



POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Autonomous Vehicles

Course Assignments

Author:

N. Visentin

10797203 - 252081

Lecturers:

S. Arrigoni

Academic Year: 2024/25

Contents

1	Assignment I	3
1.1	Part A	3
1.2	Part B	4
2	Assignment II	6
2.1	Main part	6
2.2	Extra A	8
2.3	Extra B	9
2.3.1	Posture Regulation	9
2.3.2	LQR control	10
3	Assignment III	14
4	Assignment IV	17
4.1	Map elaboration	17
4.2	Searching algorithms	18
4.3	Results	19
5	Assignment V	25
5.1	Main part	25
5.2	Extra	27

1 Assignment I

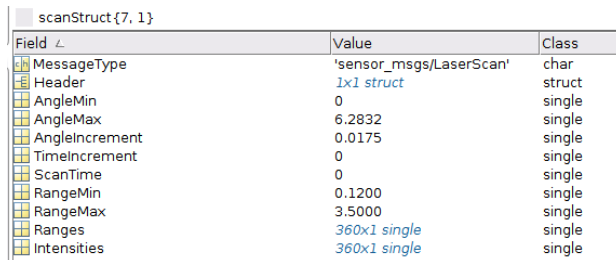
The first assignment consists in the analysis of a .bag file that contains some data of a simulation performed using ROS and Gazebo, where a turtlebot3 waffle pi robot moves in a certain environment. The available topics in the provided bag are:

- \clock: clock time of the simulation
- \imu: data of the IMU sensor
- \odom: odometry data
- \scan: laser scan data
- \tf: related to the reference frames

It is required to evaluate the minimum distance of the robot from obstacles at each time instant, estimate the input velocity command that made the robot move and finally check the reconstructed signal by simulating it.

1.1 Part A

Minimum distance from obstacles Laser scan data are considered to study the minimum distance of the robot from obstacles during the simulation. Using MatLab, these data are extracted from the bag in form of a cell array, where each cell contains the information provided by the laser scan in a certain time instant, as a structure. As we can see in *figure 1*, the laser scan has an angular resolution of 1° (0.0175 rad), from 0 to 360° (6.2832 rad) and a range from 0.12 m to 3.5 m; moreover, data are provided at 5 Hz.



Field	Value	Class
MessageType	'sensor_msgs/LaserScan'	char
Header	1x1 struct	struct
AngleMin	0	single
AngleMax	6.2832	single
AngleIncrement	0.0175	single
TimeIncrement	0	single
ScanTime	0	single
RangeMin	0.1200	single
RangeMax	3.5000	single
Ranges	360x1 single	single
Intensities	360x1 single	single

Figure 1: Laser scan data structure at a certain time instant

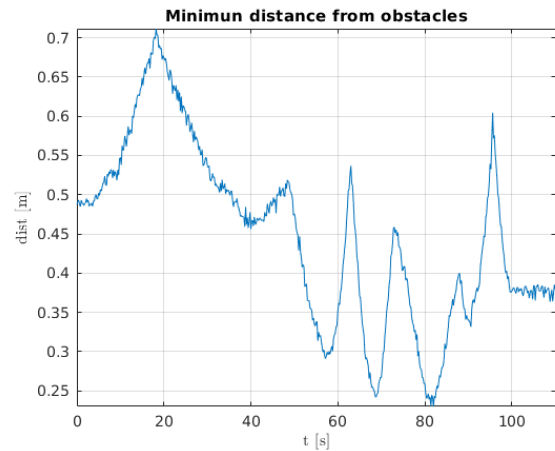


Figure 2: Minimum distance from obstacles during the simulation

The field Ranges can be used to track the distance from obstacles around the robot: by considering the minimum value for each time instant, the plot in *figure 2* is obtained. Notice that in this figure time has been rescaled, making it start from zero.

Velocity command Since the published command velocity signal was not provided in the bag, odometry can be used to estimate it. Similarly to what was done with the laser scan, data are loaded in MatLab and processed to obtain the measured trajectory (*figure 3a*) and the measured linear velocity and yaw rate (*figure 3b*). Once again, time is rescaled starting from zero, while this time odometry topic has a frequency of 30 Hz.

Data in *figure 3b* are filtered using a median filter (*figure 4a*) in order to reduce noise/spikes, and then the command input is reconstructed by rounding the obtained signals at step increments of 0.01 m/s (for linear speed) or 0.1 rad/s (for yaw

rate) (figure 4b): these values are chosen by looking closely to the shape of the signals and are consistent with the typical increments given by teleop control from keyboard, which is probably the method that was used to move the robot during the simulation. This procedure is shown in figure 4.

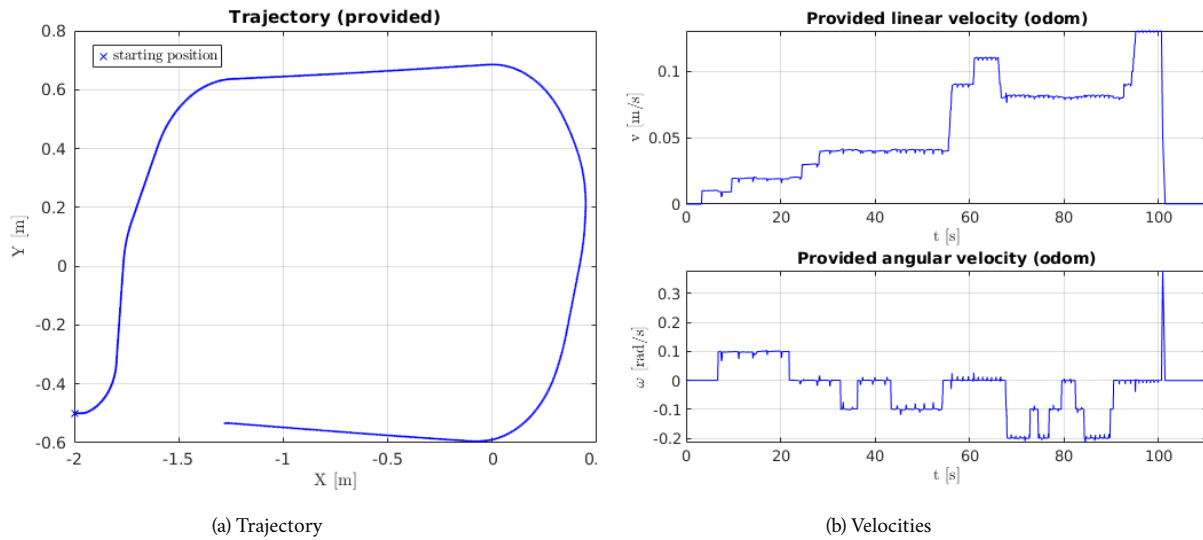


Figure 3: Odometry data

1.2 Part B

The reconstructed command is tested by publishing it through `\cmd_vel` in ROS, using a very simple Simulink model. Simulation data are saved in another .bag file, that is then processed the same way as the provided one, as we saw in section 1.1.

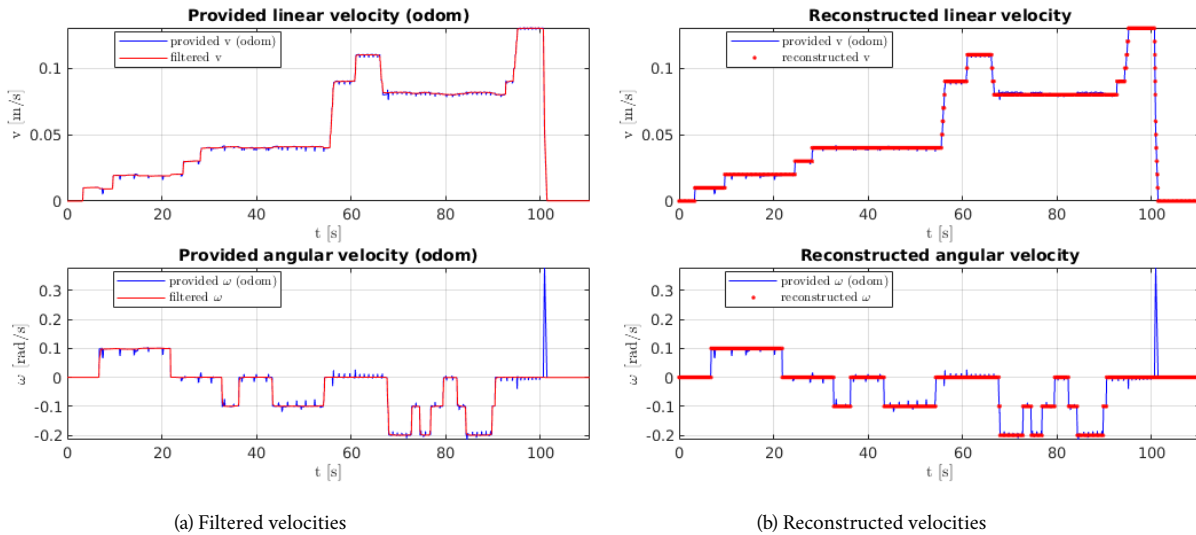


Figure 4: Reconstructing velocity commands

The results of the simulation are shown in figure 5, where a comparison is made with the data of the provided bag, where the velocity command was unknown. As we can see, the trajectory followed by the turtlebot is almost the same, in particular at the beginning, and also the data of the laser scan are similar. The measured velocities are also very close, apart some noisy spikes, meaning that the reconstruction of the command velocity was done properly.

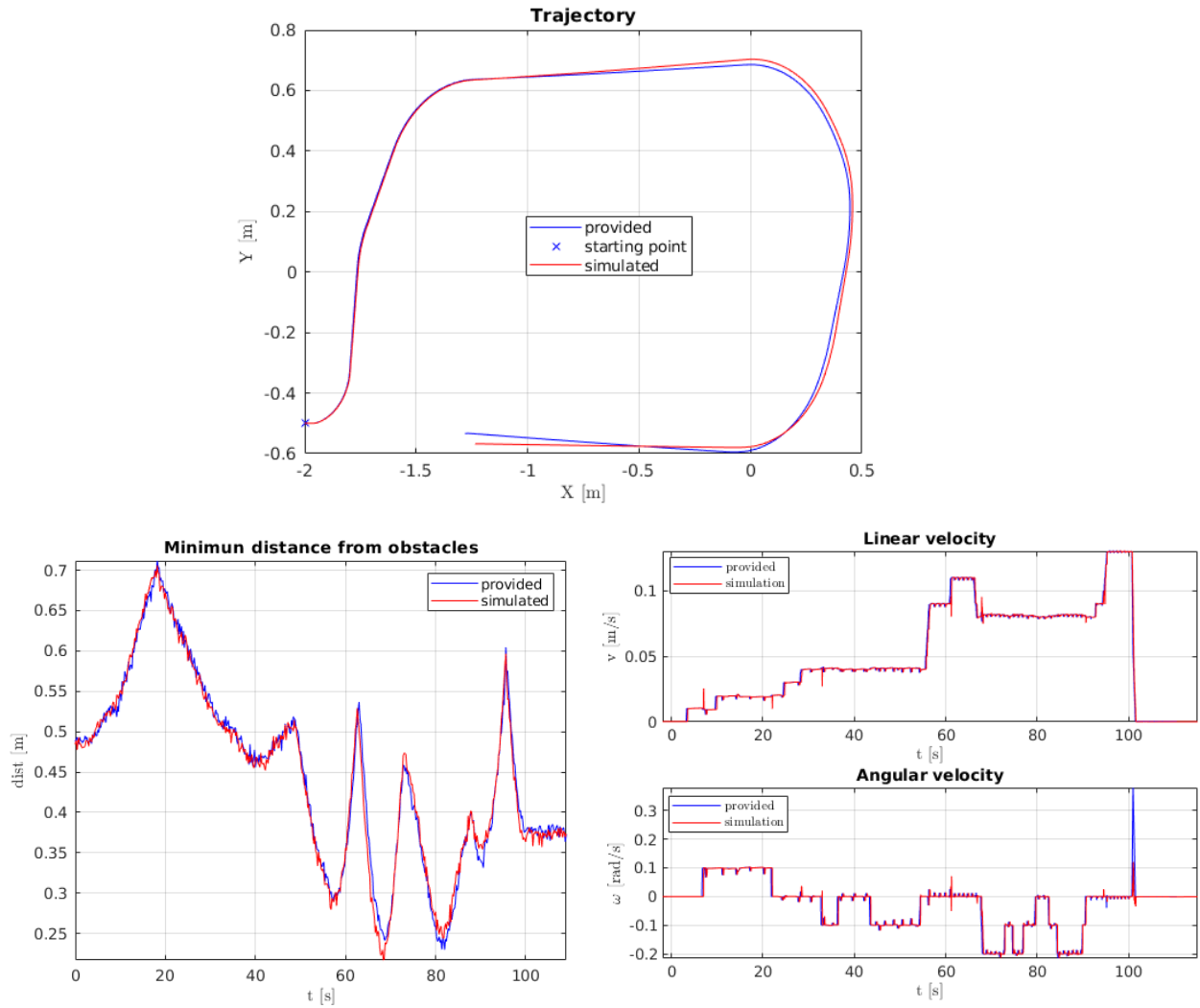


Figure 5: Comparison between simulated and provided data

2 Assignment II

In the second assignment, a list of waypoints is provided in terms of position x, y and orientation θ (figure 6). The task consists in making the turtlebot burger move in an empty world through all these waypoints by controlling its linear velocity and yaw rate with a simple feedback control, and then stop at the end. Waypoints are initially loaded in MatLab/Simulink, while in the second part of the assignment, the same waypoints are provided through an external source (a Python node, in this case). Finally, two other tracking/feedback strategies are implemented to drive the robot through the waypoints: a posture regulation algorithm and an LQR control on an optimal trajectory.

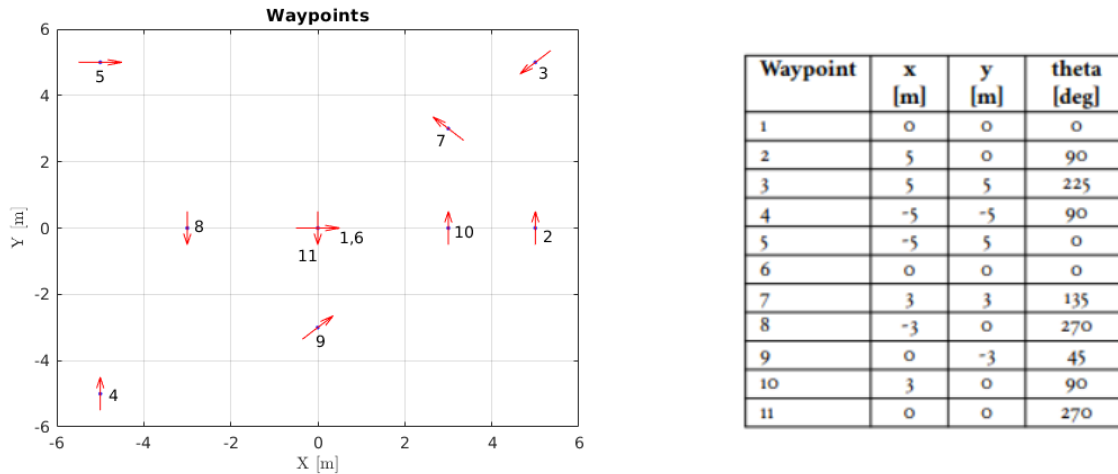


Figure 6: Waypoints

2.1 Main part

At the beginning, the following restrictions were taken:

- linear speed: $|v| \leq 0.2$ m/s
- yaw rate: $|\omega| \leq 0.4$ rad/s

This implies that the robot is very slow with respect to the distances between waypoints (minimum distance between a waypoint and the following one is 3 m, meaning that even in that case it takes more or less 15 seconds to cover that distance at maximum speed). For this reason, the control strategy between two waypoints (x_i, y_i, θ_i) and $(x_{i+1}, y_{i+1}, \theta_{i+1})$ was simply and intuitively organized in two "parts", as follows:

- **"travel"**: starting from the i -th waypoint, the turtlebot advances "as straight as possible" towards the next one. In this phase, the linear velocity v (saturated at v_{max}) is:

$$v = K_{dist} \cdot d, \quad d \text{ is the distance from the next waypoint}$$

where the proportional gain is chosen such that $v = v_{max}$ during most of the path, while the yaw rate is controlled as:

$$\omega = K_{dir} \cdot e_{dir}, \quad e_{dir} \text{ is the error on the orientation with respect to the position of the next waypoint}$$

In particular, $K_{dist} = 1$ and $K_{dir} = 10$.

- **"heading regulation"**: as soon as the robot is close enough to the next waypoint, i.e. $d < toll_d$, it simply stops and adjusts its orientation to match the required angle θ_{i+1} , so:

$$v = 0$$

$$\omega = K_{or} \cdot e_{or} \quad , \quad \text{where } e_{or} \text{ is the error with respect to } \theta_{i+1}$$

When the orientation is close enough to θ_{i+1} , i.e. $e_{or} < toll_{or}$, the controller switches back to "travel", considering the next pair of waypoints.

This scheme, with an additional check on the reaching of the final waypoint that stops the robot and the simulation, was implemented in Simulink, whose model can be seen in *figure 7*. Odometry was exploited to compute d , e_{dir} and e_{or} and provide the feedback action, while the waypoints were loaded directly from MatLab, as previously mentioned.

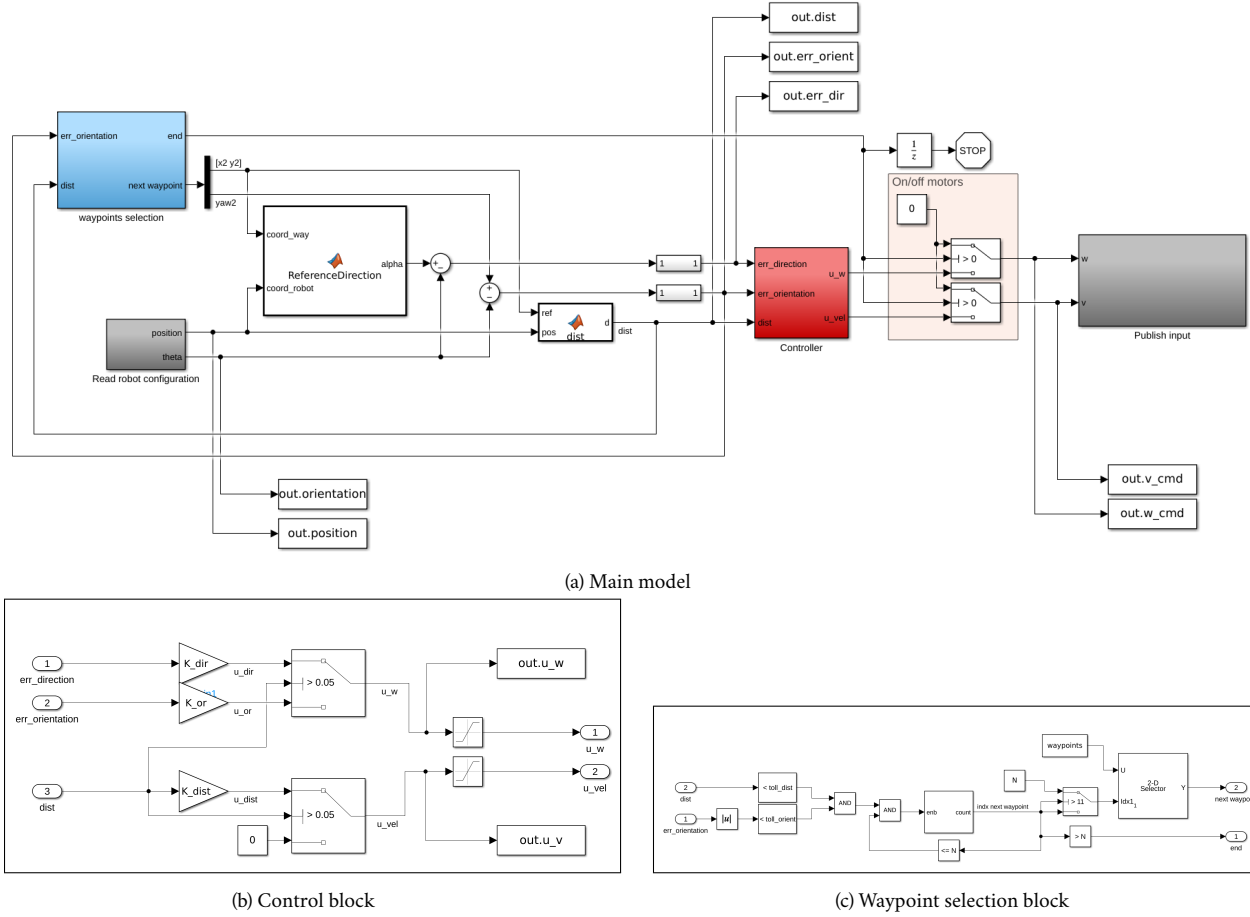


Figure 7: Simulink model for assignment II, first part

Results of the simulation (*figure 8*) show that the *turtlebot* is able to pass through all the waypoints. As expected, the control action is very simple, basically imposing a constant maximum linear speed and a null yaw rate during the "travel" phases, while setting $v = 0$ and turning the robot with maximum yaw rate during the "heading regulation" phases. This is of course effective in this scenario, where the velocities are limited to small values, and the vehicle's dynamics is almost irrelevant: in fact, during the simulation in Gazebo, the robot has never been observed to slide or to be subject to "heavy" inertial effects. Also notice that it took more than 5 minutes to complete the full trajectory.

In section 2.3, where two different feedback control strategies will be presented, these restrictions on the linear speed and yaw rate will be relaxed.

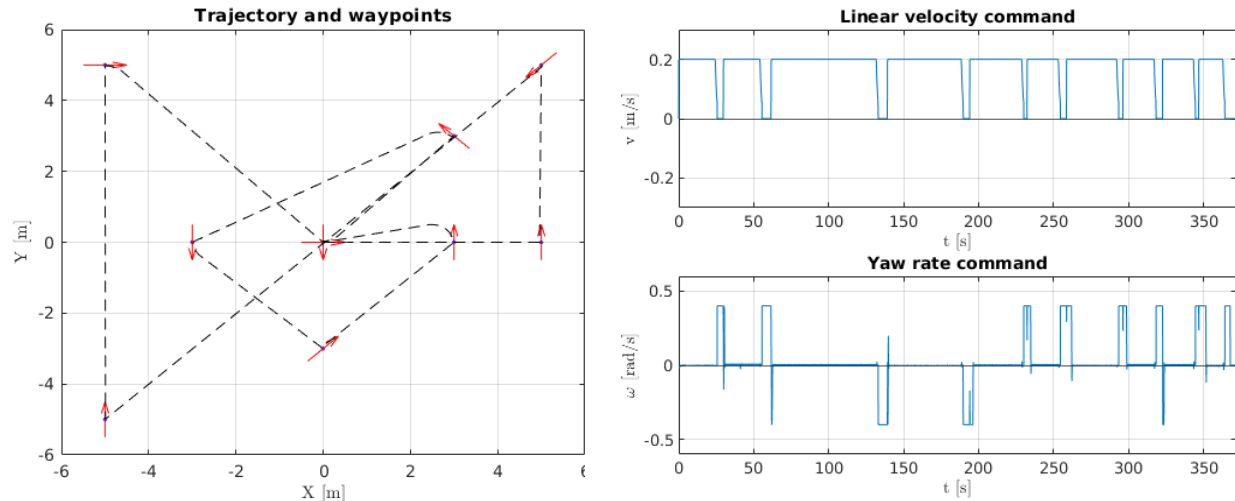


Figure 8: Simulation results. Click [here](#) to visualize the animation

2.2 Extra A

It is also possible to provide the waypoints with an external publisher. There are actually many ways to do that: an idea is, for example, to write a Python node that let the user input a waypoint, publishes it, and then, when the `turtlebot` reaches the goal, it asks for another waypoint and so on. However, the chosen implementation in our case is much simpler: the entire waypoints list is directly included in a Python node, that then exploits the "Covariance" field of a ROS geometry message (`PoseWithCovariance`) to publish the previously mentioned list. At this point, Simulink can subscribe to it and apply the control we saw in section 2.1. This node is shown in *figure 10*.

The rest of the codes and control strategies are not different from what seen in section 2.1, where the only difference is the "waypoint selection block" in *figure 7c*, which now must be able to subscribe to `PoseWithCovariance` message and read the waypoints, instead of taking them directly from MatLab workspace. Also the results of the simulation are identical to the ones obtained before, since the only thing that changed is the way the reference points are provided. Thus they are not reported.

geometry_msgs/PoseWithCovariance Message

File: `geometry_msgs/PoseWithCovariance.msg`

Raw Message Definition

```
# This represents a pose in free space with uncertainty.
Pose pose
# Row-major representation of the 6x6 covariance matrix
# The orientation parameters use a fixed-axis representation.
# In order, the parameters are:
# (x, y, z, rotation about X axis, rotation about Y axis, rotation about Z axis)
float64[36] covariance
```

Compact Message Definition

```
geometry_msgs/Pose pose
float64[36] covariance
```

Figure 9: `PoseWithCovariance` ROS message

It's relevant to highlight that "Covariance" inside `PoseWithCovariance` message is a vector that contains 36 `float64` elements (*figure 9*), meaning that we are limited at most to 12 waypoints. Also, it's worth to mention that the code shown in *figure 10* could have been simpler, just directly filling the vector `msg.covariance` with the desired points. However,

the idea in this case was to somehow provide the waypoints as a matrix, and let the code handle the fact that the given points can also be less than 12, setting in that case a flag and the actual number of waypoints in the last positions of the Covariance vector.

```

1#!/usr/bin/env python
2
3import rospy
4import math
5import numpy
6from math import pi
7from geometry_msgs.msg import PoseWithCovariance
8
9# define waypoints
10
11waypoints = [
12    [0, 0, 0],
13    [0, 0, pi/2],
14    [5, 5, 5/4*pi],
15    [-5, -5, pi/2],
16    [-5, 5, 0],
17    [0, 0, 0],
18    [3, 3, 3/4*pi],
19    [-3, 0, 3/2*pi],
20    [0, -3, pi/4],
21    [3, 0, pi/2],
22    [0, 0, 3/2*pi]
23]
24
25def pub_waypoints():
26    pub = rospy.Publisher('waypoints', PoseWithCovariance, queue_size=100)
27    rospy.init_node('publisher_waypoints', anonymous=False)
28    rate = rospy.Rate(100) # 100 Hz
29
30    while not rospy.is_shutdown():
31        msg = PoseWithCovariance()
32
33        for i,row in enumerate(waypoints):
34            msg.covariance[3*i:3*(i+1)] = row
35
36            if i == len(waypoints)-1 and 3*len(waypoints)<36:
37                msg.covariance[34]=1001
38                msg.covariance[35]=len(waypoints)
39
40            #rospy.loginfo(msg) # this allows for displaying the message in the terminal
41            pub.publish(msg)
42            rate.sleep()
43
44if __name__ == '__main__':
45    try:
46        pub_waypoints()
47    except rospy.ROSInterruptException:
48        pass

```

Figure 10: Python node for assignment II, part A

```

vise@vise-IdeaPad-Gaming-3-16IAH7: ~
vise@vise-IdeaPad-Gaming-3-16IAH7: ~$ rostopic /list
/clock
/cmd_vel
/gazebo/link_states
/gazebo/model_states
/gazebo/parameter_descriptions
/gazebo/parameter_updates
/gazebo/performance_metrics
/gazebo/set_link_state
/gazebo/set_model_state
/jnu
/joint_states
/odom
/rosout
/rosout_agg
/scan
/rf
/waypoints
vise@vise-IdeaPad-Gaming-3-16IAH7: ~$
vise@vise-IdeaPad-Gaming-3-16IAH7: ~$ rostopic pub -r 100 1000 /waypoints PoseWithCovariance
covariance: [0.0, 0.0, 0.0, 5.0, 0.0, 1.5707963267948966, 5.0, 5.0, 3.9269908169
872414, -5.0, -5.0, 1.5707963267948966, -5.0, 5.0, 0.0, 0.0, 0.0, 0.0, 3.0, 3.0,
2.356194490192345, -3.0, 0.0, 4.71238898038469, 0.0, -3.0, 0.7853981633974483,
3.0, 0.0, 1.5707963267948966, 0.0, 0.0, 4.71238898038469, 0.0, 1001.0, 11.0]
pose:
position:
x: 0.0
y: 0.0
z: 0.0
orientation:
x: 0.0
y: 0.0
z: 0.0
w: 0.0
covariance: [0.0, 0.0, 0.0, 5.0, 0.0, 1.5707963267948966, 5.0, 5.0, 3.9269908169
872414, -5.0, -5.0, 1.5707963267948966, -5.0, 5.0, 0.0, 0.0, 0.0, 0.0, 3.0, 3.0,
2.356194490192345, -3.0, 0.0, 4.71238898038469, 0.0, -3.0, 0.7853981633974483,
3.0, 0.0, 1.5707963267948966, 0.0, 0.0, 4.71238898038469, 0.0, 1001.0, 11.0]
min: 0.009s max: 0.011s std dev: 0.00013s window: 4696
average rate: 100.000
min: 0.009s max: 0.011s std dev: 0.00013s window: 4796
average rate: 100.000
min: 0.009s max: 0.011s std dev: 0.00013s window: 4896
average rate: 100.000
min: 0.009s max: 0.011s std dev: 0.00013s window: 4996
average rate: 100.000
min: 0.009s max: 0.011s std dev: 0.00013s window: 5096
average rate: 100.000
min: 0.009s max: 0.011s std dev: 0.00013s window: 5196

```

Figure 11: PoseWithCovariance message being published in the \waypoints topic

2.3 Extra B

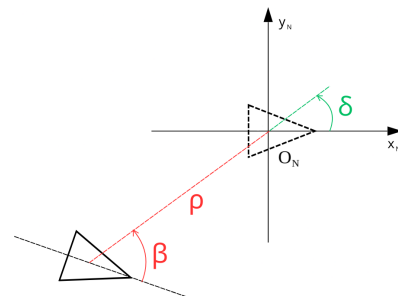
In this section, two different feedback control strategies are considered to drive the robot through the provided waypoints. To make them effective compared to the simple control presented in section 2.1, speed limitations have been relaxed.

2.3.1 Posture Regulation

The first method is a posture regulation feedback control that allows a vehicle to stabilize in a final "parking" position (reference: Aicardi, Michele, et al. "Closed loop steering of unicycle like vehicles via Lyapunov techniques." *IEEE robotics & automation magazine* 2.1 (1995): 27-35.). A unicycle model of the robot is considered, and the main purpose is to define a smooth and time-invariant control law that allows the vehicle to asymptotically stabilize in a final goal position. The proposed control law is the following:

$$\begin{cases} v = K_1 \rho \cos \beta \\ \omega = K_2 \beta + K_1 \frac{\sin \beta \cos \beta}{\beta} (\beta + K_3 \delta) \end{cases} \quad (1)$$

Notice that this equations will stabilize the robot to the "origin" configuration (0, 0, 0). To bring it to an arbitrary waypoint (x₂, y₂, θ₂) from a general configuration (x, y, θ), we need to change coordinates, computing β and δ in our global reference frame O_{x,y}:



$$\beta = \alpha - \theta$$

$$\delta = \alpha - \theta_2$$

We also need to pay attention to $\beta \rightarrow 0$, since this is an asymptotical control strategy. For small β , i.e. small heading error, the term $\frac{\sin \beta \cos \beta}{\beta}$ can lead to numerical instability in Simulink, so it is necessary to impose:

$$\beta = K_1 K_3 \delta, \quad \text{when } \beta < \text{toll}_\beta$$

This comes from Equation (1) and:

$$\lim_{\beta \rightarrow 0} \frac{\sin \beta \cos \beta}{\beta} = 1$$

These laws were implemented in a similar setup to the one seen in section 2.1, so the expected outcome is the robot "jumping" from one waypoint to the next one, following a path defined by this posture regulation parking feedback control. Of course, as before, a tolerance was set so that when a certain waypoint is reached, the next one becomes the new goal, and the simulation automatically stops, also stopping the `turtlebot`, when the final position is met. Moreover, $|v| < 0.5 \text{ m/s}$ and $|\omega| < 2.84 \text{ rad/s}$ were assumed in this case.

Results of the simulation are shown in *figure 12*.

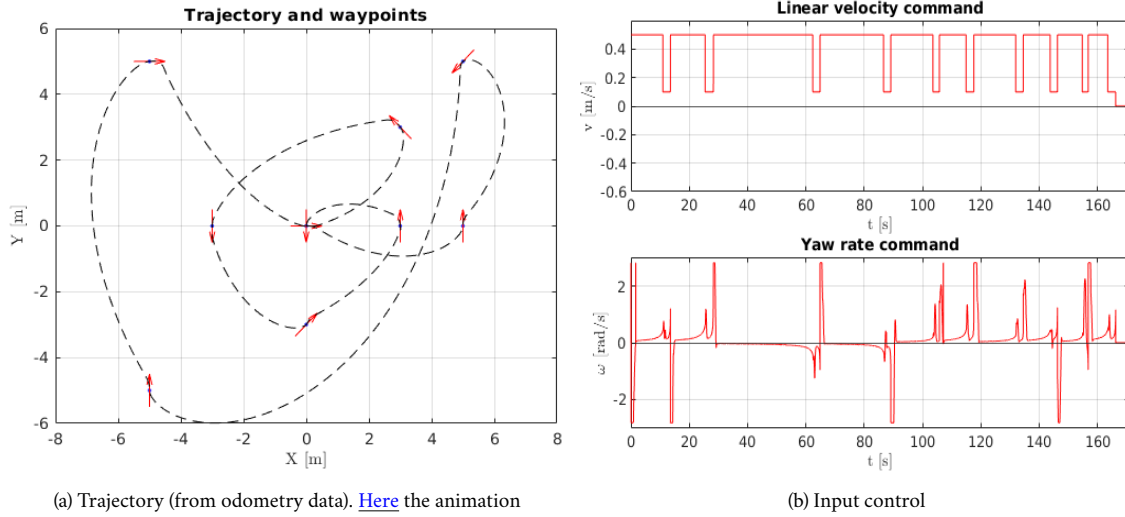
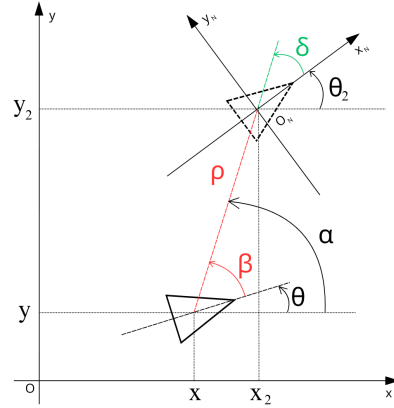


Figure 12: Results of assignment II, part B, posture regulation method

As we can see, in this case the time required to cover all the waypoints is much lower (more or less 2 minutes and 45 seconds), since the limit linear speed is higher. In addition, the trajectories are more complex, like the vehicle is "adapting" the heading to correctly approach the next waypoint. Notice how also in this case, the linear velocity gain is set such that v is almost always equal to v_{max} .

Finally, we need to remark that the problem is posed in this case in a slightly different way: the goal is not to cover all the waypoints in the fastest way possible, but to "park" in each one of the goal positions and then go to the next one: that's why the posture regulation "parking" control law was used.

2.3.2 LQR control

The last additional control strategy that was considered consists in a Linear Quadratic Regulator (LQR) that operates on an optimal trajectory.

Optimal trajectory The trajectory was computed by solving an optimal control problem with a direct method (direct transcription), considering the unicycle model of the robot. The strategy was the following:

- Divide the whole path into "pieces": each "piece" is the connection between two consecutive waypoints (x_i, y_i, θ_i) and $(x_{i+1}, y_{i+1}, \theta_{i+1})$.
- Solve the optimization problem with initial condition $(x_{init}, y_{init}, \theta_{init})$ equal to the final configuration of the previous optimization (which should be hopefully close to (x_i, y_i, θ_i)), and final target $(x_{fin}, y_{fin}, \theta_{fin}) = (x_{i+1}, y_{i+1}, \theta_{i+1})$. Optimization variables are the control (v, ω) , the states (x, y, θ) and time (t_f) , with proper weights. It's a bounded problem, with $0 < v < 0.5$ and $-1 < \omega < 1$. Cost functional was chosen as:

$$J = \Phi(\underline{\xi}(t_f), t_f) + \int_0^{t_f} L(\underline{\xi}, \underline{u}) dt, \quad \text{with } \underline{\xi} = [x, y, \theta]^T \text{ and } \underline{u} = [u, v]^T$$

where

$$\Phi(\underline{\xi}(t_f), t_f) = \frac{1}{2} (\underline{\xi}(t_f) - \underline{\xi}_{i+1})^T \cdot \underline{P} \cdot (\underline{\xi}(t_f) - \underline{\xi}_{i+1}) + \alpha \cdot (t_f - 0)$$

$$L(\underline{\xi}, \underline{u}) = \frac{1}{2} \underline{\xi}^T \cdot \underline{Q} \cdot \underline{\xi} + \frac{1}{2} \underline{u}^T \cdot \underline{R} \cdot \underline{u}$$

Matlab was exploited to solve the minimization problem. The resulting trajectory, along with the optimal control and states, is plotted in figure 13.

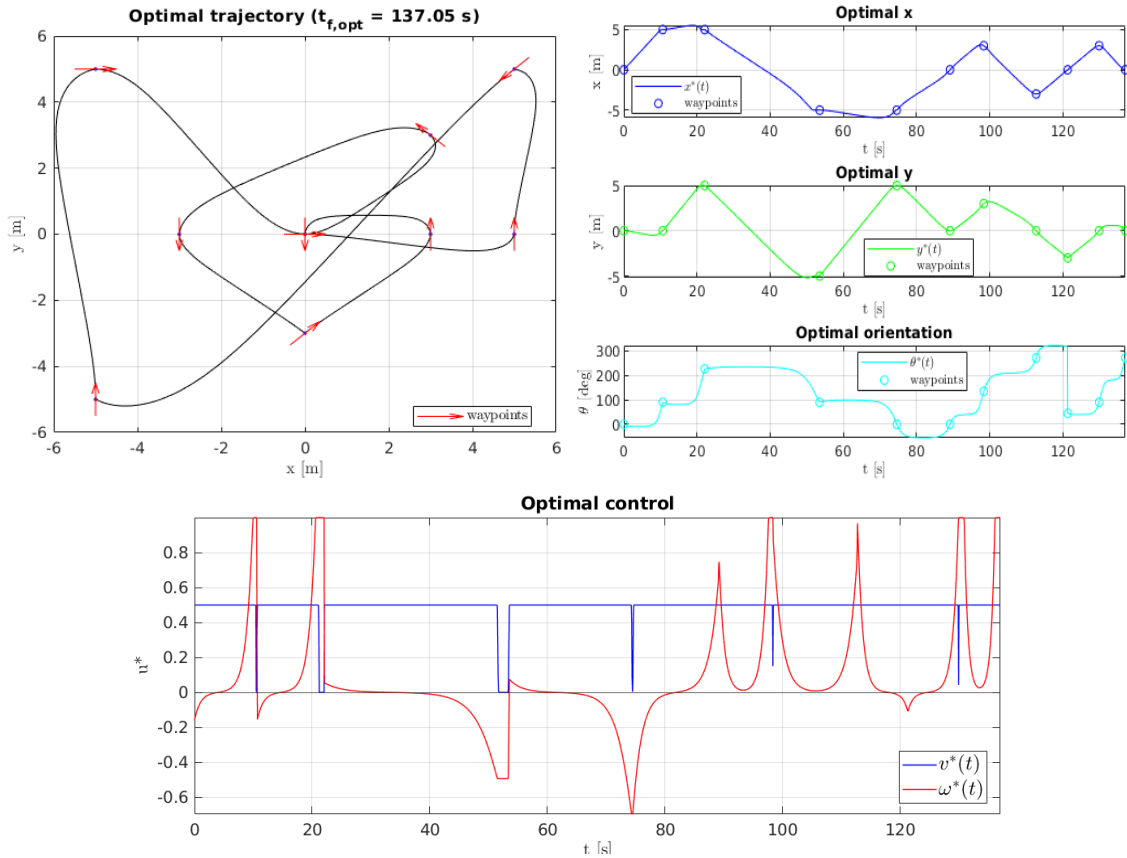


Figure 13: Optimisation results

LQR definition A finite-time Linear Quadratic Regulator is implemented so that a state-feedback control allows the vehicle to follow the previously computed path. Differential Riccati Equation is solved offline in MatLab to define the

gains for the controller at each time instant, and these gains, together with the optimal references, are then generated in Simulink in real-time during the simulation using timeseries. This setup is tested, as usual, using ROS and Gazebo, subscribing to odometry data and publishing velocity commands from Simulink. Speed limits are set to 0.6 m/s for linear velocity and 1 rad/s for yaw rate.

Results of this simulation are shown in *figure 14*. As we can see, the robot is able to cover all the waypoints staying quite close to the reference trajectory. The most critical parts are near the waypoints themselves, due to the discontinuity (both in the trajectory and the control input) given by the method used to generate the optimal trajectory.

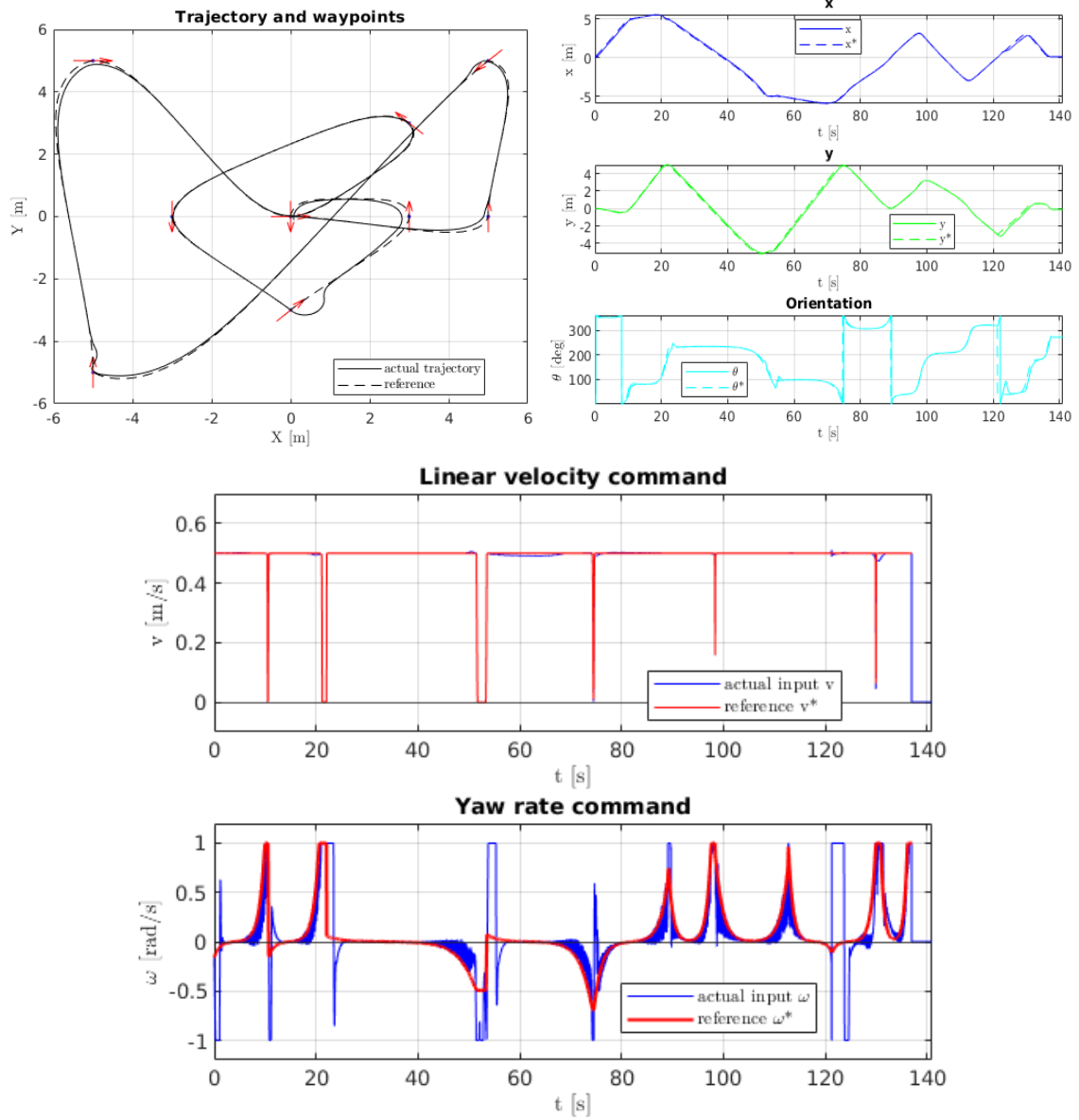


Figure 14: LQR simulation results. [Here](#) the animation

Finally, we can observe that the turtlebot was able to complete the path in 137 seconds (2 minutes and 17 seconds), which is the optimal time from optimization and also the fastest time among all tested control strategies. It's worth to mention that the actual expected time should be slightly higher than the optimal one, since the minimization procedure is neglecting all inertia and there are some inaccuracies in the tracking. However, to overcome possible "delays" due to

inertial effects, the bound on the linear speed during optimization ($0 < v < 0.5$ m/s) was a bit lower than the actual limit imposed during the simulation ($0 < v < 0.6$ m/s), so that the vehicle is able to "catch back" the optimal reference position if it gets delayed for any reason, for example during acceleration phases. This is probably why the final time is very close to the optimal one.

3 Assignment III

In the third assignment it is required to load an empty Gazebo world with `turtlebot waffle pi` and some objects, then exploit the sensors of the robot to track one of these objects (a red sphere) and use a feedback control to collide with it.

Environment setup First of all, we need to spawn the required objects, i.e. a red sphere with radius 0.1 m which is located 6 m on the left of the robot and a purple sphere with radius 0.2 m located 6 m in front of the robot. This is done by using the dedicated class for controlling Gazebo objects in MatLab. Code and result (*figure 15*) are shown below.

```
%% Spawn stuff in Gazebo

% Create matlab object to communicate with gazebo

gazebo=ExampleHelperGazeboCommunicator;

% Add balls to gazebo

ball1=ExampleHelperGazeboModel('Ball1');
addLink(ball1,'sphere',0.1,'color',[200 48 48 1]); % color: #c83030
spawnModel(gazebo,ball1,[0 6 0.1]) % 6 meters on the left

ball2=ExampleHelperGazeboModel('Ball2');
addLink(ball2,'sphere',0.2,'color',[200 0 103 1]); % color: #c80067
spawnModel(gazebo,ball2,[6 0 0.1]) % 6 meters in front
```

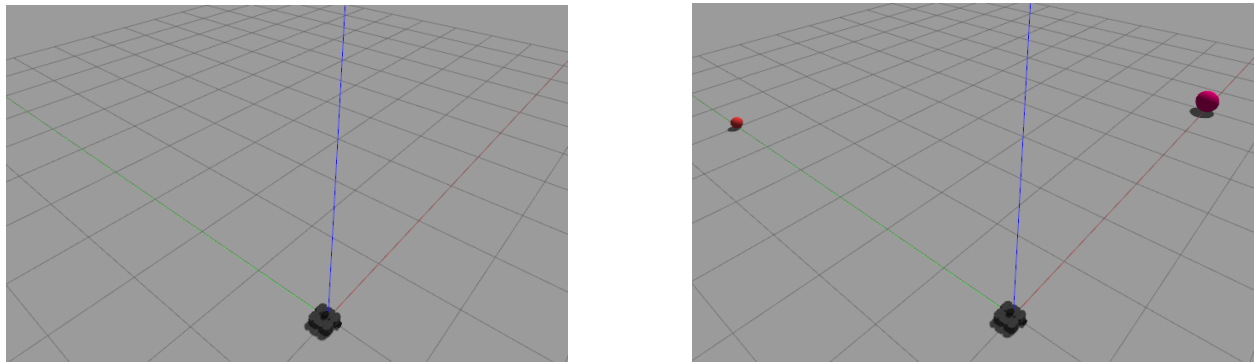


Figure 15: Environment setup for assignment III

Ball tracking and feedback control Simulink was used to subscribe to laser scan, odometry and camera data of the turtlebot (*figure 16*). Laser scan data are exploited to stop the vehicle and the simulation once the target has been hit, while odometry is used to plot the robot trajectory at the end.

To detect the correct ball, also taking into account for shadows and reflections, raw data from the camera are converted from RGB to HSV and a simple processing in this domain is performed, as shown in *figure 17*. Then, the target is tracked (blob analysis) using the dedicated blocks of the Computer Vision Toolbox (*figure 18*). Notice that a check is set such that when no targets are detected, the vehicle starts turning around at constant yaw rate until the red sphere does not appear. The feedback logic consists in a simple PID control that takes the error $e = u - u_{center}$, where u is the horizontal position (in terms of pixels) of the centroid of the detected ball in the camera frame, while u_{center} is the center of the camera frame itself. Basically it is telling the turtlebot to "look straight to the target".

The outcome of the simulation is presented in *figure 19* (click [here](#) for the animation): centroid position plot shows that the turtlebot is able to "lock" onto the target in more or less 0.7 seconds, with a typical second-order response. Notice that velocities were limited to $|v| < 0.26$ m/s and $|\omega| < 2.86$ rad/s, according to turtlebot `waffle pi` specifics.

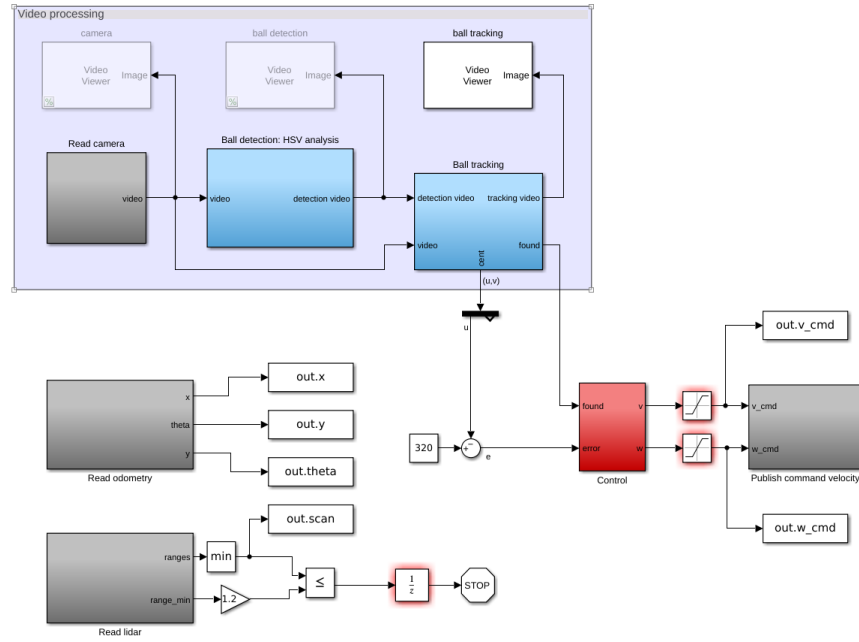


Figure 16: Simulink model of assignment III

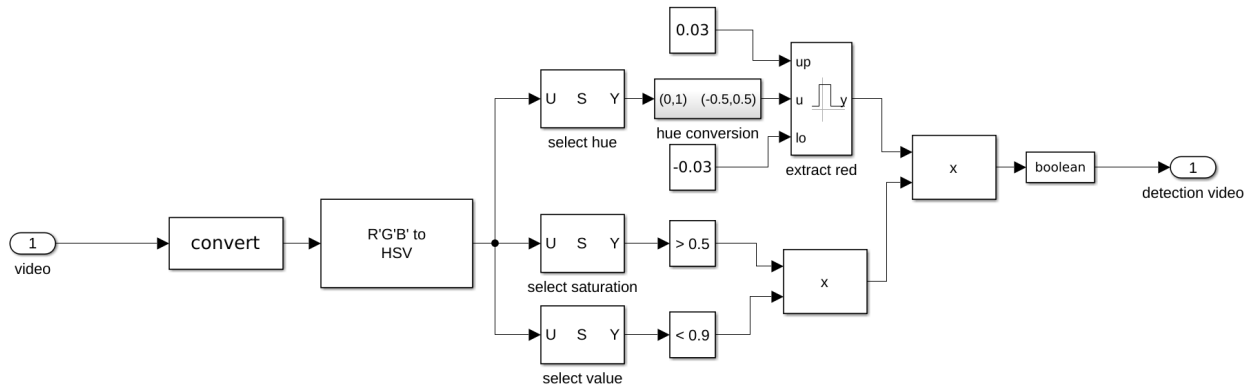


Figure 17: HSV detection block

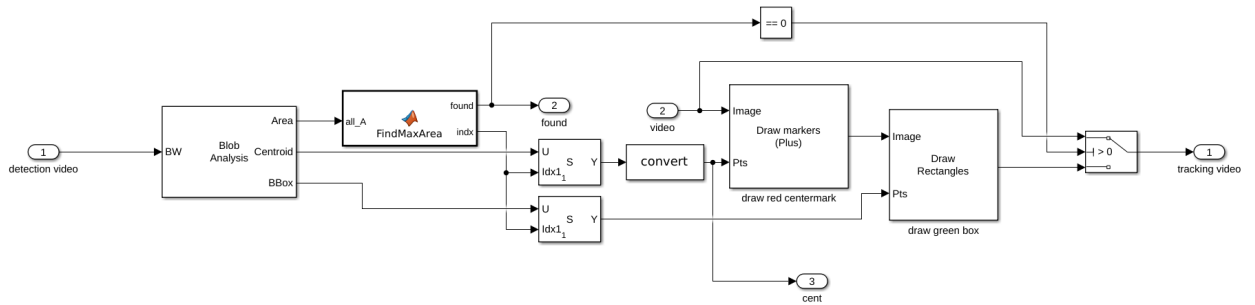


Figure 18: Target tracking block

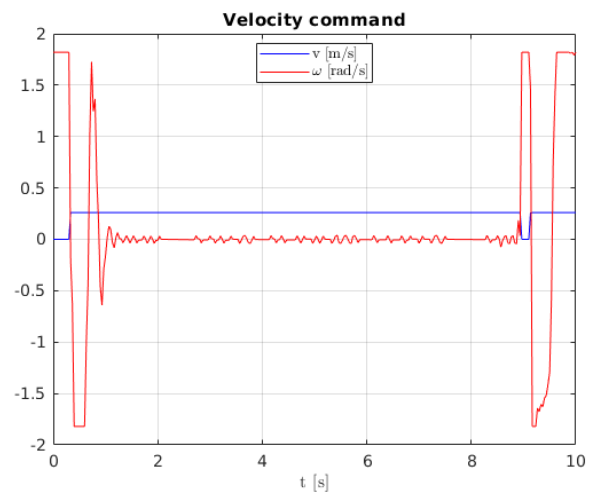
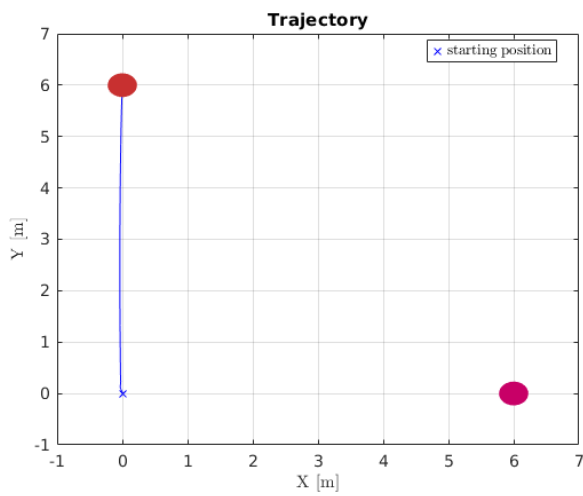
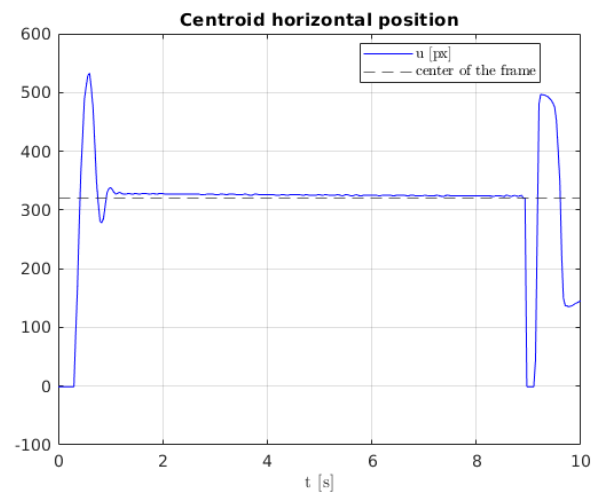
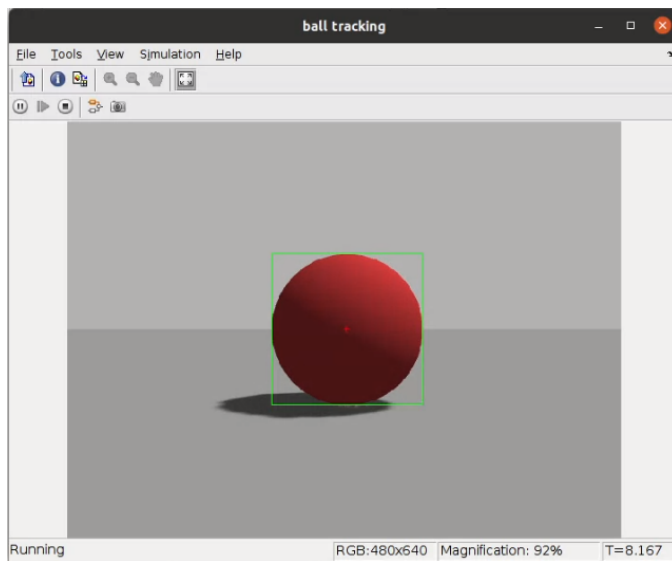


Figure 19: Simulation results of assignment III

4 Assignment IV

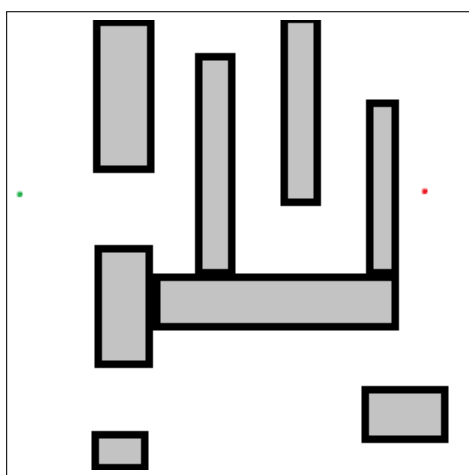
The task of this assignment is to write some well-known path searching algorithms used for motion planning. In particular, it is asked to start from a provided MatLab code which implements Dijkstra (a version that only "moves" in horizontal or vertical directions) and modify it to:

- be also able to "move" in diagonal directions;
- obtain A* algorithm (only moving in horizontal and vertical directions);
- obtain A* algorithm (also moving in diagonal directions).

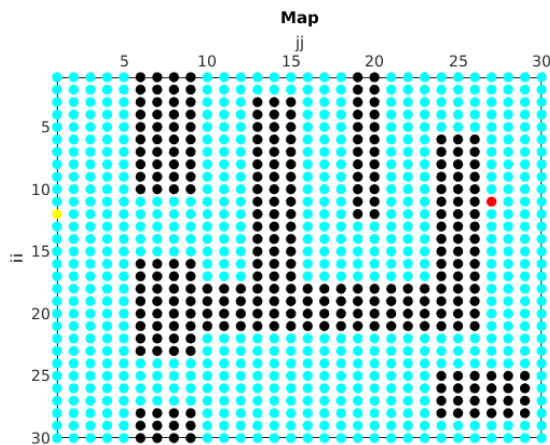
The resulting codes are then to be tested on four different maps.

4.1 Map elaboration

Dijkstra and A* are graph-search algorithms that are able to find the best path from a start position to a goal position in a given graph, which is an object made of "positions" (nodes) and "allowed connections" between nodes (edges). In our case the graph is a simple 2D grid of points (nodes), representing a bi-dimensional space in x and y, and finding the best path means finding the fastest (in terms of distance) way to navigate from the starting node to the goal node. For this reason, first it's important to understand how we can describe this graph in order to make the algorithm operate on it.



(a) Initial png map



(b) Elaborated BW map

Figure 20: Example of map processing for grid-based path searching

Nodes The definition of the nodes is immediate. We have a N-by-M png image, so N·M pixels, that represents the map we want to move on (from now referred as "png map"; an example is provided in *figure 20a*): green and red pixels are the start and goal positions, black pixels are obstacles, gray pixels are uncertain areas (interpreted as inaccessible), while white pixels are free positions. This map is loaded in MatLab and:

1. converted in HSV, so that start and goal positions are found and saved. Then all red and green pixels are turned white (as free nodes);
2. rescaled as needed, from N-by-M to n-by-m (goal and start coordinates are rescaled as well);
3. converted in gray-scale and then converted in binary form, where 1 is associated to free nodes and 0 to obstacles (from now, "BW map").

The outcome of this process is shown in *figure 20b*, where obstacles are painted in black, free nodes are cyan, while start and goal nodes are yellow and red. In this example the map was rescaled from a 300x300 png image to a 30x30 binary grid.

Edges Building the edges is a little bit more complicated. The idea proposed in the provided code is to build a "G" matrix that contains all the information about how nodes are connected. In particular, after having associated each (ii, jj) pixel of the BW map to a linear index i (or j), then G is a $N \cdot M$ -by- $N \cdot M$ matrix where the (i, j) element is:

- 0, if $i = j$ (same node) and that node is free;
- a , if nodes i and j are connected and free (a is the cost (distance) from node i to node j);
- -1 , otherwise.

Before proceeding, it's worth to mention a couple aspects.

- To implement Dijkstra (or A*) that works also with diagonal edges, the only difference is in the definition of this matrix G: in that case, in fact, node in position (ii, jj) will not only be connected to nodes $(ii + 1, jj)$, $(ii, jj + 1)$, $(ii - 1, jj)$, $(ii, jj - 1)$, with cost (distance) $a = 1$, but also to "diagonal adjacent nodes" $(ii + 1, jj + 1)$, $(ii + 1, jj - 1)$, $(ii - 1, jj + 1)$, $(ii - 1, jj - 1)$, with cost $a = \sqrt{2}$. For this reason, the distinction between the two algorithms ("diagonal" and "non-diagonal") will not be considered in section 4.2.
- The approach of using a G matrix to store the information about the edges of the graph is extremely inefficient, both in terms of time and memory requirements. This gives a strong limitation on the dimension of the resulting BW map, which is the space where path searching algorithms eventually operate. For this assignment, the analysis was limited to 50x50 maps and the approach was not changed, however, a more efficient method (that was also applied in Project II) and further considerations are presented at the end of section 4.3.

4.2 Searching algorithms

The aim of this section is not to explain extremely into detail Dijkstra and A* algorithms, but to give a general idea on how they were implemented.

Dijkstra Three vectors are initialized:

- **dist**, whose $i - th$ element is the distance of the $i - th$ node from the start along a certain path;
- **parent**, whose $i - th$ element is the index of the parent-node of the $i - th$ node;
- **status**, whose $i - th$ element is the status of the $i - th$ node (1=to visit (alive set), -1 =not visited, 0=visited)

The start node is set as the first active node, and its children are inserted in the alive set. Then an iterative procedure starts, where all children of the active node are checked, updating their cost (distance from start) and parenting if needed, i.e. if their updated cost is lower than the current one. After all children have been checked, the current active node is set as "visited", thus removed from the alive set, and the lowest-cost node in the alive set is taken as next active node. Again, all children of the new active node are inserted in the alive set, unless they have been already visited. The iteration stops whenever there are no more nodes in the alive set or the goal node is reached as active node.

At the end of this process, if a solution was found, it is possible to reconstruct it "backwards" by starting from the goal node and looking at the parenting chain in the **parent** vector.

Pseudo code of the algorithm is given below (Algorithm 1).

A* The algorithm is exactly the same, with the only difference that now the next active node is selected from the alive set not only basing on the minimum distance from the starting point, but also taking into account for the distance from the goal position. This means that we are choosing the "most promising" node among the available ones, i.e. the one that is expected to "reach the goal before the others". This heuristic correction allows A* to find the same solution Dijkstra finds (or an equivalent one, in terms of cost), i.e. the optimal one, but in less (or at worst equal) time. Notice that euclidean distance from the goal is just one possibility for adjusting the choice of the next active node, and this contribution, in any case, is **not** influencing the cost itself of the node, but only the selection of *act_node*.

Pseudo code is exactly the same of Dijkstra, but the function *PICK_NEXT* will be modified as explained above.

Algorithm 1 Dijkstra

```
1: Input: mapBW, G, start, goal ▷ nodes, edges, start, goal
2: Output: length, path, nnodes ▷ length of the solution, solution path, number of visited nodes
3: Initialize dist, parent, status ▷ status is to_visit, visited, not_visited
4: Initialize act_node, child ▷ child are children of act_node
5: while to_visit ≠ ∅ and act_node ≠ goal do
6:   for i in child do
7:     if dist[i] > NEW_DIST(i) then ▷ check new cost through act_node
8:       dist[i] ← NEW_DIST(i)
9:       parent[i] ← act_node
10:    end if
11:  end for
12:  act_node ← PICK_NEXT(to_visit) ▷ next act_node: lowest cost in alive set
13:  child ← CHILDREN(act_node) ▷ find children of the active node
14:  status[act_node] ← visited
15:  for i in child do
16:    if i ∉ visited then
17:      status[child] ← to_visit
18:    end if
19:  end for
20: end while
21: path ← RECONSTRUCT_PATH(parent)
22: length ← dist[goal]
23: nnodes ← SUM(visited)
24: return path, length, nnodes
```

4.3 Results

All four algorithms ("simple" Dijkstra, "diagonal" Dijkstra, "simple" A*, "diagonal" A*) are tested on four provided maps, rescaled to 50x50, leading to the results in *figures 23,24,25,26*. Blue nodes are the visited ones, while red ones are the optimal path. The performances are instead reported in Tables 1,2,3,4.

According to theory, we can see that both Dijkstra and A* always find the best solution: this may be different in terms of shape (due to the priority used to check the nodes), but has always the same (shortest) length possible. Moreover, as expected, A* is able to do that in less time (i.e. by checking less nodes), due to its heuristic logic, and this is particularly significant in "open" maps where the path from start to goal is "quite clean from obstacles", as for example in map 2 (*figure 24*). Finally, it's possible to clearly visualize the "pick next" logic of A* by looking at the last picture (the one on the bottom right, showing "diagonal" A*) in *figures 23,24,25,26*, or also from the animations: the algorithm is always trying to "push" towards the goal.

Concerning the diagonal movements, it's obvious that, if they are allowed, the solution path is in any case shorter (or equal) due to simple geometrical considerations.

Note on edges definition for map processing As anticipated before, the use of the previously defined matrix *G* to store information about the edges of the graph is not an efficient strategy. In fact, for a *n*-by-*m* map, the dimension of *G* will be *nm*-by-*nm*, leading quickly to enormous memory requirements for the calculator (for example, more or less 60 GB of RAM are needed just to process a 300-by-300 map). This limits remarkably the resolution of the maps we can work with, which is particularly relevant for these grid based approaches, since they are able to find the best solution only in terms of the provided map. This means that the output path is optimal only inside the given grid, and not necessarily also for the "real" environment, in particular if the resolution is low.

Another strategy was developed (but not applied to this assignment) to build the map in a more efficient way. Without entering too much in detail, the implementation was the following:

1. nodes are created as usual, returning *mapBW*;
2. all variables are transferred to the GPU, which is much faster for big matrix operations;

3. map is divided into blocks, each one of them as big as possible (compatibly with memory specifics);
4. these blocks are taken two at the time, by two for loops (parallelization can be used to further increase the speed);
5. a distance matrix is computed between the two blocks: it contains the distances between all nodes in the considered blocks; `pdist2` function of Machine Learning Toolbox is exploited to efficiently do that;
6. all distances higher than $\sqrt{2}$ are set to zero, so that sparse matrices can be exploited;
7. all non-null elements of distance matrix are checked to store the connected nodes in the output matrix "edges"

This procedure returns the list of nodes and edges that can then be fed to Navigation Toolbox `navGraph` function. This function creates a graph object which is then used to run graph-searching algorithms directly implemented in MatLab (`plannerAStar`, ecc...). The use of GPU, parallelization, sparse matrices and built-in functions improves the performances by a big margin, allowing in particular to build arbitrarily big maps. An example of map 1 obtained with this strategy is shown in *figure 21*, where the solution found with the built-in MatLab A* planner is highlighted in red; as we can see, it is the same solution found in *figure 23d*.

Finally, an even more efficient approach is shown in *figure 22*, where MatLab `plannerAStarGrid` function, that handles automatically edges generation, is used on map 1 (5000x5000).

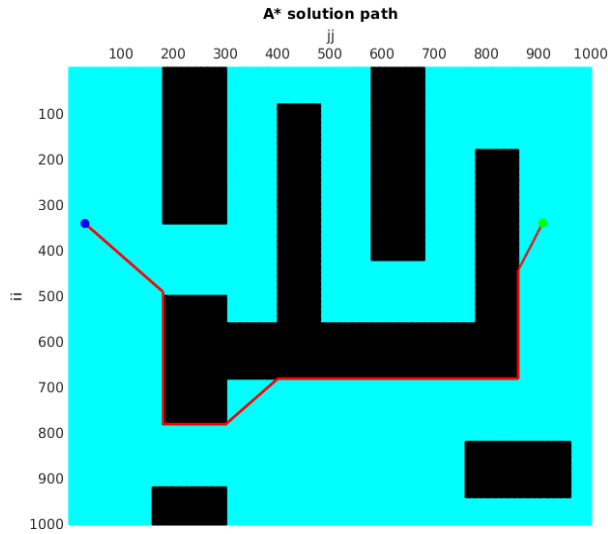


Figure 21: Map 1 and solution developed with the improved elaboration algorithm + `plannerAStar` on a 1000x1000

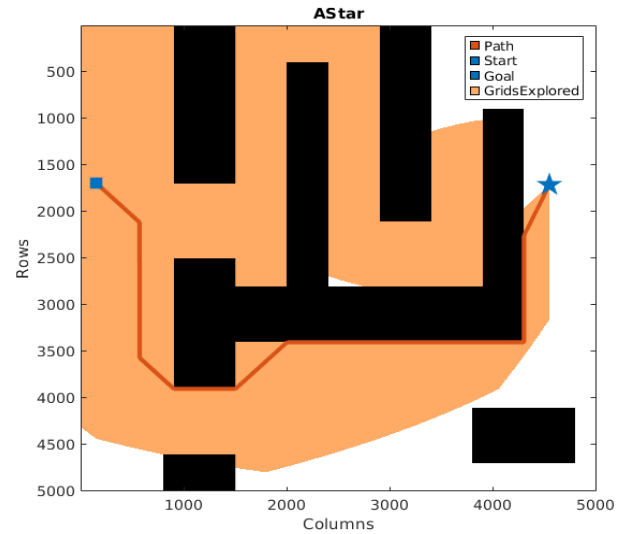
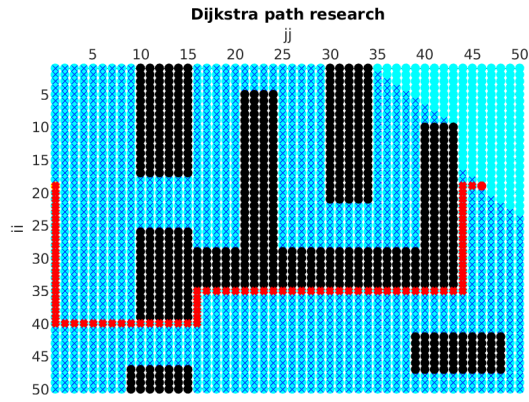
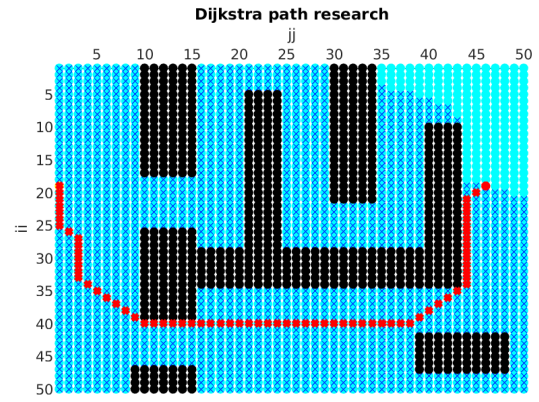


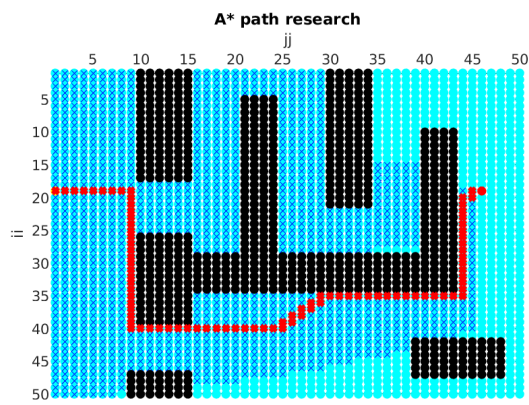
Figure 22: Map 1 and solution developed with `plannerAStarGrid` on a 5000x5000



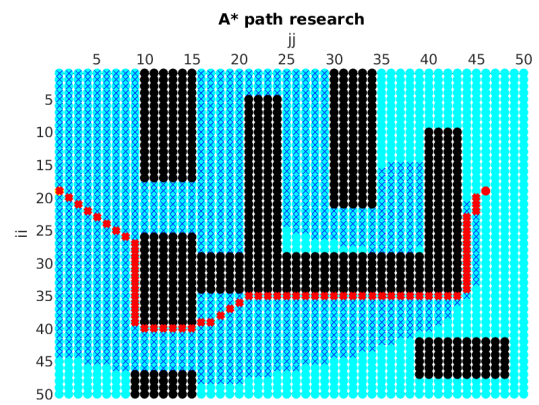
(a) [animation](#)



(b) [animation](#)



(c) [animation](#)

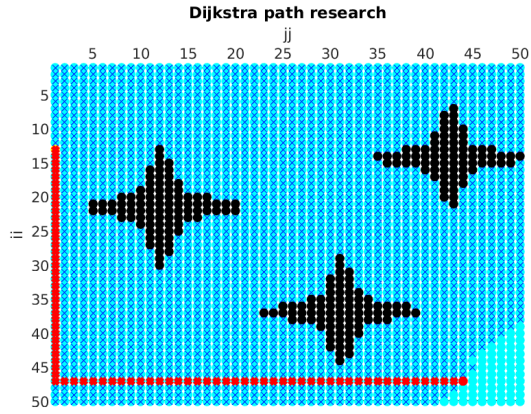


(d) [animation](#)

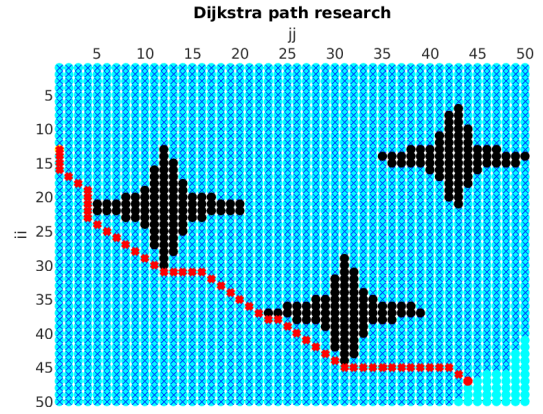
Figure 23: Map 1

Algorithm	Optimal path length	Number of evaluated nodes
"simple" Dijkstra	87	1599/1781
"diagonal" Dijkstra	77.0416	1604/1781
"simple" A*	87	1285/1781
"diagonal" A*	77.0416	1184/1781

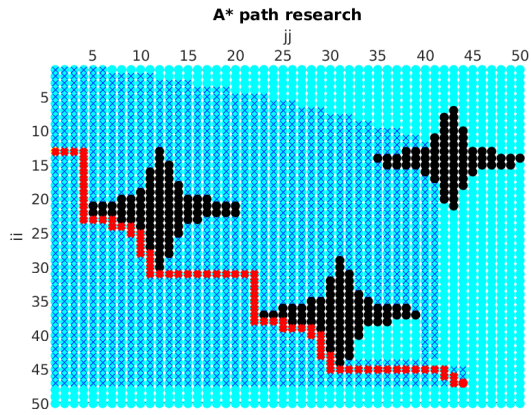
Table 1: Performances of the different algorithms on map 1



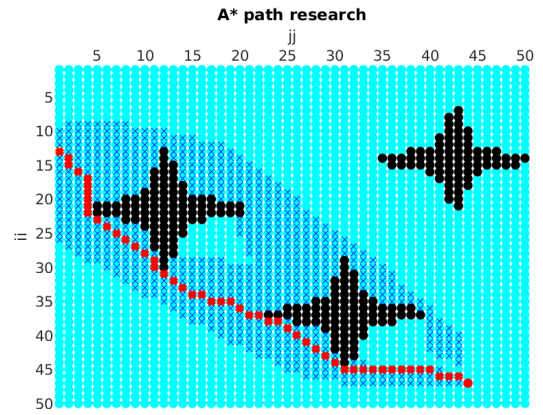
(a) [animation](#)



(b) [animation](#)



(c) [animation](#)

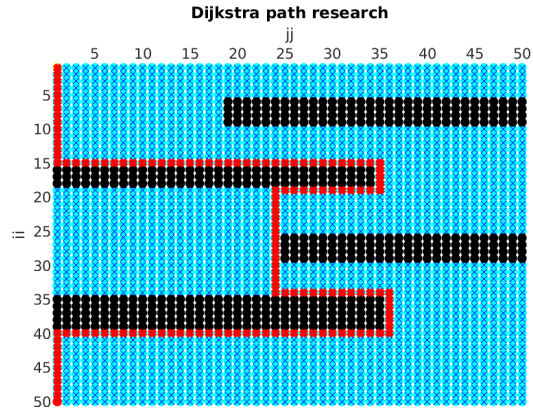


(d) [animation](#)

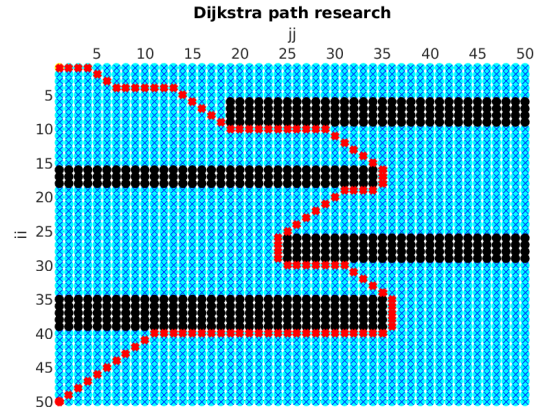
Figure 24: Map 2

Algorithm	Optimal path length	Number of evaluated nodes
"simple" Dijkstra	77	2194/2256
"diagonal" Dijkstra	61.1838	2215/2256
"simple" A*	77	1528/2256
"diagonal" A*	61.1838	690/2256

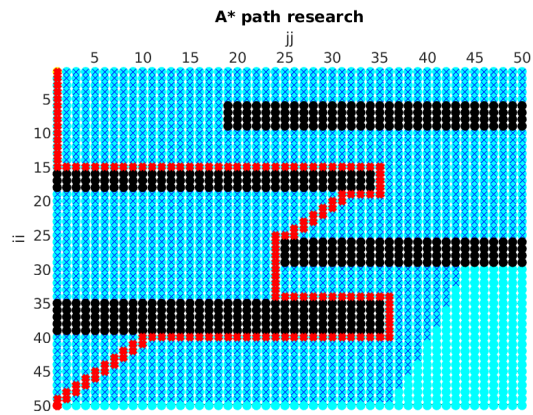
Table 2: Performances of the different algorithms on map 2



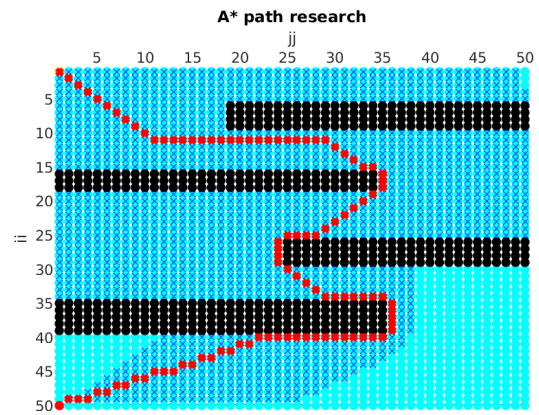
(a) [animation](#)



(b) [animation](#)



(c) [animation](#)

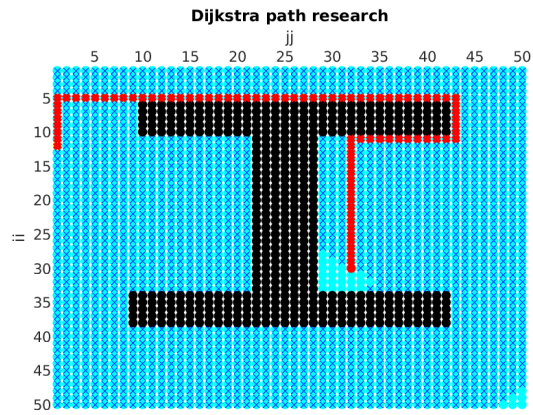


(d) [animation](#)

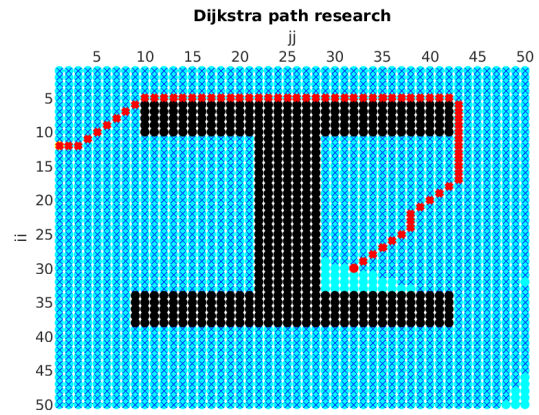
Figure 25: Map 3

Algorithm	Optimal path length	Number of evaluated nodes
"simple" Dijkstra	141	1991/1991
"diagonal" Dijkstra	117.569	1991/1991
"simple" A*	141	1739/1991
"diagonal" A*	117.569	1580/1991

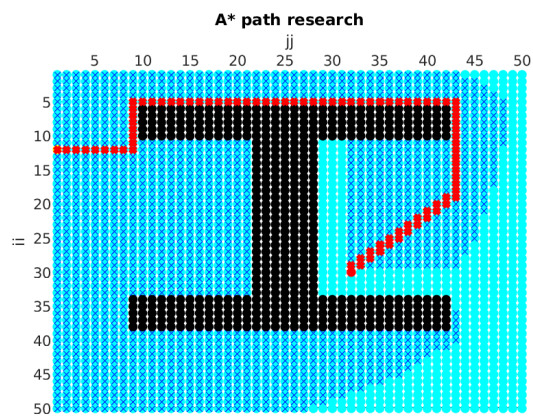
Table 3: Performances of the different algorithms on map 3



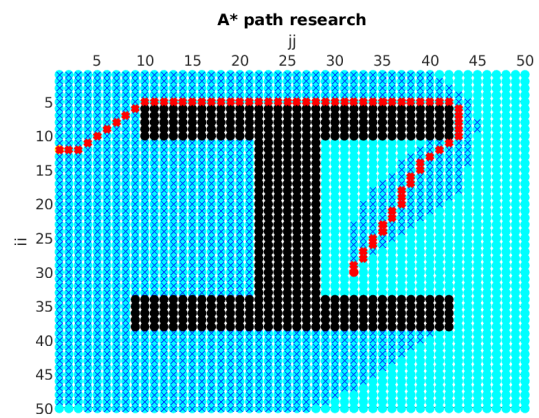
(a) [animation](#)



(b) [animation](#)



(c) [animation](#)



(d) [animation](#)

Figure 26: Map 4

Algorithm	Optimal path length	Number of evaluated nodes
"simple" Dijkstra	85	1976/2004
"diagonal" Dijkstra	73.8701	1966/2004
"simple" A*	85	1547/2004
"diagonal" A*	73.8701	1319/2004

Table 4: Performances of the different algorithms on map 4

5 Assignment V

A finite state model (FSM) of a car moving through a parking gate is provided in Simulink. It is required to:

1. understand how it works;
2. create the finite state model of the parking gate according to:

- max angular speed: $\omega_{max} = \pm 4 \text{ deg/s}$
- max angular acceleration: $\alpha_{max} = \pm 1 \text{ deg/s}^2$

and making so that when a car approaches the gate, it opens to 90 deg; then, once the car has passed, it closes to 0 deg and waits for the next car to arrive.

Afterwards, as an addition, the whole FSM model was modified in order to also handle the situation of a car arriving during the closing phase of the gate. In this case, the barrier should stop "mid air" and immediately open again, to let the car pass (a "telepass-like" behaviour).

5.1 Main part

Vehicle FSM The provided finite state model of the vehicle is shown in *figure 27a*. Inputs are position x (a random number with mean=-15), *start* and *gate_up*. At the beginning (initial state) the car is "waiting" in its random initial position, until the user sets *start* == *true*. So the vehicle starts advancing (x increases with constant speed v) up to $x = 0$ (position of the gate), where it stops and enters the status *waiting*. Once the gate is up (*gate_up* == *true*), the car advances again at constant speed and its status is not *waiting* nor *passed*, because it is transitioning under the gate. After a distance *gate_end* the transition is considered completed, so the status is set to *passed* and the vehicle simply advances with constant speed for 10 more seconds before disappearing while another car spawns in the random initial position (initial state). The status of the car (*passed*, *waiting* or neither of the two) is given as output.

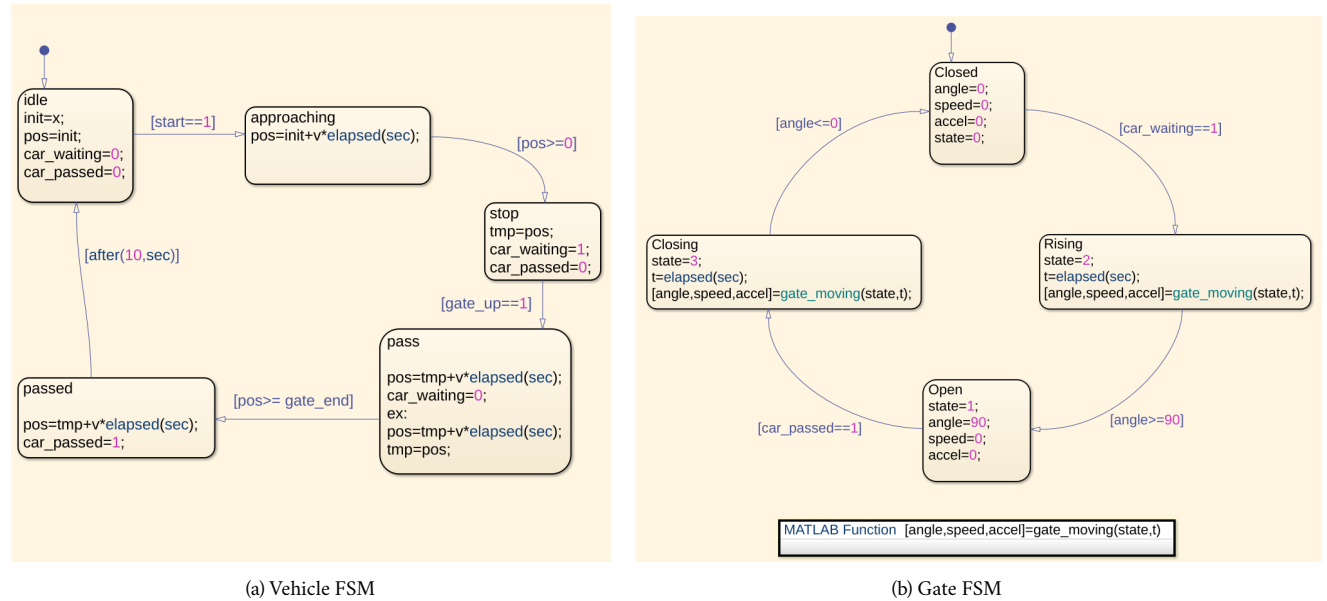


Figure 27: Finite state models in assignment V

Gate FSM The finite state model of the gate (*figure 27b*) takes as input the status of the car and outputs the state of the gate, that can be *closed* (0), *gate_up* (1), *rising* (2) or *closing* (3). In the initial state the gate is *closed*, with null angle, speed and acceleration. When a car arrives (car status is *waiting*), the gate transitions to the *rising* phase: angle, angular speed and acceleration change according to the trapezoidal velocity profile (*figure 28*) defined inside the function

gate_moving, until the gate has opened completely ($angle = 90^\circ$). At this point the state of the barrier is outputted as *gate_up*, allowing the car to pass. Once the vehicle completed its transition, i.e. *car_passed* == *true*, the gate can simply close similarly to how it opened before, going back to the initial state and waiting for another car to arrive.

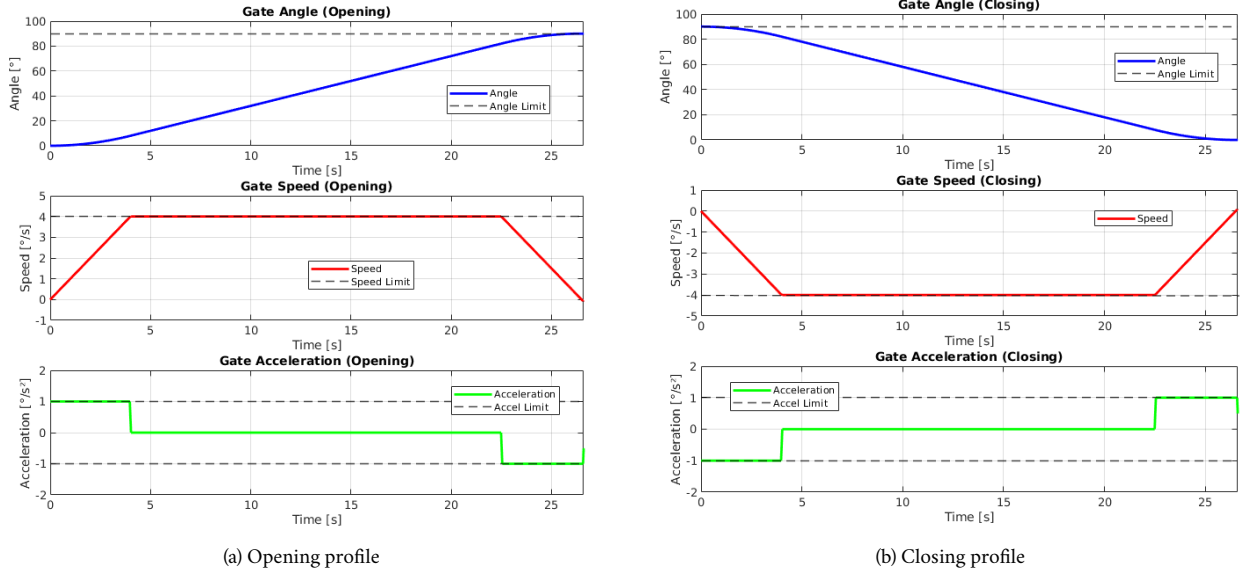


Figure 28: Trapezoidal velocities profiles for the movement of the gate

Notice that in this case, if another car arrives during the *closing* phase, the barrier will first go down to 0° , making the car wait, and then it will start opening again. Results for 100 seconds of simulations are shown in figure 29.

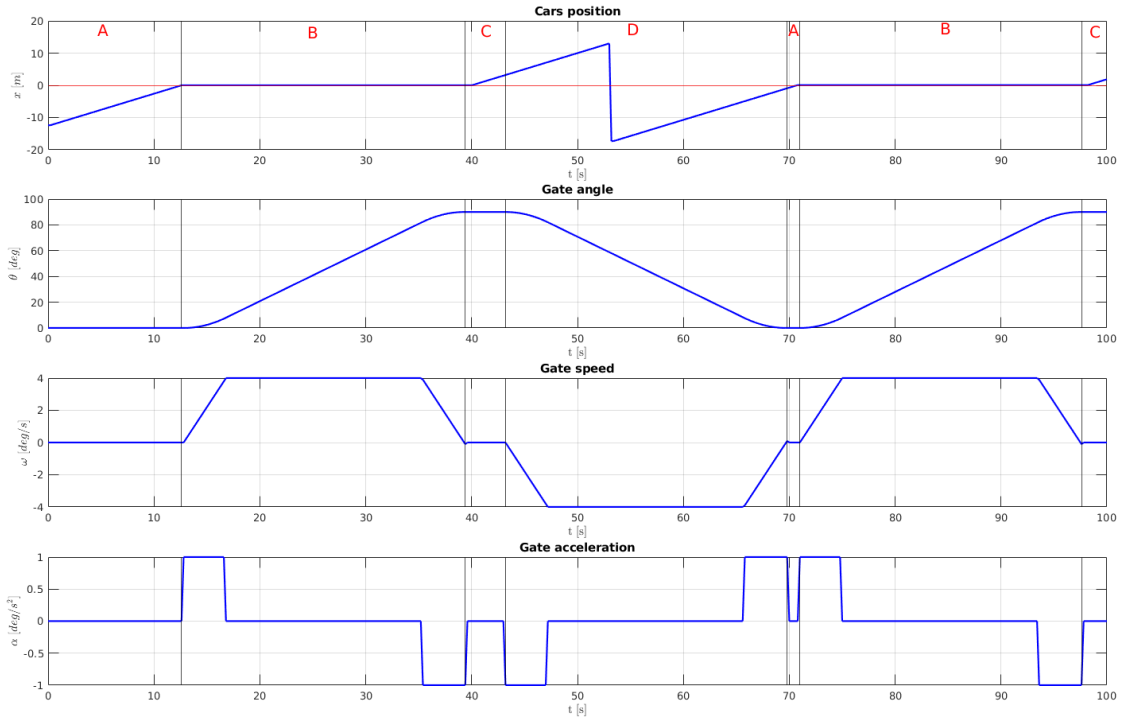


Figure 29: (A) car approaching, gate down; (B) car waiting for the gate to open, gate opening; (C) gate open, car passing through; (D) car passed (another car spawning), gate closing

5.2 Extra

In this last part, as anticipated, the FSM of the gate was modified in order to immediately open if a car arrives during the *closing* state, so that the waiting time of the vehicle is minimized. The only difference with respect to the previous implementation is that there is an additional condition, which can be activated during the *closing* phase if a car arrives (*car_waiting* == *true*), that simply switches the control of the gate back to *opening*, by imposing the "opening" velocity profile in *figure 28a* starting from proper initial conditions.

Notice that the *gate_moving* function was removed by simply defining the velocity profiles directly inside the model (but this does not affect the working principle of the algorithm).

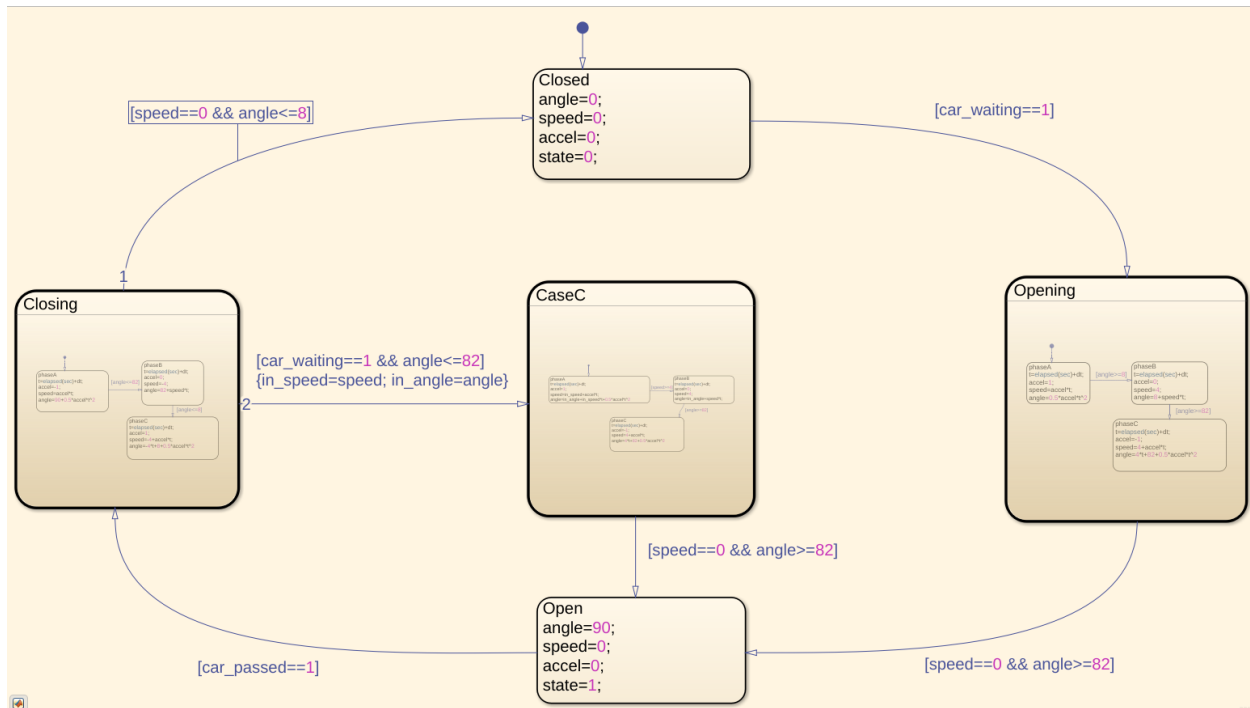
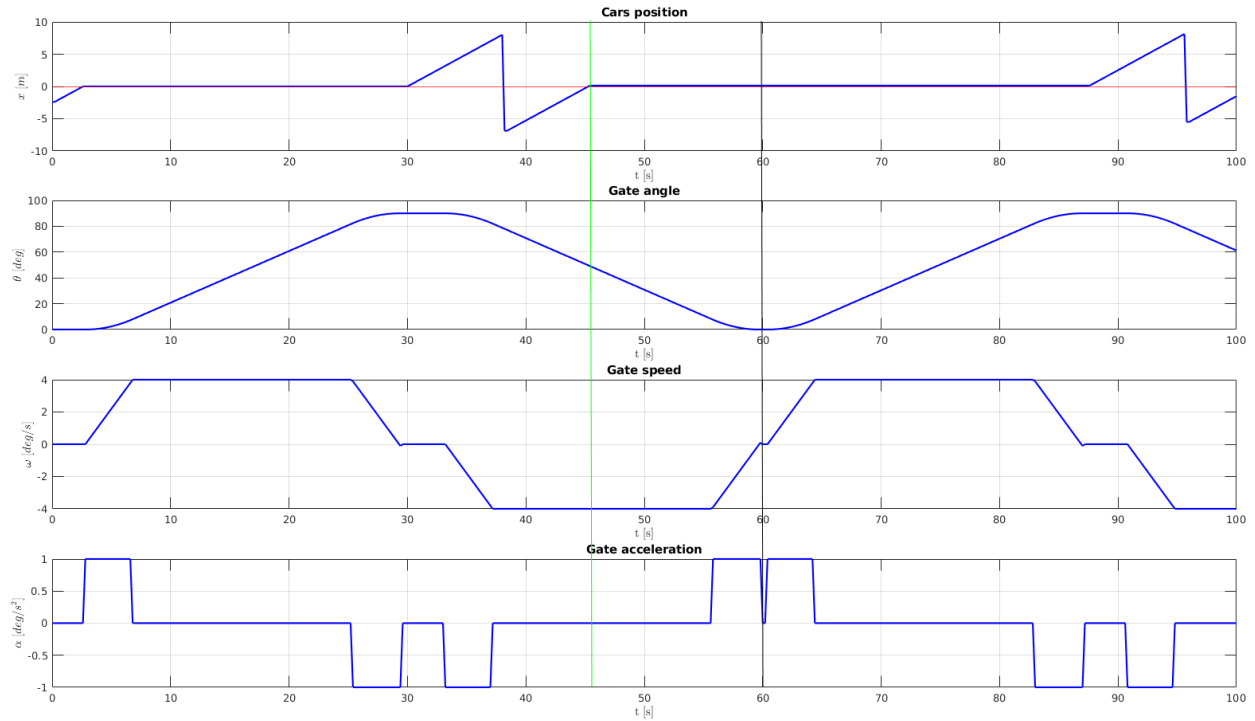
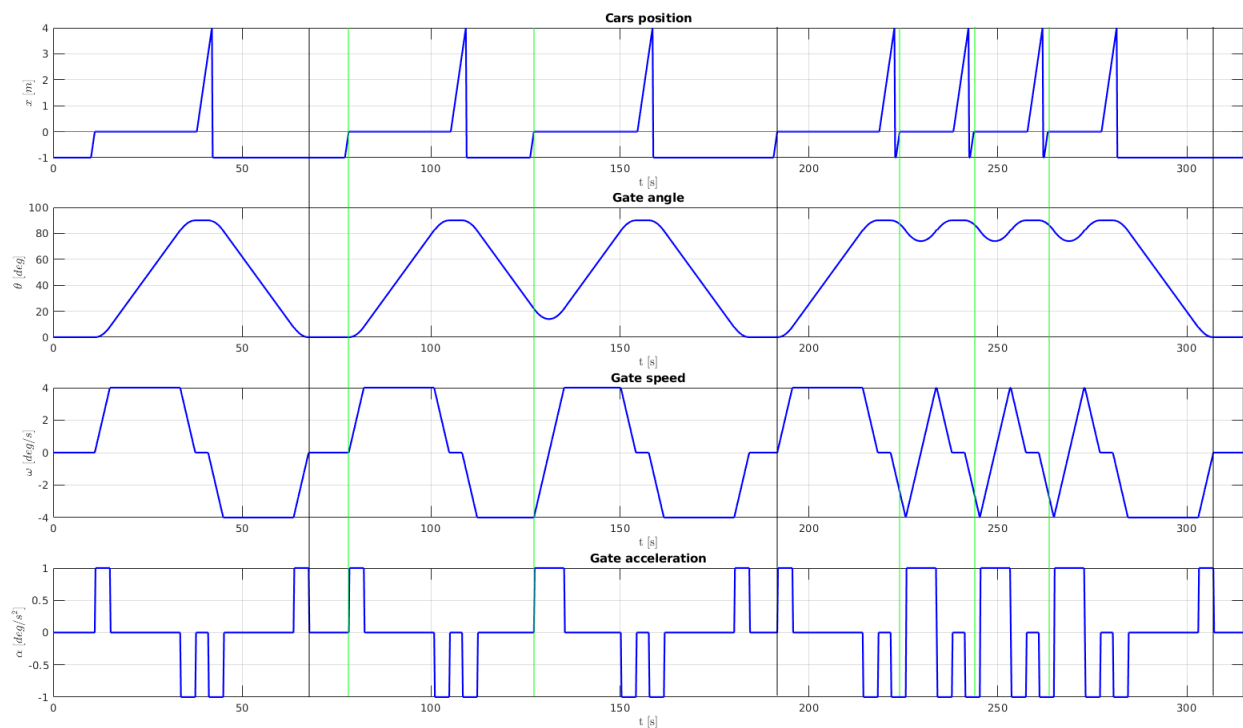


Figure 30: Improved FSM of the gate

This improved model was tested by letting another car arrive while the gate was closing (spawn distance and spawning time were both reduced). Results are plotted in *figure 31*, along with a comparison with the previous FSM handling the same situation.



(a) Simple gate control. When the car arrives at the gate (green line), the barrier is still closing; however, we need to wait until it is completely down (black line) to start opening again: time between the green and the black line is "wasted"



(b) Improved gate control. When the first car passes, the gate is able to close completely (first black line); then after the second vehicle, a third one arrives while the barrier is still closing (second green line): as we can see, there is an immediate inversion of the control of the bar (opening); then it is able to close completely again (second black line); finally, three cars arrive in a very short time (last three green lines), but the gate immediately opens as soon as a vehicle starts waiting

Figure 31: Green lines mark when the car reaches the gate, black lines indicate that the gate closes completely