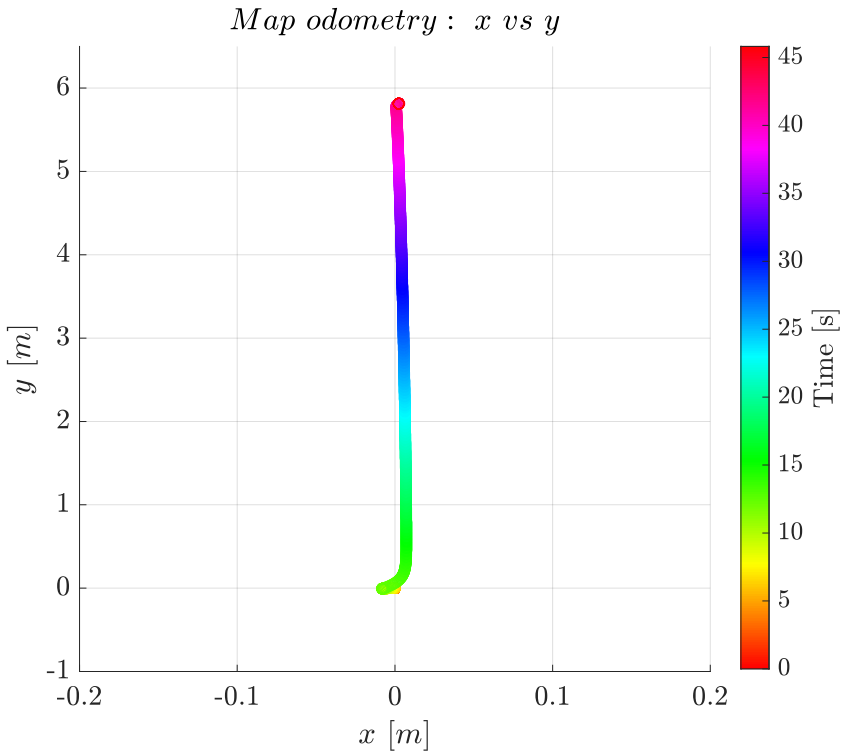

Figure 25: movement of the Turtlebot going toward the ball. The final drift is due to the crash.

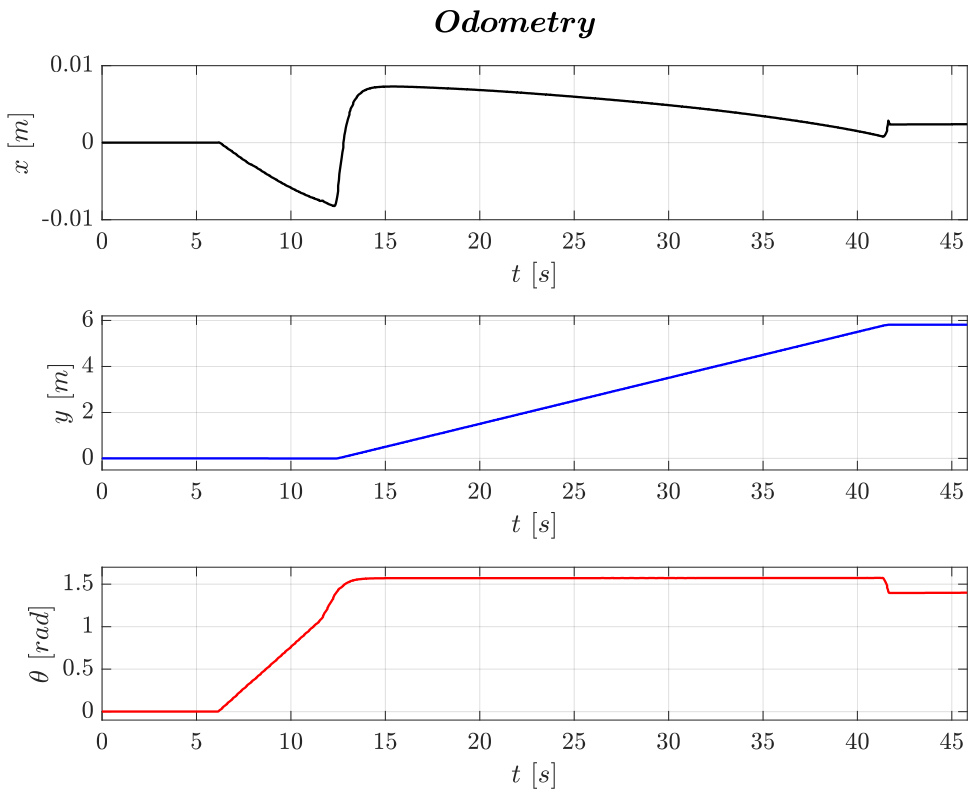

Figure 26: odometry comparison.

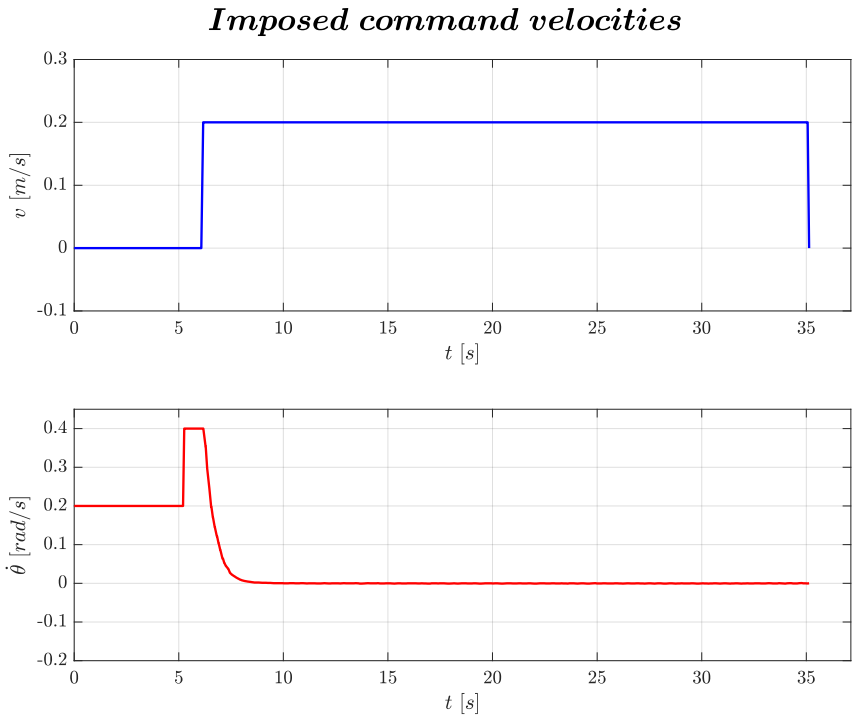

Figure 27: plot of imposed command velocities.

From figure 25, 26 and 27 is clear that the wobbling movement is completely eliminated.

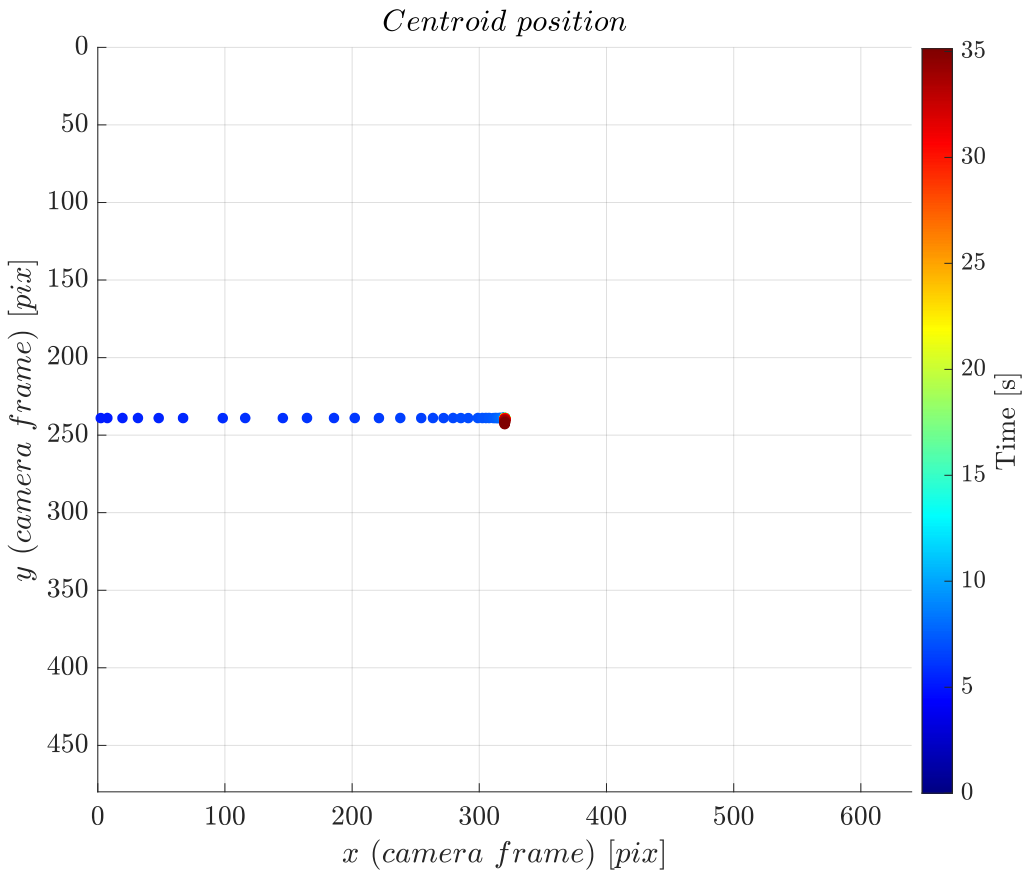

Figure 28: evolving position of the centroid during the movement.

4. Assignment 4

The fourth assignment asks us to modify a provided MATLAB script that implements a simplified Dijkstra algorithm to find the best route from a starting position to a final goal in a set of four provided images, which are binarized to graphs (0 is free, 1 is occupied).
The provided script can only consider vertical or horizontal movements, but not diagonal ones, which we are asked to consider in our code. The last request is to further modify the algorithm to make it become an A* graph-search method.
In order to be consistent with the requests, some inefficiencies of the code weren't addressed: one of those is the dependencies matrix $\boldsymbol{G}$, which is to be read as such: the entry G_{ij} is equal to 1 if from the *i-th* node a path can go to the *j-th* node and equal to 0 otherwise. If the image is square NxN, the graph contains N^2 nodes, thus $\boldsymbol{G}$ has size $N^2 x\ N^2$, which makes the memory use for saving $\boldsymbol{G}$ beyond any rational boundary if a square image has as little as 300 pixels per side. This limitations were solved in the context of the solution to the *Final Project 2*.
Let's first tackle the modification of the algorithm to include diagonal movements.
This modification holds for both Dijkstra and A* algorithms.
The code was modified in two points: first, when the $\boldsymbol{G}$ matrix is assembled and, then, when the potential traveling costs are updated.

```
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)
            %      A
            %     \|/
            % L ---+--- R
            %     /|\
            %      B

            BCon = ii + 1 > 0 && ii + 1 <= size(BW, 1) && BW(ii + 1, jj) == 1;
            ACon = ii - 1 > 0 && ii - 1 <= size(BW, 1) && BW(ii - 1, jj) == 1;
            RCon = jj + 1 > 0 && jj + 1 <= size(BW, 2) && BW(ii, jj + 1) == 1;
            LCon = jj - 1 > 0 && jj - 1 <= size(BW, 2) && BW(ii, jj - 1) == 1;

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

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

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

            % left
            if LCon
                G((ii - 1)*size(BW, 1) + jj, (ii - 1)*size(BW, 1) + jj - 1) = 1;
            end
```

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

end
```

As it can be seen, some new exploring conditions had to be implemented to look in the image and see if a node is reachable from another one, namely if G_{ij} is 1 or 0.

Furthermore, the G_{ij} entry for a diagonal connection is square root of 2, to recognize it from the horizontal and vertical displacements, highlighted by 1.

For the second modification to apply, we have to distinguish between the cases of Dijkstra and A*.

The main difference lies in the fact that the A* star algorithm has to account for the so-called *cost-to-go*, i.e. an underestimate of the cost to go from the considered node to the goal – the euclidean distance between them, in our case -, in order to discourage the research in regions of the graph far from the goal.

```
switch AssignmentCase
            case 'DijkstraBaseCase'
                    if dist(con_nodes(i_con)) > dist(cur_node) + 1

                    % if not measured or shorter
                        dist(con_nodes(i_con)) = dist(cur_node) + 1;
                    % calc dist for the new node
                        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
```

Modifying this part was relatively easy, as it was just a matter of adding a cost different from 1 if a diagonal path is taken into consideration.
When it comes to the implementation of the A* algorithm, the only necessary addition is as easy as adding the cost-to-go to evaluate if a transition to a certain node is convenient in term of distance from the goal.

Let's look at the first image that has been analysed.

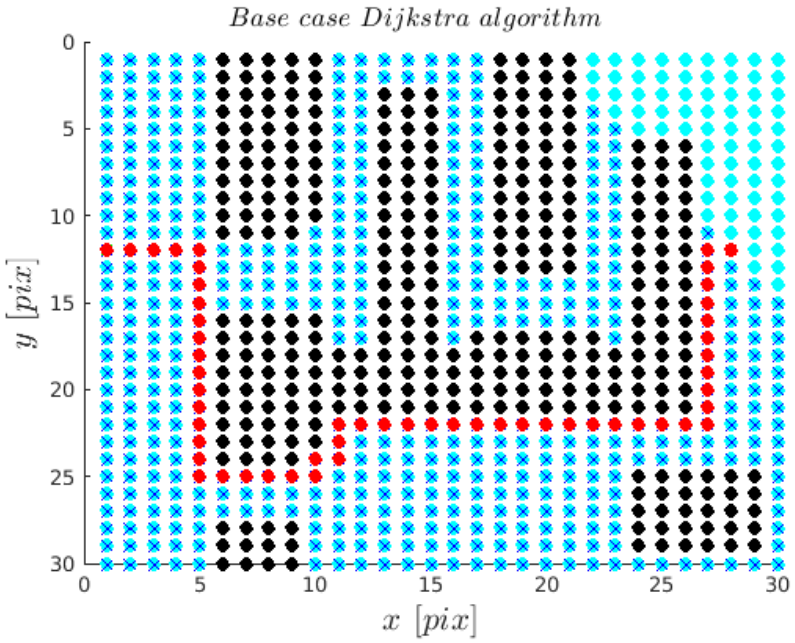

Figure 29: base case Dijkstra algorithm.

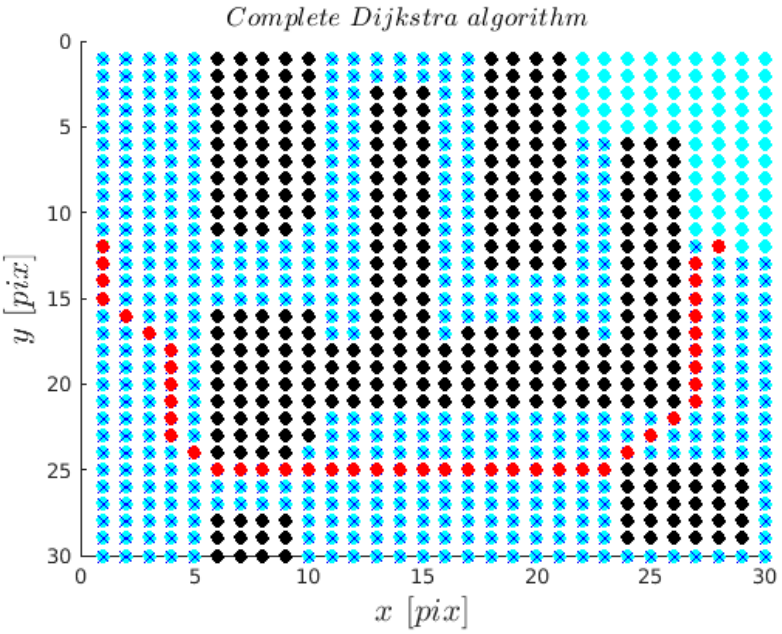

Figure 30: complete Dijkstra algorithm, with diagonal movements.

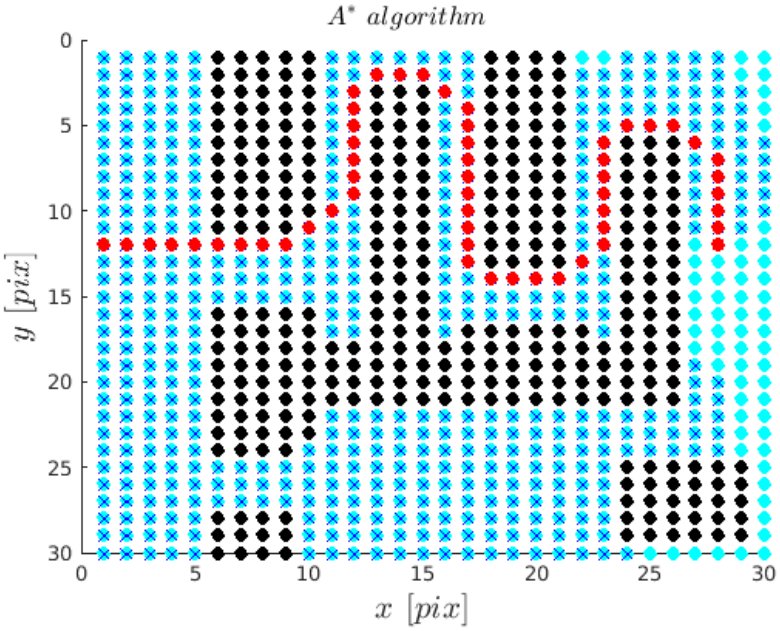

Figure 31: A algorithm.*

The nodes that have been analyzed are highlighted in the figures 29, 30 and 31 as light blue with crosses superimposed.

Case	Numbers of analyzed nodes	Path length [nodes]
Base case Dijkstra	494	54
Complete Dijkstra	493	44
A*	518	54

Table 1: numbers of analyzed nodes between the three cases. First map.

We can see that A* seems to perform worse than the other two algorithms, because of a higher analysis burden and a longer path length, but this is due to the specific shape of the considered map.
In fact, if we look at another map, the actual strength of the A* algorithm pops up.

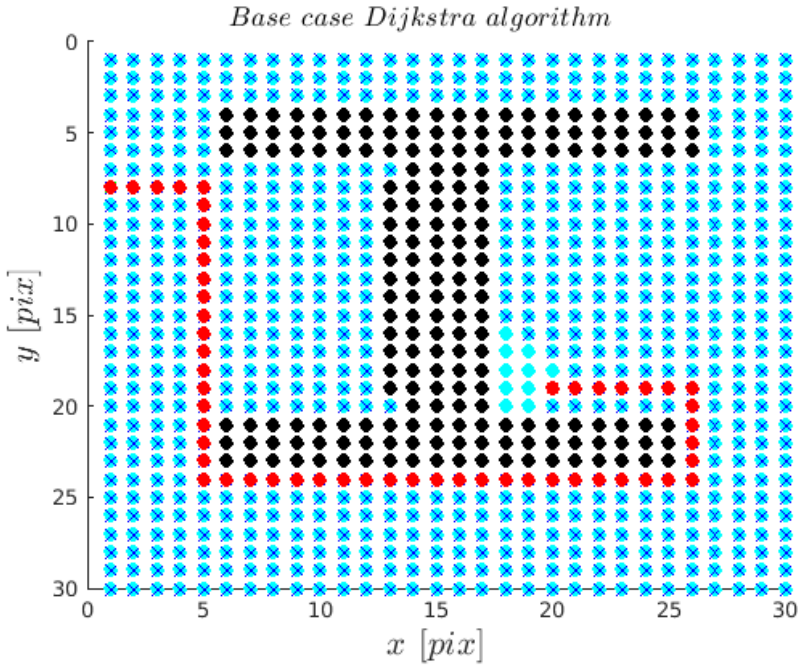

Figure 32: base case Dijkstra algorithm.

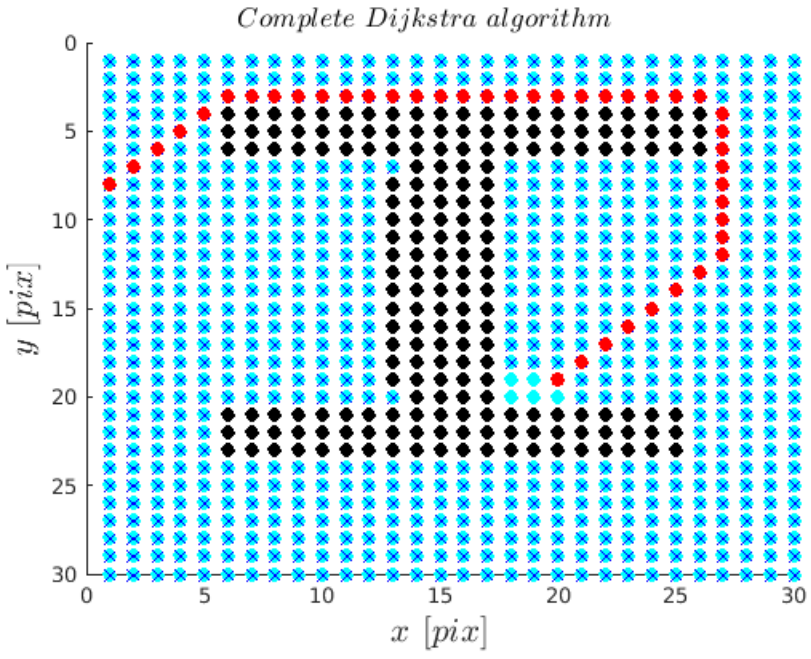

Figure 33: complete Dijkstra algorithm, with diagonal movements.

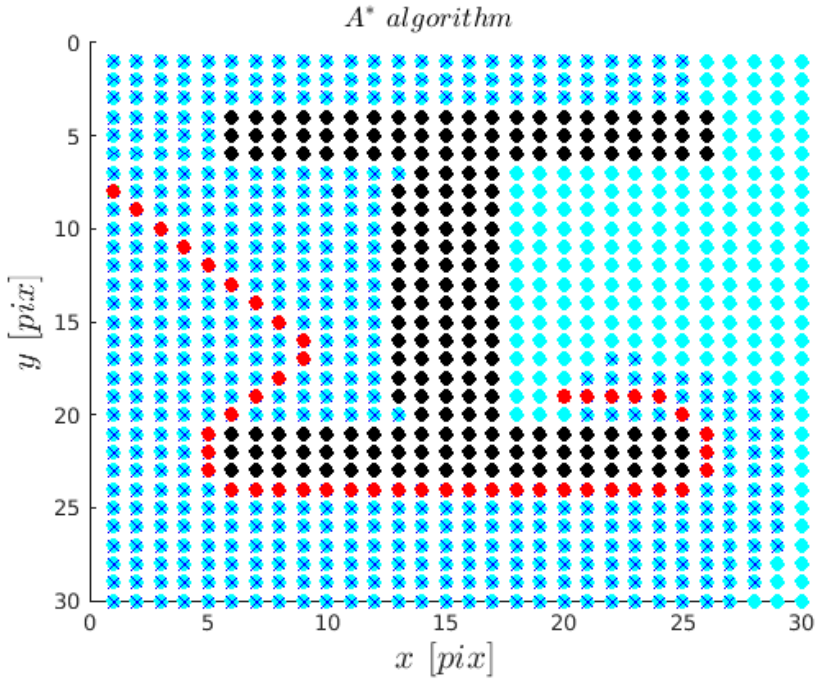

Figure 34: A algorithm.*

Case	Numbers of analyzed nodes	Path length [nodes]
Base case Dijkstra	703	53
Complete Dijkstra	707	42
A*	546	45

Table 2: numbers of analyzed nodes between the three cases. Second map.

As it can be appreciated by the plots and the table, with this kind of map, the A* algorithm clearly performs better. It is also to point that a longer path length is achieved for A*, but managing to drastically decrease the number of needed explorations. The lengths of the path between the complete Dijkstra and the A* algorithms are not completely comparable, because for the latter one the distance between the goal and the node is considered and, thus, some displacements are followed, that the Dijkstra algorithm would discard.

Let's see the exploration of the third map.

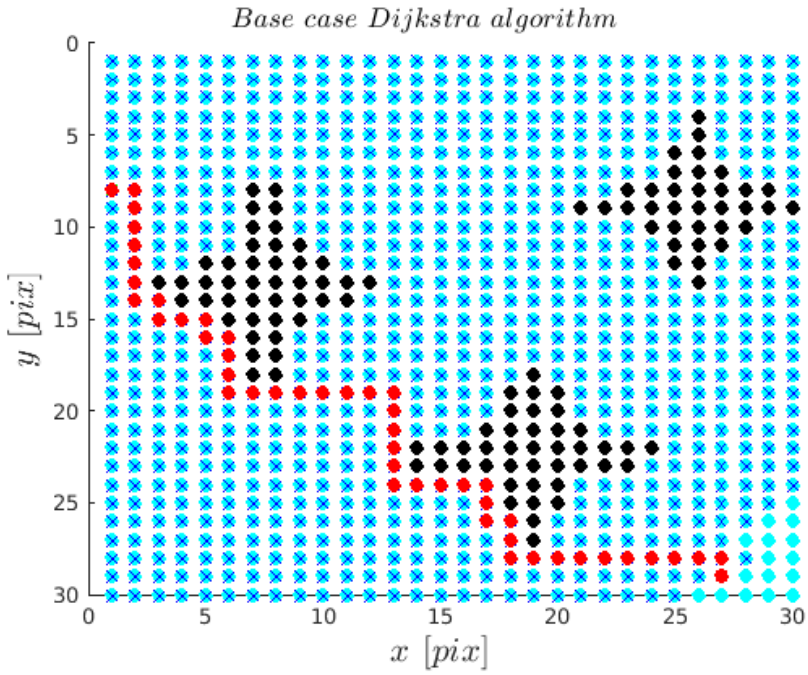

Figure 35: base case Dijkstra algorithm.

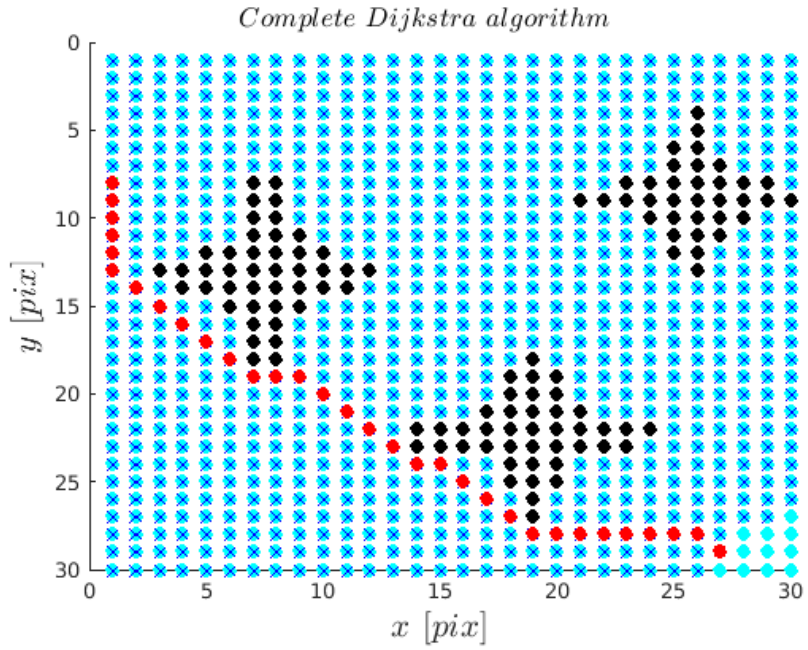

Figure 36: complete Dijkstra algorithm, with diagonal movements.

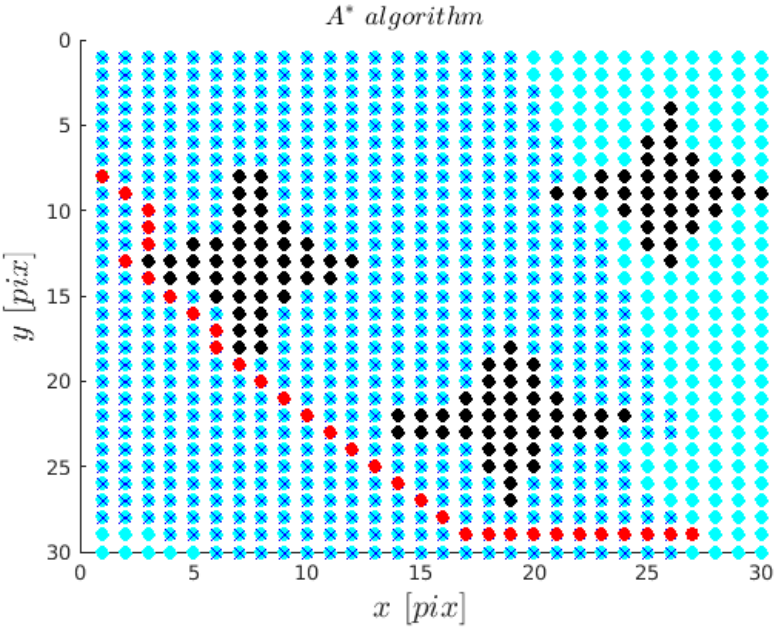

Figure 37: A algorithm.*

Case	Numbers of analyzed nodes	Path length [nodes]
Base case Dijkstra	771	48
Complete Dijkstra	776	32
A*	646	32

Table 3: numbers of analyzed nodes between the three cases. Third map.

In this case, the advantage of using the A* algorithm over the other two is confirmed.
The same path length between Dijkstra and A* witnesses once again that the differences in the performances of the algorithms are strongly map-dependent.

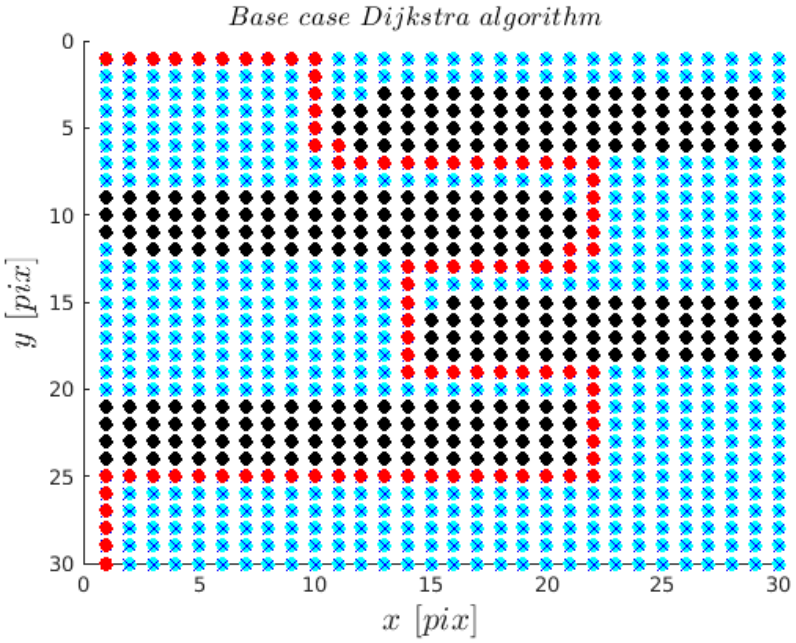

Figure 38: base case Dijkstra algorithm.

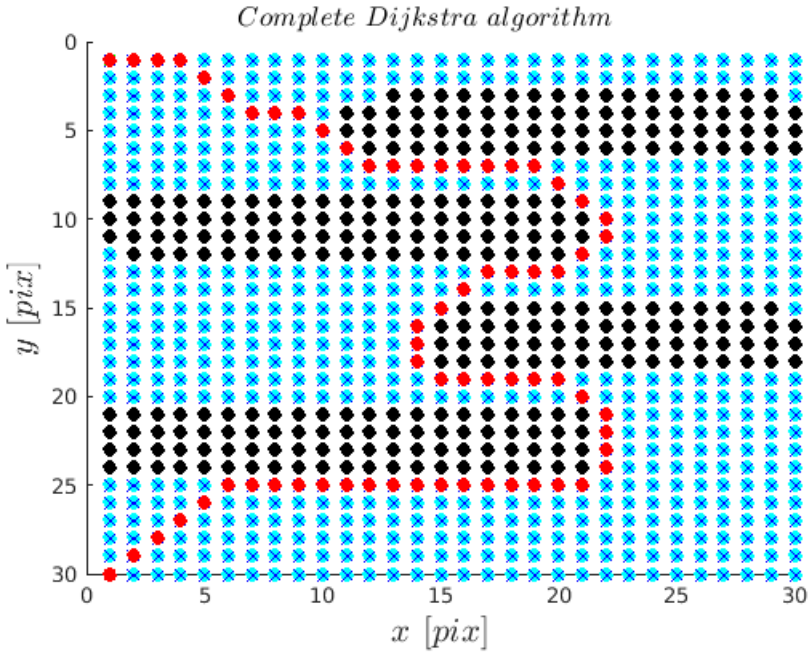

Figure 39: complete Dijkstra algorithm, with diagonal movements.

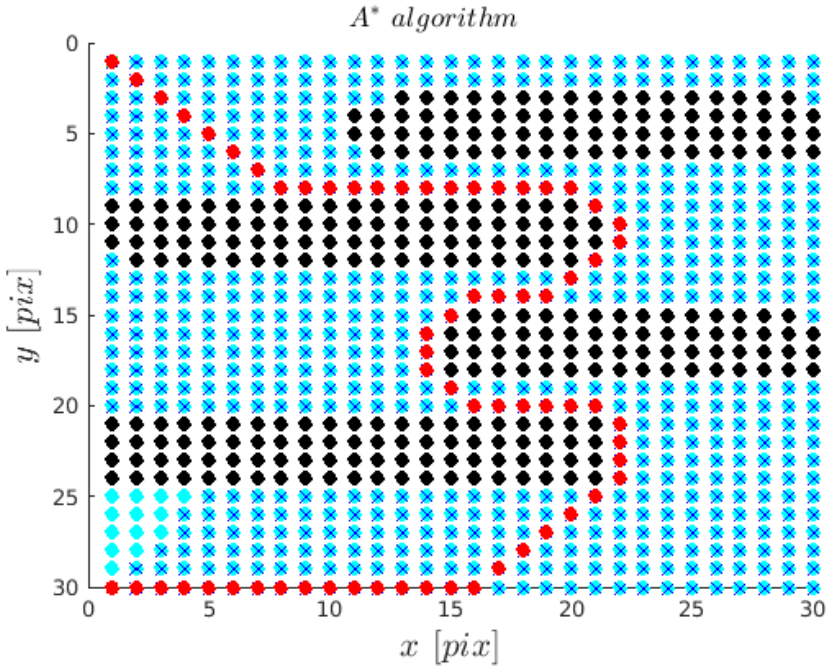

Figure 40: A algorithm.*

Case	Numbers of analyzed nodes	Path length [nodes]
Base case Dijkstra	597	88
Complete Dijkstra	597	65
A*	592	65

Table 4: numbers of analyzed nodes between the three cases. Fourth map.

Given the particular "demanding" nature of the map, the whole set of node is explored by the two Dijkstra algorithms, while the A* manages to spare some nodes.

In conclusion, even if some maps make the performances of the algorithms almost indistinguishable, the A* one proves to be the best one, thanks to its capability of discarding paths that are too far from the goal.

5. Assignment 5

The assignment asks to implement a finite state machine model of a parking gate, so that it raises when a car is waiting in front of it and it lowers when this has passed.
The movement of the car has been modeled with a finite state machine as well, based on the modification of a provided one.

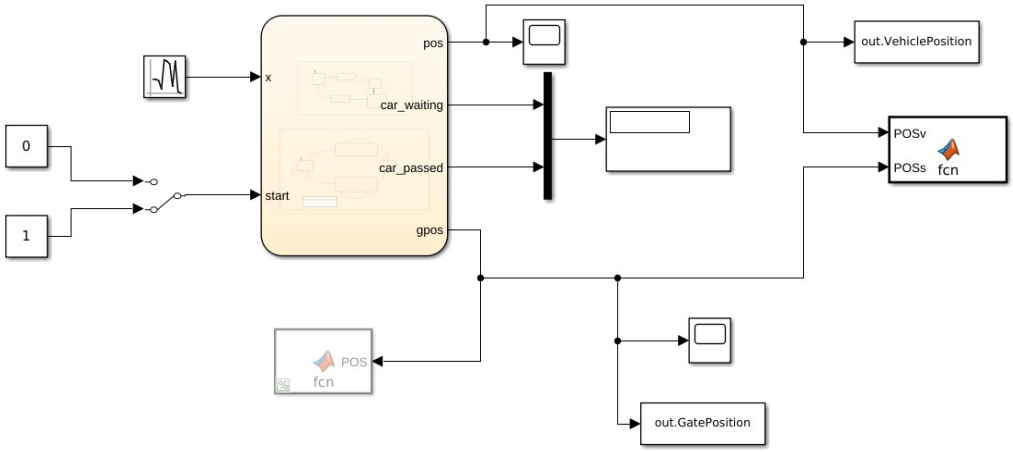

Figure 41: Simulink model of the whole system.

The real work is made inside the yellow chart, which contains two finite state machines running in parallel: one simulates the movement of the car, while the other is responsible for the control of the motion of the parking gate.

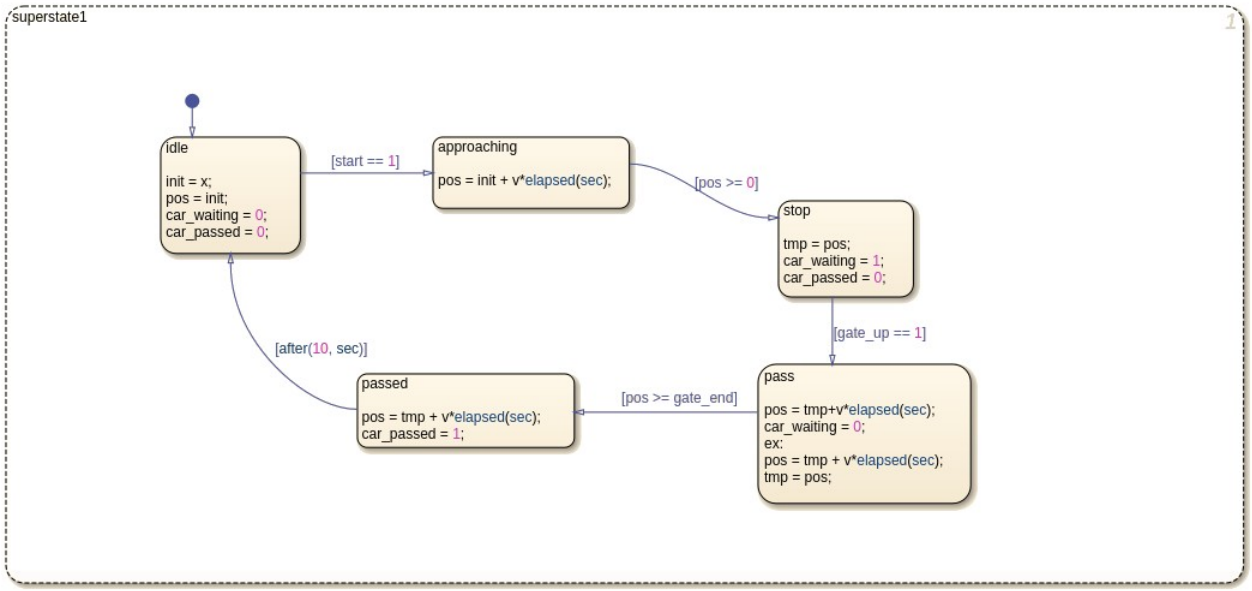

Figure 42: finite state machine modeling the car movement.

The initial position of the car is initialized by mean of a random number inputted to the system. Until the switch isn't manually triggered, namely until the *start* variable isn't equal to one, the car doesn't move. The initial position of the car is randomly negative, and it reaches zero in front of the parking gate, when it enters the waiting state, where the variable *car_waiting* equals one. Here, the gate starts raising and, when it has reached the upward position, i.e. *gate_up* is outputted by the gate's finite state machine as equal to one, the car can pass following a constant velocity profile. After a certain fixed distance of the car from the gate, the car enters the state *passed* and the gate can lower. 10 seconds after the passing of the car, another can is spawned.

It's probably more interesting to give a look to the finite state machine that describes the motion of the parking gate.

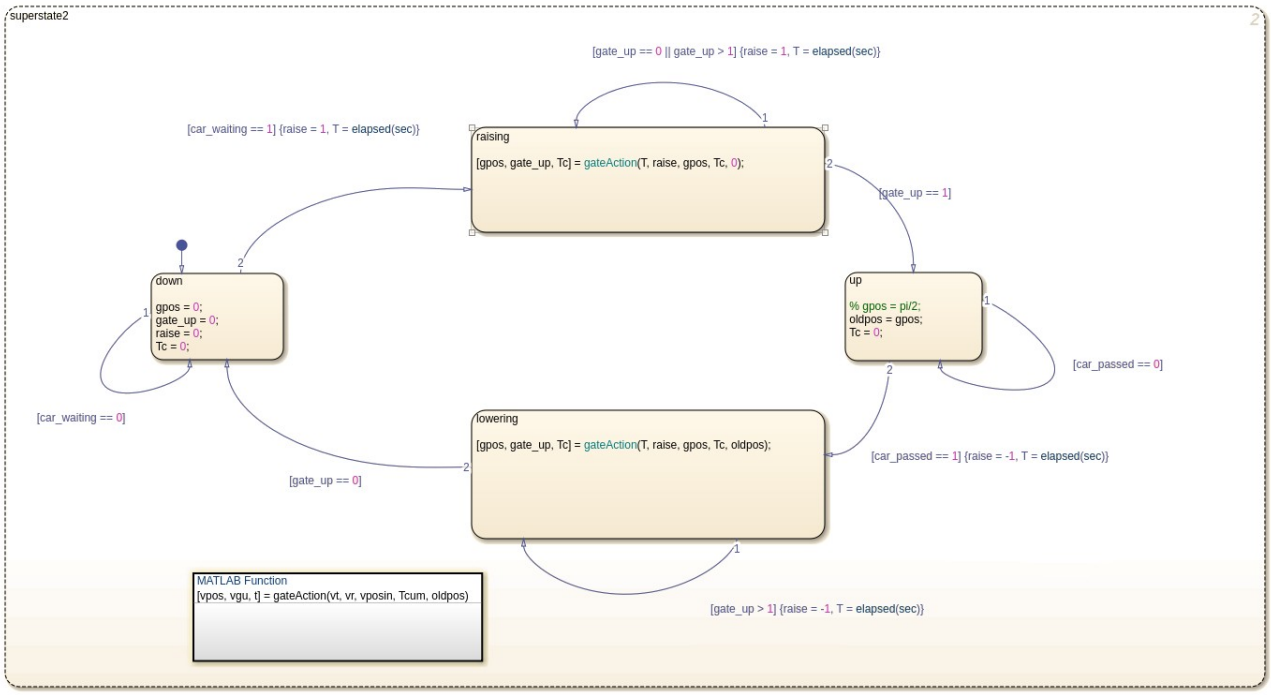

Figure 43: finite state machine modeling the gate displacement.

As there's no car waiting, the gate stays in the lowered position, but as soon as a car is before the gate, this goes in the *raising* state, where a MATLAB function does the job of moving the gate, while keeping track of the elapsed time, in order to make the gate follow a specific velocity profile in its motion. When the position of the gate is 90°, the variable *gate_up* equals one and the car can pass: the gate maintains the position until the car hasn't gone to the other side. Here, the gate enters the *lowering* state, where the same MATLAB function behaves in a similar fashion with respect to the raising case, in order to bring the gate in the resting position. When there, the cycle can begin from the start.

```
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
```

As it was anticipated, there's only one MATLAB function that operates the motion of the gate and its behavior is pretty straightforward. A simple maximum acceleration profile is imposed, where the acceleration is positive or negative if the gate has to raise or lower, respectively. An "offline" check was made in order to make sure that the maximum velocity reached is smaller than the limit one.
When the gate is in the vertical condition, the variable *gate_up* is outputted as equal to one, so that the car can pass. A tolerance is used to recognize the position of the gate with respect to the resting and upward references, because a strict identity showed unreliable behavior.

Let's now look at the plot of the position of the gate and of the car versus time.

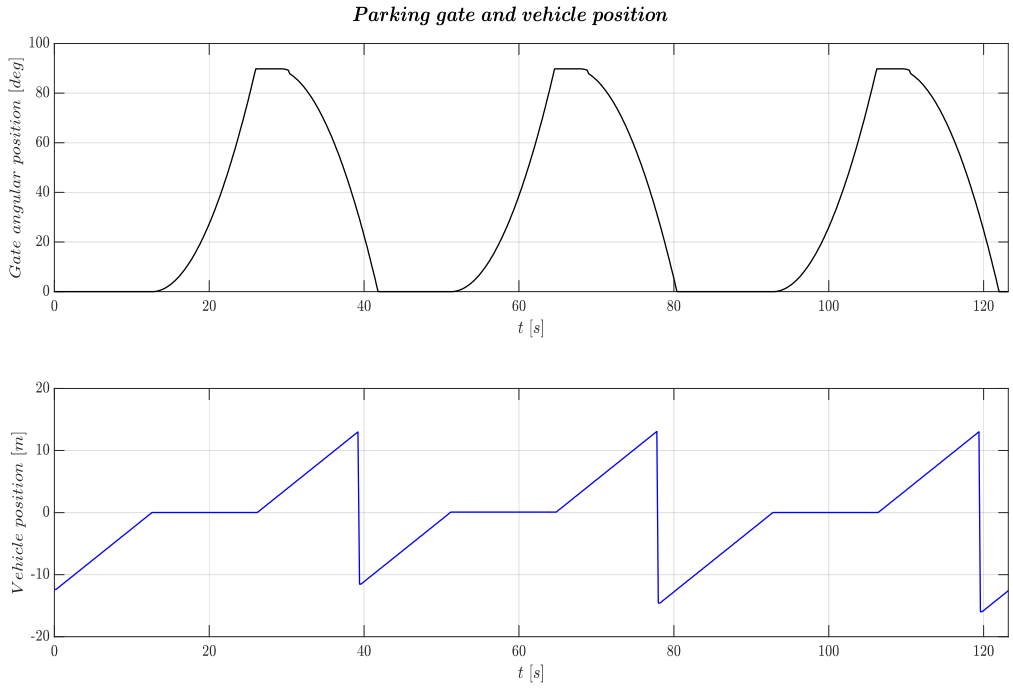

Figure 44: gate and car position in time.

A typical maximum acceleration/maximum deceleration profile is visible in the plot of the angular position of the gate, meaning that the MATLAB function properly does its job.
The discontinuity in the car position is due to the spawn of a new car, because the previous one is sufficiently far from the gate.
The interested reader is referred to car and gate animation on the github repository.

Second part: final projects

1. Final project 2

The second final project imposes three objectives:

- simulate the Turlebot inside the *Gazebo house* and use the *gmapping* tool to create a map of the environment;
- modify the A* algorithm from assignment 4 to let it analyze maps bigger than 300x300 pixel, overcoming the limitations related to the dependencies matrix G;
- apply a feedback control to move the Turtlebot.

The Turtlebot that has been used is the Waffle Pi.
The first point is pretty easy to tackle, even though a pretty high amount of time (≈ 1 hour) is requested to map such a big environment by manually operating the Turtlebot and, due to the use of standard SLAM routines without customization, an acceptable yet not perfect result was achieved, as shown in the next figure.

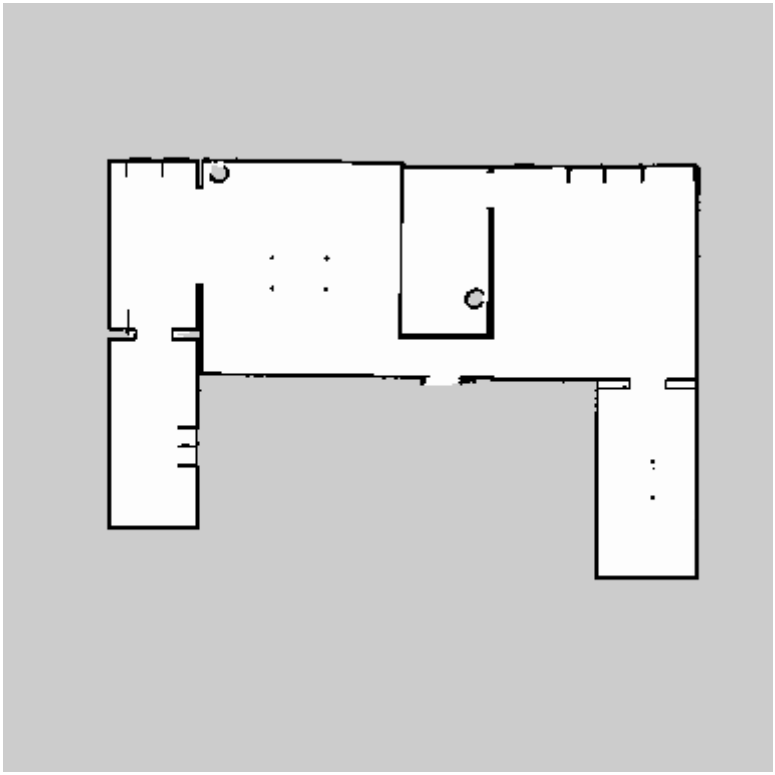

Figure 45: map obtained throught the built-in SLAM method.

An important reason for some defects to have been left inside the obtained map is related to the fact that, after a certain instant, the attempt of solving a mapping defect in some part of the environment made other problems arise in other positions of the map.
As it will be shown in the final section, these approximations are acceptable in the end, because the closed-loop trajectory is very similar to the open-loop one (computed with the A* algorithm): the main yet slight differences are imputable to these aberrations.

Before feeding the map to the algorithm, once again a binarization of it is needed: unknown space (gray) will be set to 0 as the free space (white), while the occupied one (black) will be represented by a 1.
For what it concerns the modification of the A* algorithm, as previously anticipated in the discussion of assignment 4, the provided version has to be modified, because of the highly inefficient way it's used to calculate the dependencies, i.e. the nodes that can be reached from a given one.
Remind the meaning of the dependencies matrix $\boldsymbol{G}$, which is to be read as such: the entry G_{ij} is equal to 1 if from the *i-th* node a path can go to the *j-th* node and equal to 0 otherwise. If the image is square NxN, the graph contains N^2 nodes, thus $\boldsymbol{G}$ has size $N^2 x\, N^2$, which makes the memory use for saving $\boldsymbol{G}$ beyond any rational boundary if a square image has as little as 300 pixels per side.
The solution to this problem was found in the elimination of the $\boldsymbol{G}$ matrix, because it's completely useless to store the knowledge of the nodes that can be reached from the node 3040, when the current computation of the path is considering the node 5060 and its dependencies. Thus, a dependency calculation is performed at each solving step only for the considered node and, when a new one is taken into consideration, the previous dependency information is substituted by the current one.
Thus, the same algorithm that in assignment 4 was used to assemble the $\boldsymbol{G}$ matrix is here re-implemented and "shrunk", so that a MATLAB function takes as input a free node inside the image and looks in vertical, horizontal and diagonal directions around the node to understand the dependencies.

```
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
    BCon = ny + 1 > 0 && ny + 1 <= size(BW, 1) && BW(ny + 1, nx) == 1;
    ACon = ny - 1 > 0 && ny - 1 <= size(BW, 1) && BW(ny - 1, nx) == 1;
    RCon = nx + 1 > 0 && nx + 1 <= size(BW, 2) && BW(ny, nx + 1) == 1;
    LCon = nx - 1 > 0 && nx - 1 <= size(BW, 2) && BW(ny, nx - 1) == 1;

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

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

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

    % left
    if LCon
        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);
```

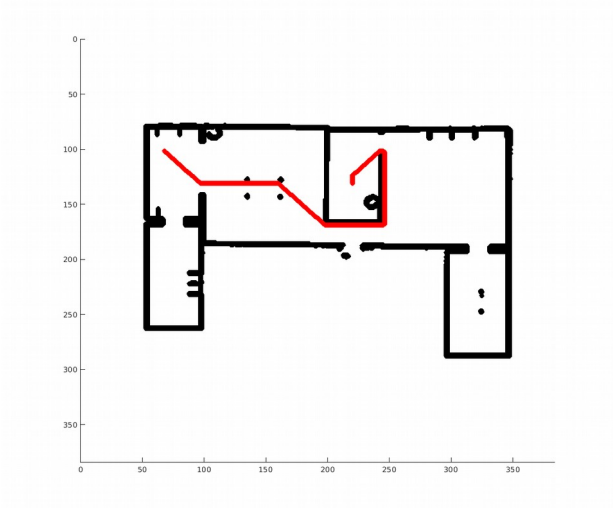

Figure 46: first path analysed, no enlargement.

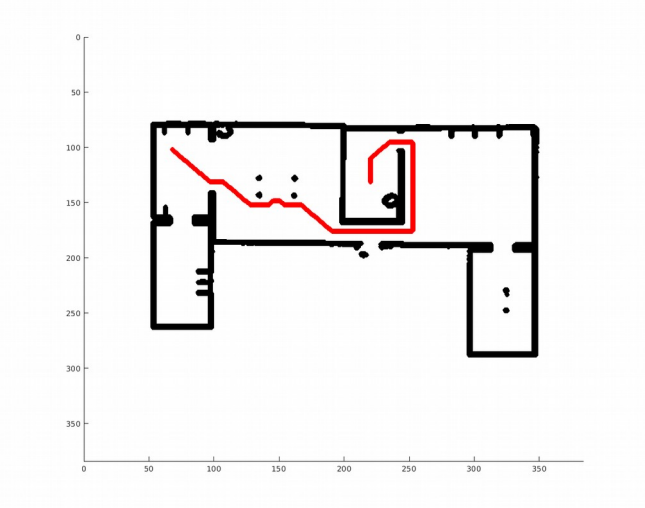

Figure 47: first path, with enlargement.

After performing this important modification to analyze big size images, 4 different starting and goal positions have been fed to the algorithm to find the best path to traverse the house.
As it can be clearly seen from figure 46, due to its intrinsic codification, the algorithm doesn't know the dimensions of the Turtlebot: therefore, the defined path is impossibly close to the wall. If we asked the Turtlebot to follow this trajectory, it will for sure jam into the wall. Thus, an obstacle enlargement method was used, thanks to the built-in MATLAB function *imdilate*, the symmetrical geometry of the Turtlebot and a safety coefficient. In figure 47 the beneficial effect of the wall enlargement can be spotted right away: we can now ask the Turtlebot to follow the hereby defined trajectory.
Using the "wall enlargement" technique makes use save quite an important amount of time, too: without wall enlargement, the calculation time is 18.086642 seconds, while only 11.120731 seconds with wall enlargement.
For the sake of brevity and because of the aforementioned reasons, for the other three paths only the ones in the wall-enlarged map are here presented.

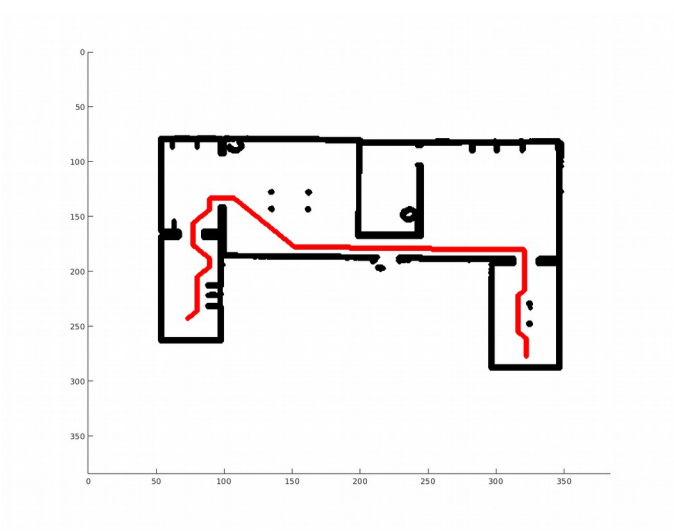

Figure 49: second path.

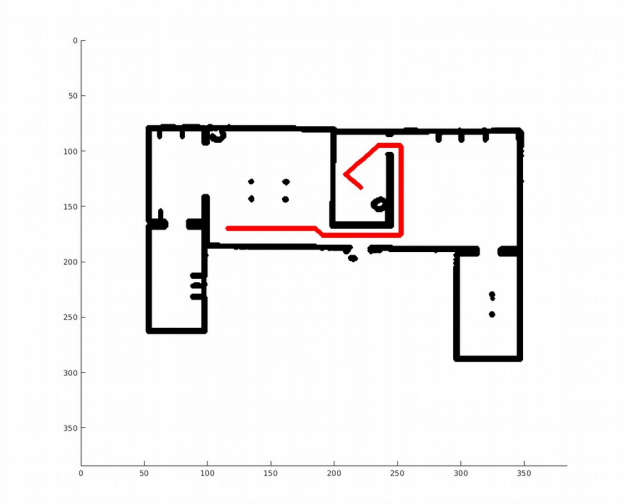

Figure 48: third path.

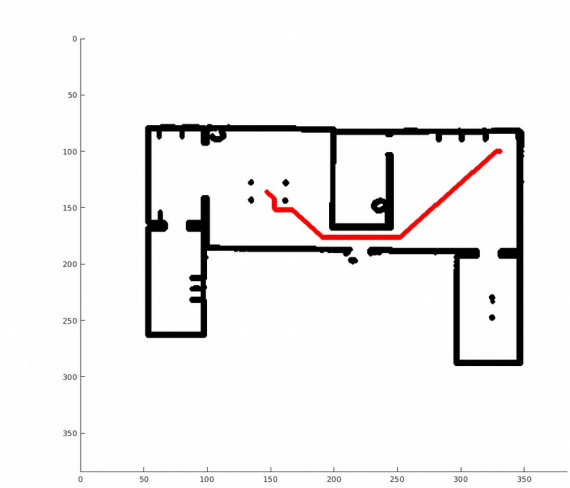

Figure 50: fourth path.

It's time to give a look to the Simulink model that is responsible for the publishing of the velocity commands to move the Turtlebot.
Before doing that, we have to address two important issues. First, the actor that is taking care of the localization of the Turtlebot inside the provided map: the ROS Navigation Stack. The Navigation Stack is a complex set of tools that is run through Rviz, and the specific one used for this project is the Augmented Monte Carlo Localization. The useful outcome of the Navigation Stack is the position and orientation of the odometry frame in the map one, that is considered as the fixed reference. This is information is published as a message in the */tf* ROS topic and, due to some unknown Simulink problem[3], a Python listener was written in order to access this useful piece of information and publish it inside the *Covariance* field of a *PoseWithCovariance* message:

```
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, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
        pub.publish(msg)
        rate.sleep()

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

This node will be run alongside of the Gazebo world, the Rviz instance and the Simulink model, that will read from this published topic the relationship between the child frame, odometry, and the parent one, map.
The second issue we have to tackle concerns the path: in order to make the Turtlebot follow the path, a feedback control law in a similar fashion as assignment 2 was used, even though with some major modifications. It's clear that we cannot feed that much amount of points that can be seen from figure 47 to 50: first, conflicts could possibly arise for crossed points that are not detected and such; second, it's completely useless to use as a list of waypoints to follow a set of points on a straight line.
In order to extract the meaningful waypoints from the A*-found trajectory, a semi-automatic method (implying some minor modifications have to be performed by hand) is implement, which consists of calculating numerically the second derivative of the path seen as a function of both x and y direction,

3 Any attempt to use a subscriber block in the Simulink model taking care of the computation and publication of the command velocities failed for unknown reasons: the only */tf* messages that were accessible belonged to the links between the base and the wheel.

so that only the points were the two derivatives are bigger than a threshold are considered meaningful, because there's a turn there. For brevity the code isn't reported: the interested reader is referred to the github repository.
The result of this crucial procedure is reported in the following figure.

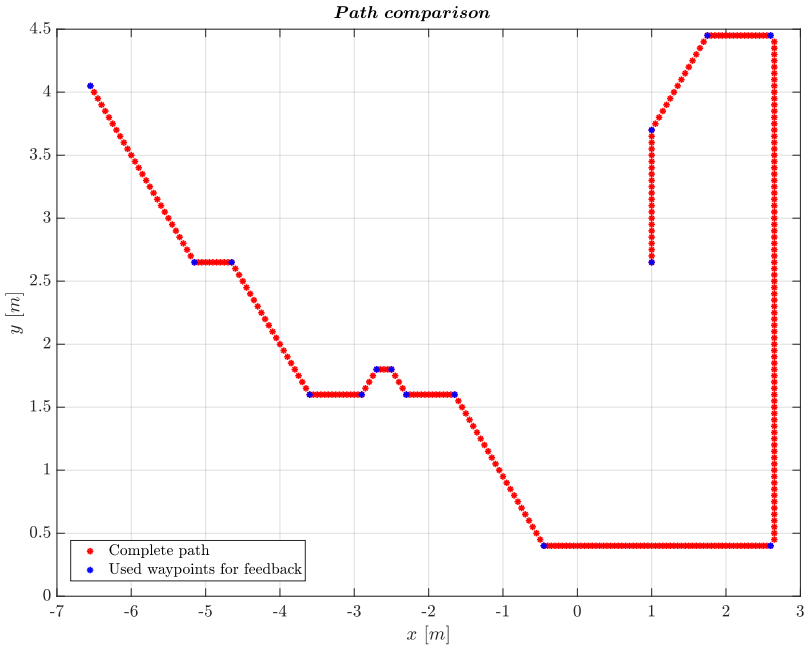

Figure 51: waypoints to publish extraction.

The Simulink model that does all the computations and publishes the command velocities is hereafter reported.

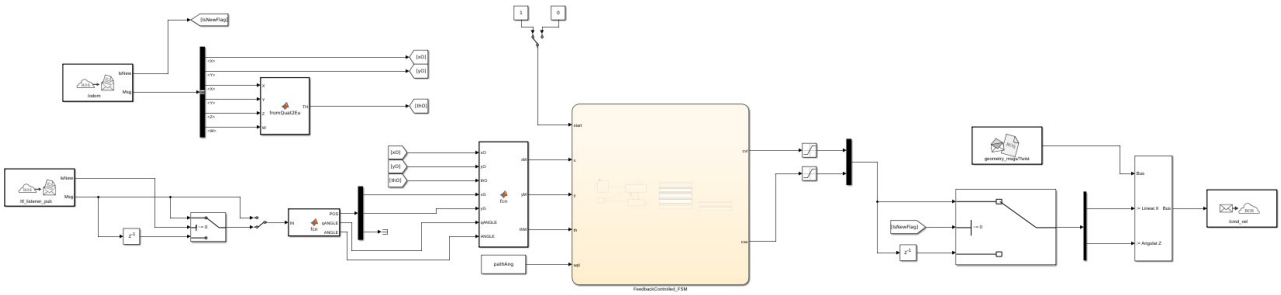

Figure 52: Simulink model.

A finite state machine (FSM) has been used to properly implement the feedback control. What lies outside the FSM is very similar to what was there in the Simulink model for assignment 2, barring from the role here played by the */tf_listener_pub* subscriber that reads the change of coordinates from the map frame to the odometry one. This information is leveraged in the very last MATLAB function positioned before the FSM, where the coordinates in the odometry frame are converted to the coordinates in the map frame, in order to compare the position of the Turtlebot in the map with the waypoints of the path, which were previously converted from pixels to meters, knowing where Rviz puts the origin of the map frame with respect to the origin of the "canonical" image reference frame, which is upper left corner.

The functioning of the FSM is pretty straightforward for what it concerns the transitions between the different states, but what it's important lies in the function *cvlCalc*, which takes care of the computation of the longitudinal command velocity, as the name suggests. The other two functions that are used are *disterrfun*, which calculates the errors in Cartesian, ***Dist***, and angular position, ***errTh***, in the same fashion as assignment 2 (the angular reference is the angle of the distance vector from the Turtlebot to the waypoint when far from it and the angle of the waypoint when close), and *cvaCalc*, which is a trivial way to modulate the proportional gain for the angular velocity (it pushes more, when the angular error is bigger than a threshold).

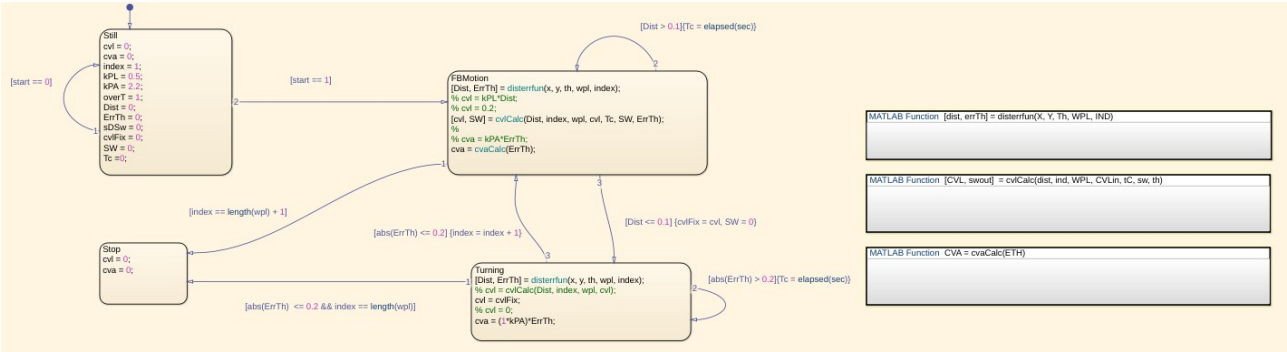

Figure 53: finite state machine controlling the velocity output.

As said, the *disterrfun* MATLAB function is used to calculate the distance from the waypoint and an angular error and, given its simplicity and description of its behavior in the discussion of assignment 2, it's not reported. We report the core of the new feedback policy here implemented, *cvlCalc*.

```
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);

if dist <= 0.5 && sw == 0
    cvlin0 = CVLin;
    if 1e-2 - pi/2 < abs(TH - th) && abs(TH - th) < pi + 1e-2
        a = ((kR1*cvlin0)^2 - cvlin0^2)*C;
        a = 5*a;
    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
```

Let's start from the first `if-else` statement: if the Turtlebot is in the proximity of the waypoint and a switch variable is equal to zero (cf. figure 53: this variable is put equal to zero when entering the *Turning* state, to prevent a negative acceleration from arising when the Turtlebot travels between two waypoints that are close) a constant deceleration for a velocity law is calculated; when the curve that the Turtlebot will have to do is tight, the deceleration is stronger with respect to a smoother case.
The second `if-else` statement puts to practice the deceleration law when the Turtlebot is close to the waypoint, and it imposes a simple constant positive acceleration velocity law, considering a saturation, otherwise (the saturation is considered here, because the FSM uses the internally calculated command velocities that are internally fed back).
The last statement before the end of the function is there to avoid the Turtlebot to slow down excessively.

What is now remained to do is looking at the results.
In order not to thicken to much the discussion, it was decided to run the simulations only for two of the four identified paths.
The publishing of the first path gave the following outcome.

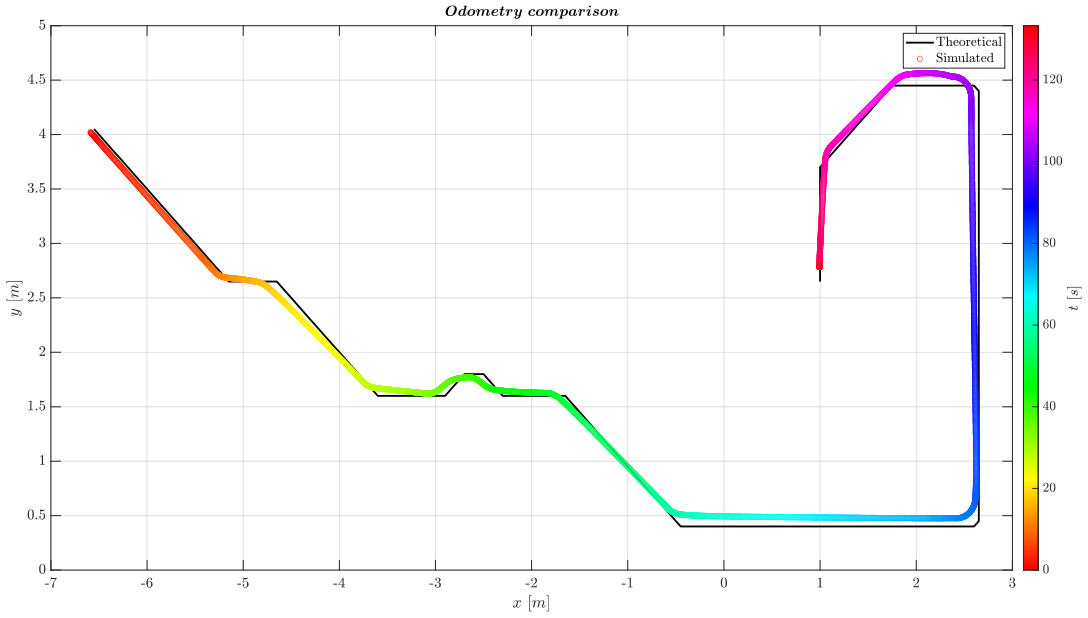

Figure 54: comparison between the theoretical trajectory versus the actual one.

The adherence between the theoretical trajectory and the actual one are due to two reasons: it was decided that, even for tight turns, the Turtlebot shouldn't stop (as in assignment 2) and a map that is not perfect. The latter phenomenon is, in fact, visible from figure 56.
It is reminded to the reader that the presence of some "strange" curves is due to the used algorithm.

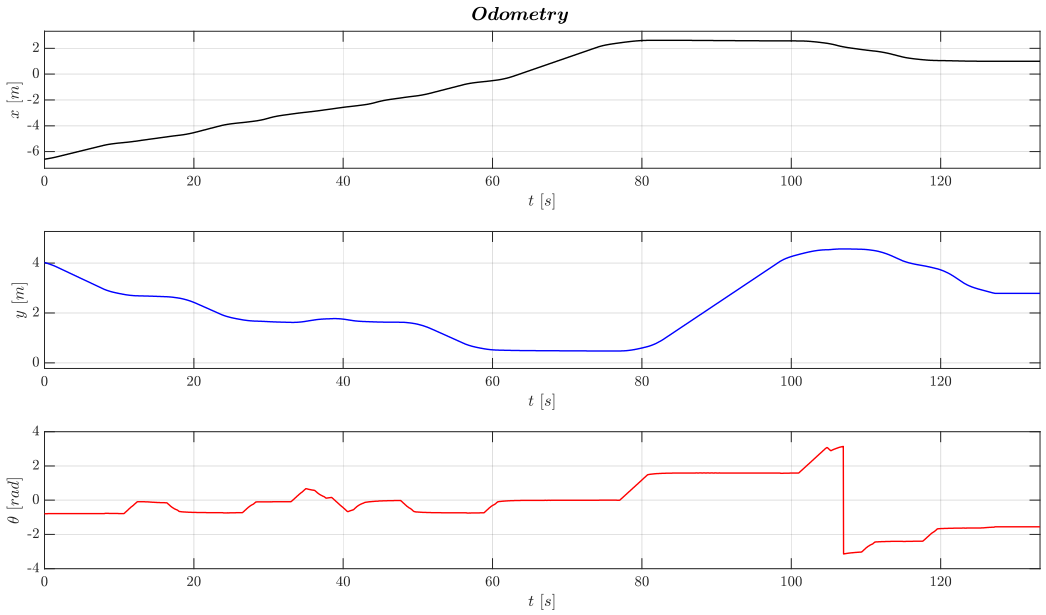

Figure 55: odometry evolution in time.

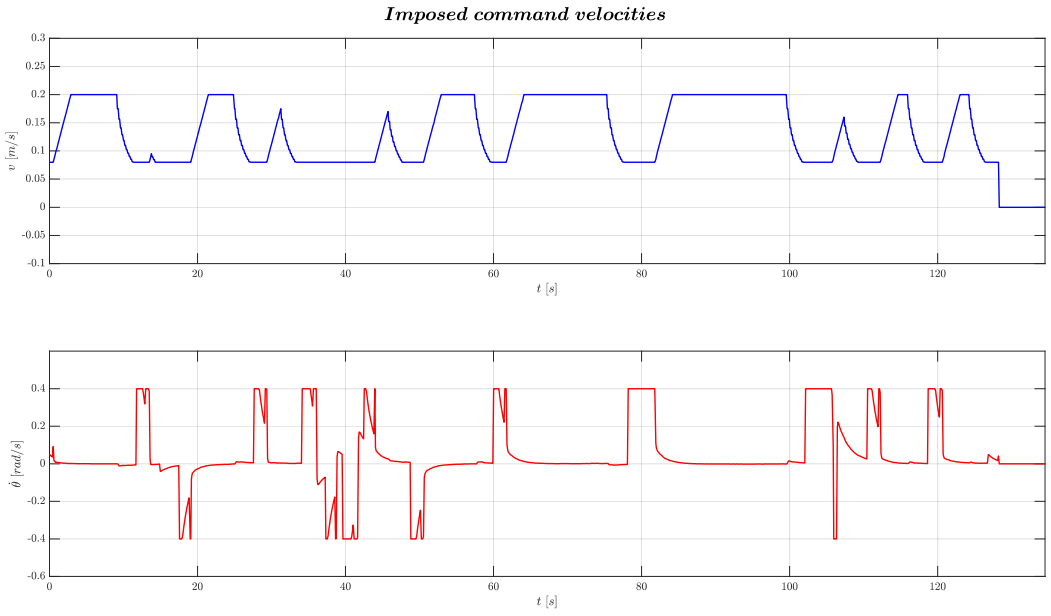

Figure 56: command velocities imposed.

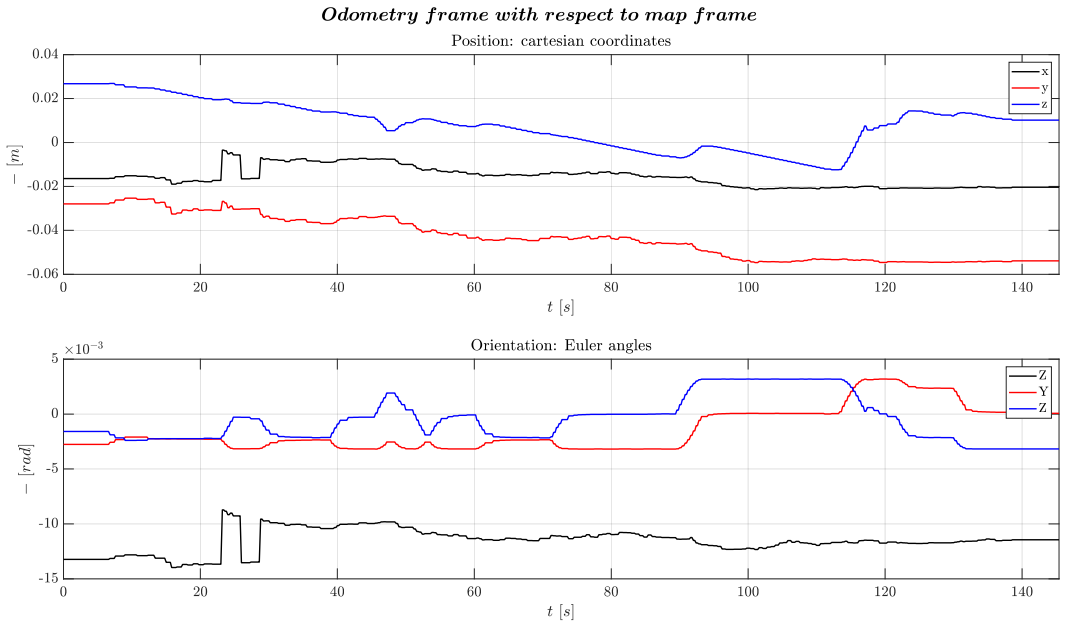

Figure 57: transformation from map frame to odometry frame.

As anticipated, the problem of a map which isn't perfect can be spotted in figure 57.
In fact, if the curves on this graph were horizontal straight lines – not necessarily zero, obviously, because that depends on the position of the origin of the map frame with respect to the origin of the odometry frame -, we'd have a perfect correspondence between the environment and the provided map.
What it can be inferred is that the variations are not that big in this case: in fact, this reflects in an almost negligible variation between the theoretical and actual trajectory.
A point of interest is the "tooth" after 20 seconds: there the Turtlebot turns for the first time (as per figure 54) and the localization updates twice, resulting in the actual trajectory deviating a bit from the theoretical one.

If we look at the results of the publication of the second set of waypoints, a similar result - without surprise – can be appreciated.

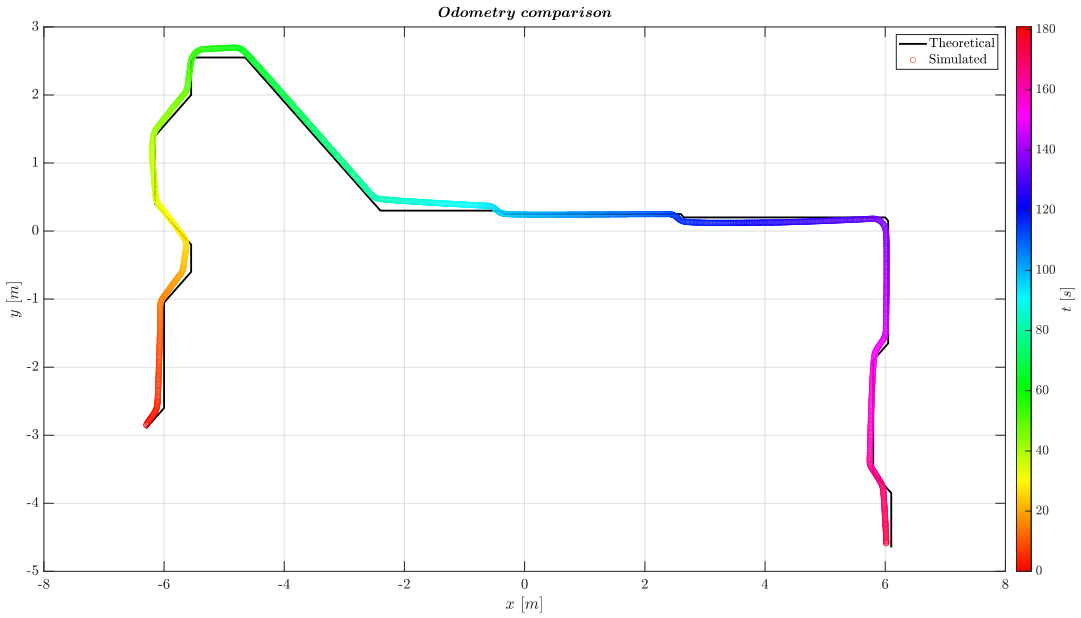

Figure 58: comparison between the theoretical trajectory versus the actual one.

Exactly as before, the differences between the theoretical trajectory and the actual one are rather small and imputable to the same causes, i.e. non-zero velocity in turns and localization update.

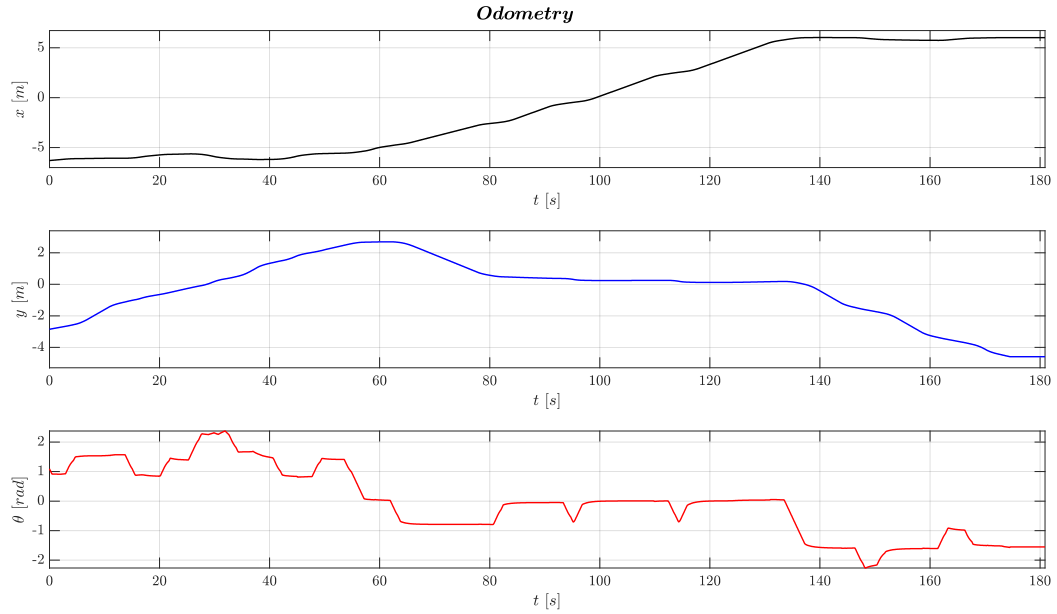

Figure 59: odometry evolution in time.

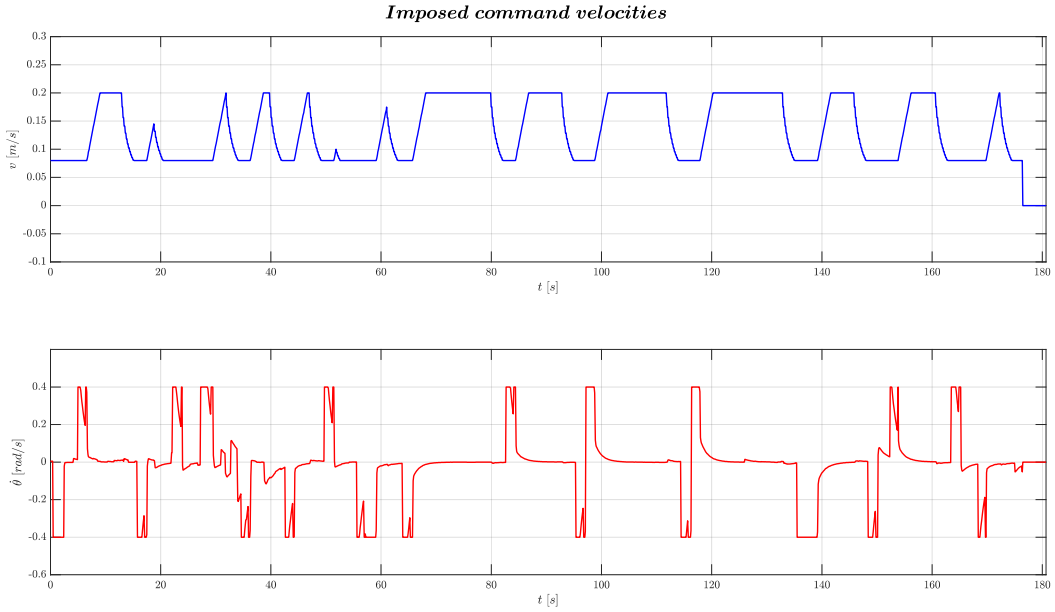

Figure 60: command velocities imposed.

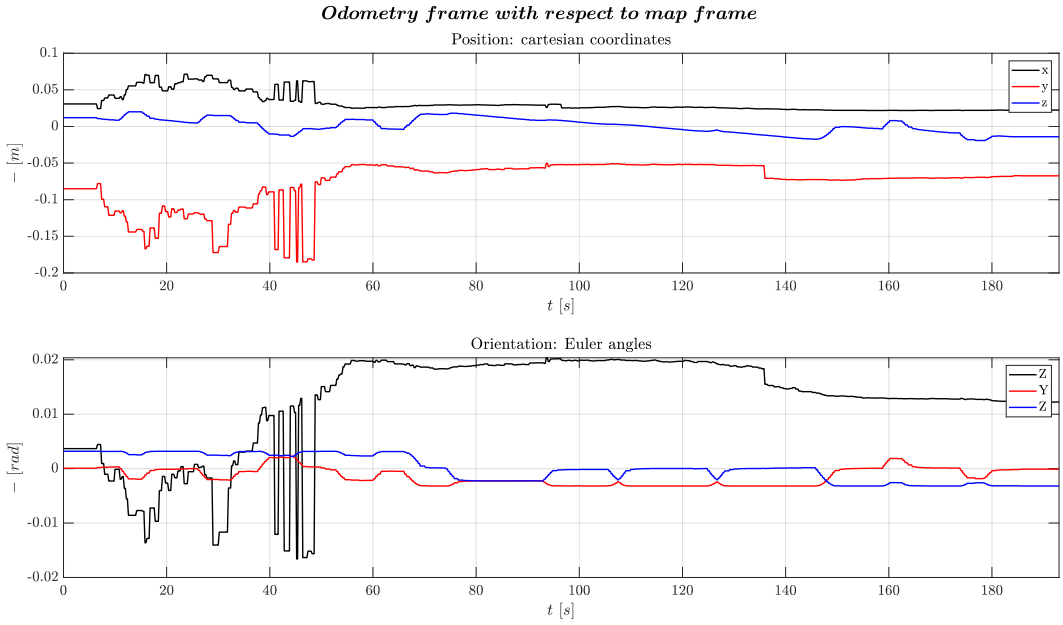

Figure 61: transformation from map frame to odometry frame.

The series of turns in the first part of the trajectory are really putting to the test the updating need of the localization services and this results in high frequency disturbances produced.
Nevertheless, as figure 58 shows, the tracking is well followed.

In conclusion, in order to get better results, the *gmapping* tool should be properly tuned, so that a more consistent map can be produced that lets the localization happen faster and without this much disturbances.

2. Final project 3

This project asks to define a path of waypoints, among which two type of spheres have to be positioned: red and green ones. A feedback control in the fashion of assignment 2 has to be implemented, but it has to be modified in order to make the Turtlebot avoid the obstacle balls: if the ball in front of it is red, the Turtlebot has to overtake it on the left, on the right otherwise. The used vehicle is the Turtlebot Waffle Pi.

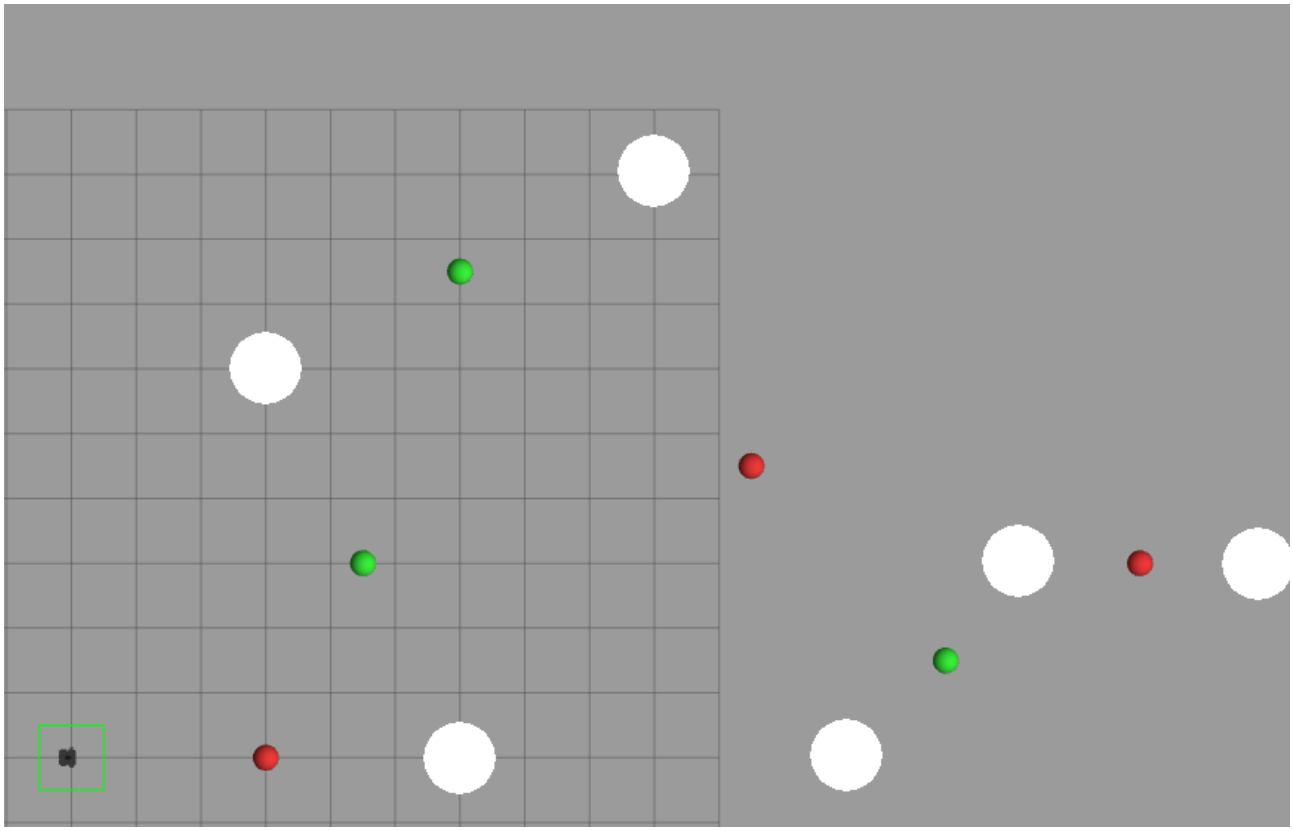

Figure 62: the white waypoint laying between the spheres.

The Simulink model used to approach the problem is divided in two distinct parts: the subsystem that reads the image coming from the camera in order to look for a ball and understand its color and a subsystem that reads the odometry and the scanned space around the Turtlebot and computes a feedback law by mean of a finite state machine (this part actually implements the same control law as for assignment 2, but in the FSM fashion).
In the camera reading part, the same procedure to recognize the red color of assignment 3 is followed, namely hue and saturation matrix in certain ranges are extracted and multiplied together, while for the green color the only needed parameter is the hue, due to specific nature of the HSV representation.
The red and green balls present on the image are then converted into BLOBs: a simple function to recognize the biggest of the two is implemented to detect the closest ball and its color, in order to output this information to the feedback controller. A very important part of this subsystem lies in the role of the information coming from the LIDAR sensor. The minimum distance from the obstacle is calculated by means of this and, leveraging a look-up table that was calibrated to relate a distance from an obstacle and its visible area in m^2 to an area in pixels along with the information that the obstacle is spherical, the diameter of the ball is calculated and inputted in the finite state machine regulator.

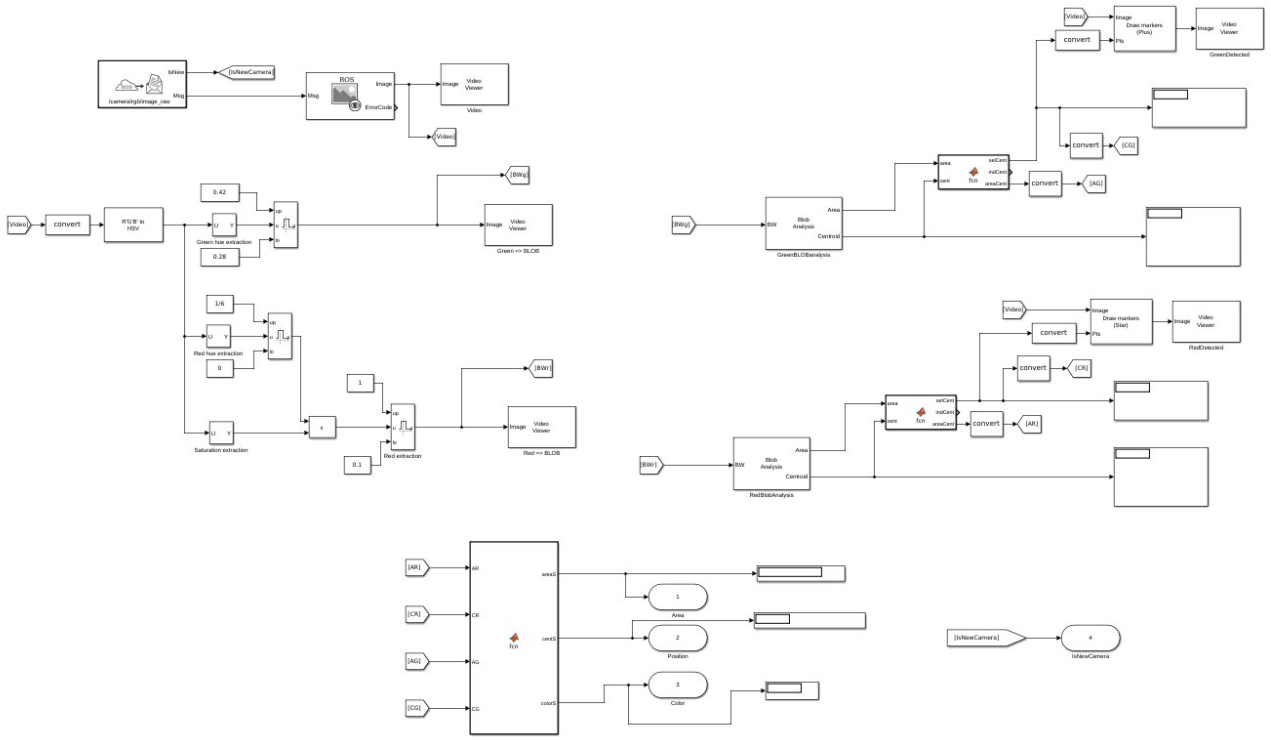

Figure 63: camera reading subsystem.

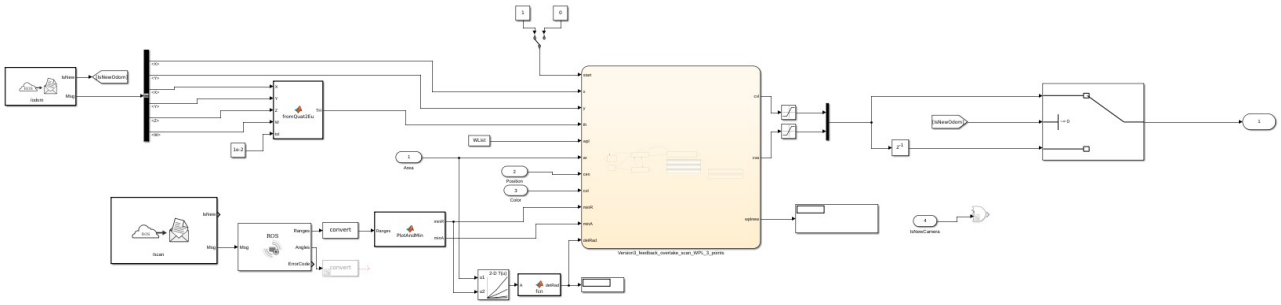

Figure 64: feedback controller subsystem.

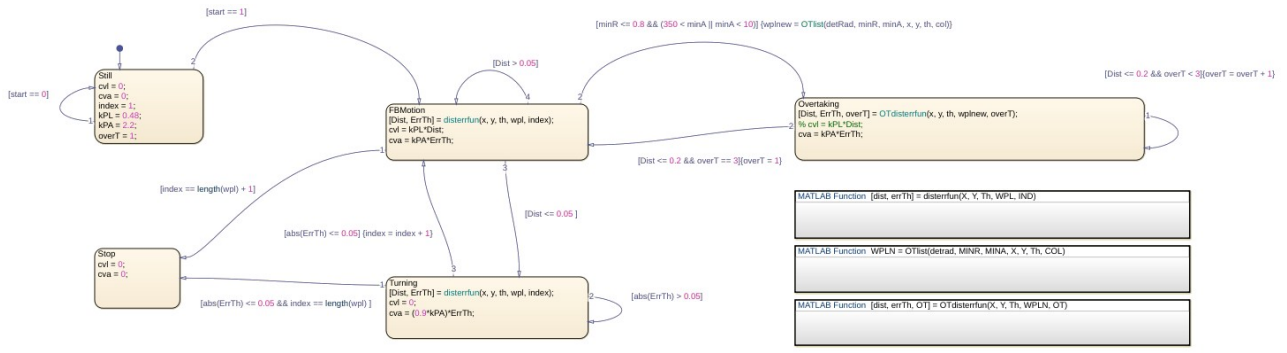

Figure 65: the finite state machine actually implementing the feedback control.

As anticipated, the finite state machine implements the same law as assignment 2, with an important exception, the *Overtaking* state. The *FBMotion* and *Turning* states actually do the same thing as previously (namely stop in every turn), but the *Overtaking* state was implemented to pass around the obstacles.
The overtaking is performed by means of a feedback control applied to a list of waypoints that is calculated before entering the *Overtaking* state. These waypoints are defined as x and y goals – no reference angular position - and are strategically positioned around the obstacle: the diameter of the ball previously calculated here comes into play to calculate their values. During the overtake, the longitudinal velocity is maintained constant and equal to the one imposed before entering this state. In order to turn promptly, a bigger proportional term was used to feedback the angular velocity.
In order to enter the overtaking state, the obstacle has to be at certain distance from the Turtlebot, and the controller makes sure that the object at minimum distance is actually in front of the Turtlebot by considering also the angle corresponding to the minimum range found by the LIDAR.

In figure 66 we can see the trajectory of the robot during its travel through the waypoints.

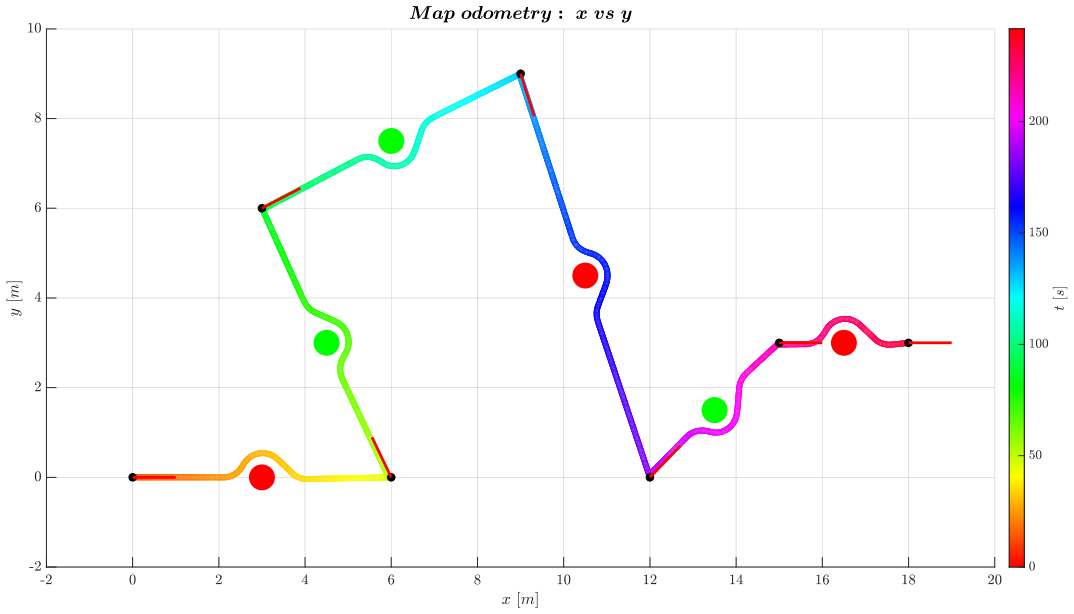

Figure 66: Turtlebot's trajectory.

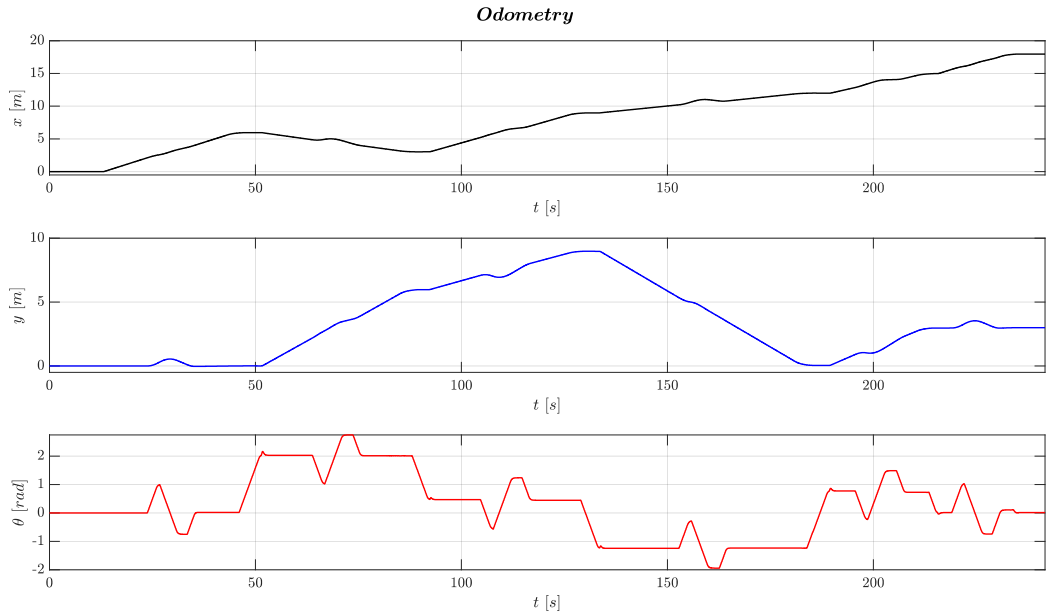

Figure 67: odometry plots.

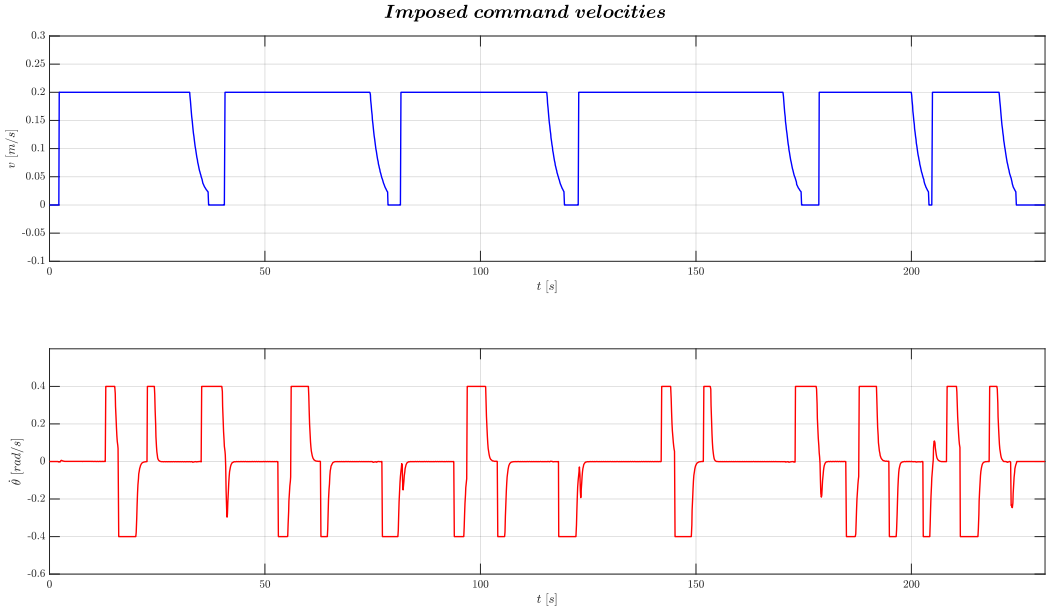

Figure 68: imposed command velocities.

The usual saturation to command velocities from assignment 2 has been taken into consideration. We see that the behaviors are pretty smooth and spikes-less.