POLITECNICO
MILANO 1863

Autonomous Vehicles - Assignments Report

a.y. 2024-2025
Lorenzo Vignoli
lorenzo.vignoli@asp-poli.it
252085

Contents

1 Assignment I

The first assignment involves processing a `.bag` file containing data from a simulation performed with ROS and Gazebo. The simulation features a `turtlebot3 waffle_pi` navigating a `turtlebot3 world` environment. The bag includes specific topics, from which the tasks are to estimate the minimum distance to obstacles and analyze the `/cmd_vel` commands sent to the robot. Topics are:

- `/clock`: simulation time information.
- `/imu`: data from IMU sensor.
- `/odom`: odometry data, including the position and velocity.
- `/scan`: laser scan data for detecting obstacles.
- `/tf`: transformation between coordinate frames (robot, its components, and the world).

1.1 Part A

Trajectory and Minimum Distance from Obstacles. Using `_odom` information, specifically the data in the *Pose* field, the robot's trajectory is computed (*figure* 1). From the `/scan` data, the minimum distance to obstacles is determined (*figure* 2). It is worth noting that the `/scan` message provides values in the range from 0.12 m to 3.5 m, with an angular resolution of 1° (0.0175 rad) across a full field of view (360°).

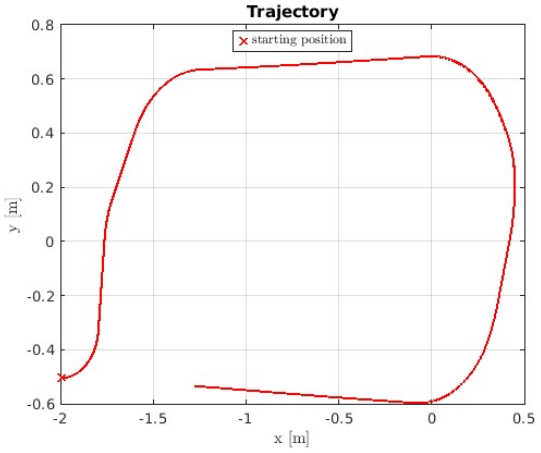


Figure 1: *trajectory*

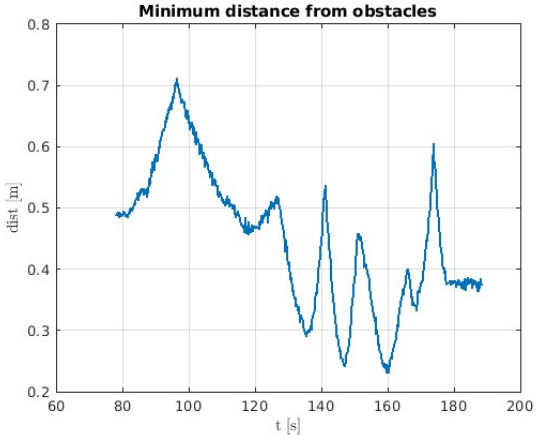


Figure 2: *minimum distance from obstacles*

Notice that in *figure* 2, the signal starts at a time greater than one minute, indicating the transitional period between the start of the simulation and the beginning of the recording. Moreover, by analyzing the data from the `/scan` topic and the corresponding time intervals, they result to be provided at 5 Hz.

Velocity command estimation. Since the `/cmd_vel` topic is deleted from the bag, the velocity information in the `/odom` topic can be processed instead. These data have a time frequency of 30 Hz.

Data in *figure* 3 are filtered using a median filter for both linear velocity and yaw rate to reduce spikes and measurement noise. However, considering that the command input likely originates from

the `turtlebot3_teleop` keyboard control, with increments of 0.01 m/s for linear speed and 0.1 rad/s for yaw rate, the signals are rounded to match these step changes. Moreover, the measured velocity signals align with this assumption: final results are shown in *figure* 4.

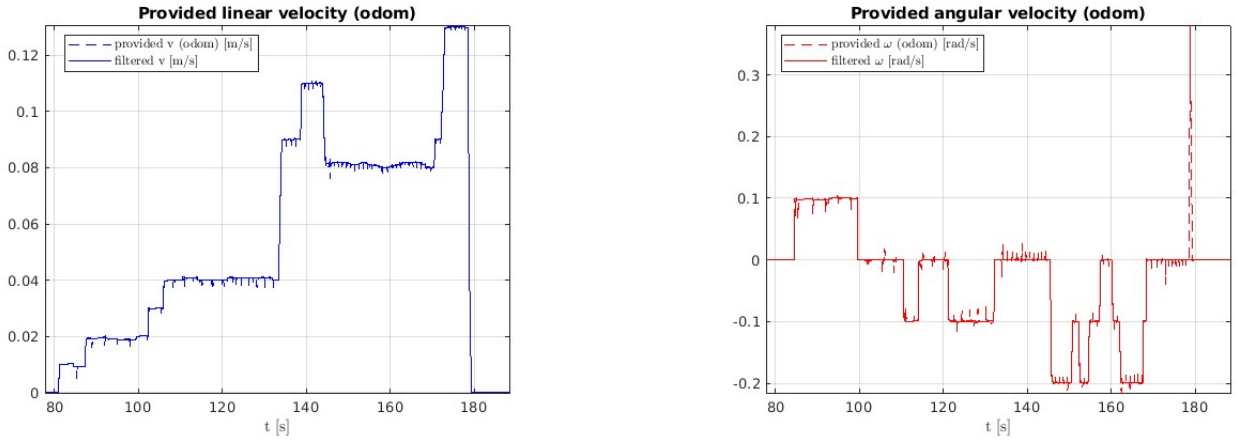


Figure 3: *measured linear velocity and yaw rate*

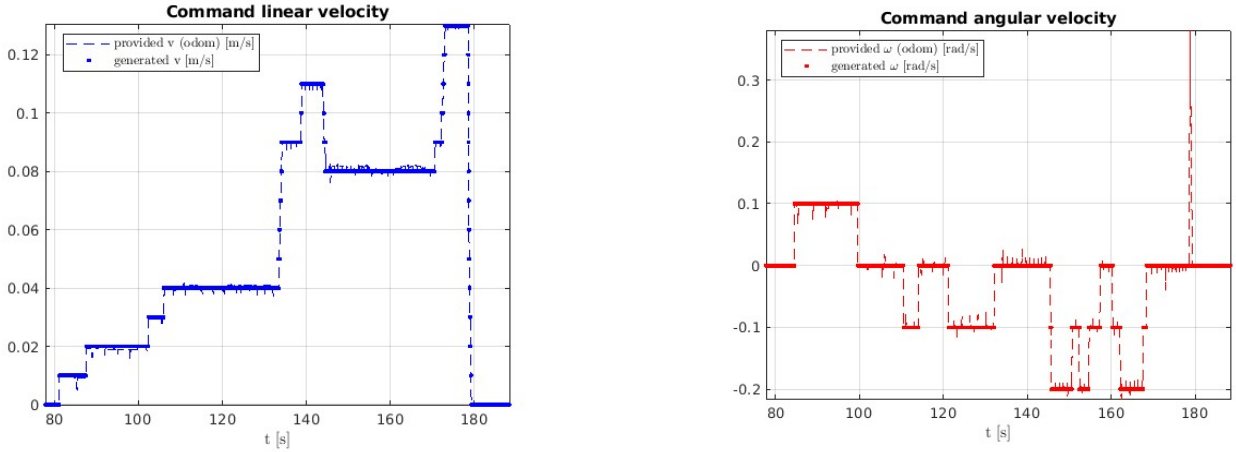


Figure 4: *evaluated command velocity*

Plotting the environment shown by LIDAR. Information from `/scan` and `/odom` can be used jointly to reconstruct the environment where the vehicle is moving. Different sampling frequencies (5 and 30 Hz) are adjusted by simple linear interpolation, while all non-infinite scanning ranges are retained. These represent the valid LIDAR information, coupled with the corresponding angles. Accordingly, the corresponding odometry information (in terms of (x, y) position) of the robot is kept, enabling the definition and plotting of the retrieved points in *figure* 5.

The apparent thickness of the walls and external obstacles in the reconstructed environment is likely due to measurement errors and noise. LIDAR sensors can produce inaccuracies, while odometry errors can accumulate over time, causing slight misalignment in the mapped points. The interpolation used to synchronize data from `/scan` and `/odom` can also introduce discrepancies. Consequently, small measurement deviations can result in a scattered point distribution around the obstacles, producing misleading wall thickness.

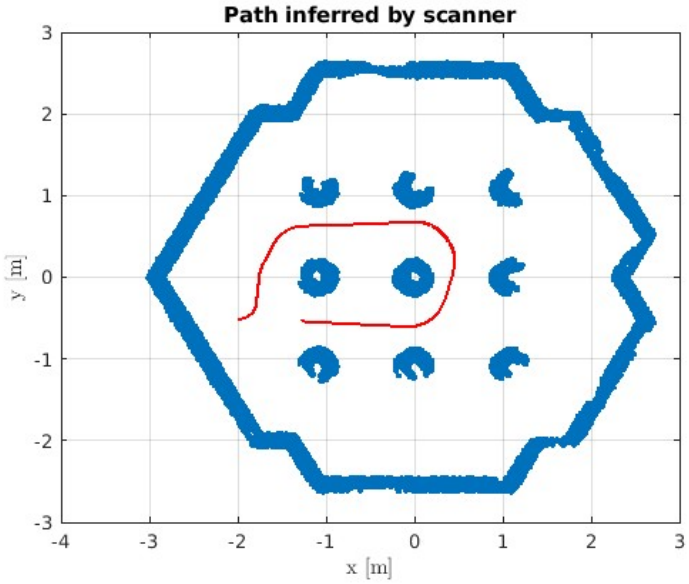


Figure 5: *reconstructed environment, using LIDAR and odometry*

1.2 Part B

The estimated commands (*Linear.X* for linear velocity and *Angular.Z* for yaw rate) are tested by publishing them in ROS through Simulink, using the `/cmd_vel` topic. Command sequences are provided to ROS as time series, maintaining a frequency of 30 Hz. Simulation data are recorded in a `.bag` file and processed as described in subsection 1.1.

The results are shown in *figures* 6 and 7. The measured velocities closely match the estimated ones, except for some noisy spikes, demonstrating effective reverse engineering. The trajectory is also highly similar (*figures* 8 and 9), as are the minimum distances from obstacles. It is worth noting that velocity data are extracted from a `data.mat` file recorded in Simulink, ensuring perfect alignment; on the other hand, distance data originate from two separate `.bag` recordings, resulting in slight misalignment.

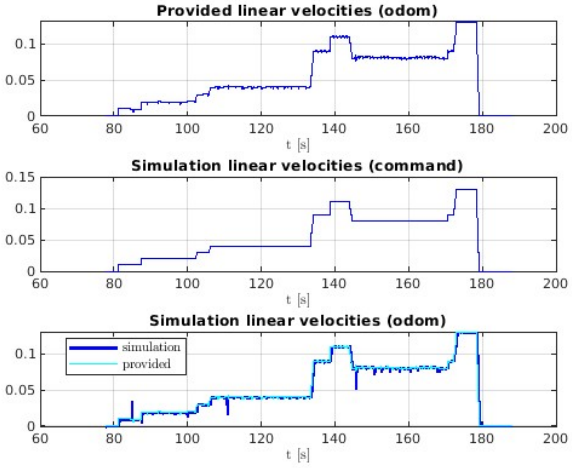


Figure 6: *linear velocities*

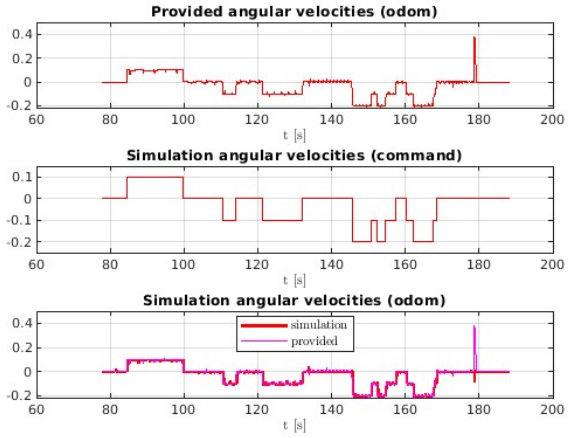


Figure 7: *yaw rates*

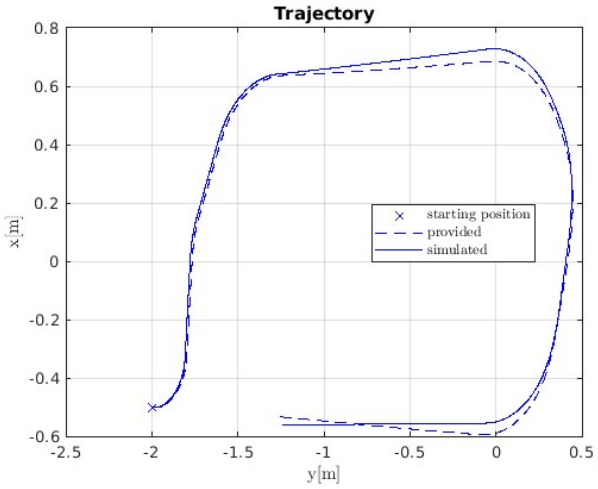


Figure 8: *trajectories*

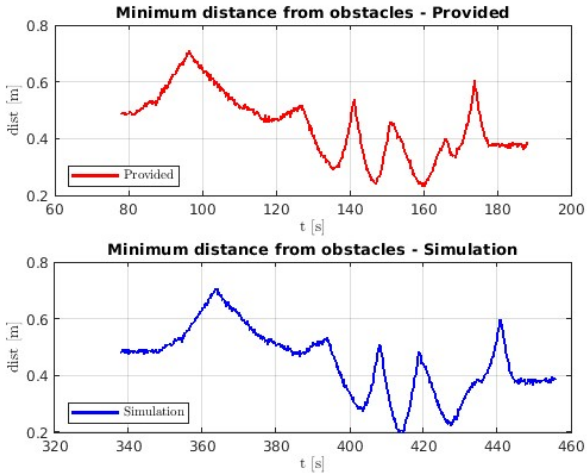


Figure 9: *minimum distances*

2 Assignment II

The second assignment involves defining a feedback control to move the `turtlebot3 burger` robot from its initial position through a list of provided waypoints, stopping at the last one (*subsection* 2.1). Waypoints are initially loaded in MatLab/Simulink, while in *subsection* 2.2, the same waypoints are eventually provided through an external source, specifically a Python node. Finally, a posture regulation algorithm, described in *subsection* 2.3, is implemented to "park" the robot at each waypoint.

Waypoint	x **[m]**	y **[m]**	θ **[rad]**
1	0	0	0
2	5	0	$\pi/2$
3	5	5	$5\pi/4$
4	-5	-5	$\pi/2$
5	-5	5	0
6	0	0	0
7	3	3	$3\pi/4$
8	-3	0	$3\pi/2$
9	0	-3	$\pi/4$
10	3	0	$\pi/2$
11	0	0	$3\pi/2$

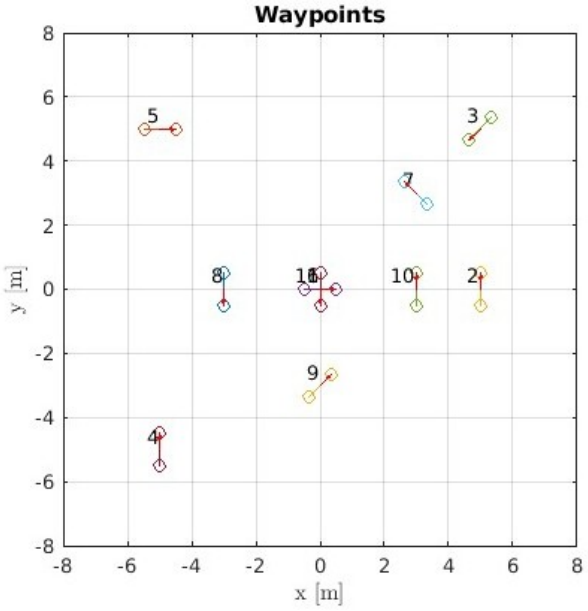


Figure 10: *waypoints*

2.1 Basic feedback control

The following control constraints are taken:

- max linear speed $|v_{max}| = 0.2\,m/s$
- max yaw rate $|\omega_{max}| = 0.4\,rad/s$

Since the waypoints are defined in terms of both (x, y) position and orientation θ, a three-step approach is used:

1. Select the next waypoint and continuously monitor the distance to it using `/odom`. The velocity is controlled through a proportional (P) controller with saturation, defined as:

$$v_{\text{cmd}} = \text{sat}\big(k_v \cdot \Delta d, v_{\text{max}}\big),$$

where v_{cmd} is always positive (robot moving forward). When the robot is within a distance of $d_{\text{max}} = 0.01\,\text{m}$ from the waypoint, its velocity is set to zero to allow proper heading adjustment.

2. Consider the direction between the robot and the waypoint, and adjust the yaw rate accordingly using a P controller. Here, θ_{ref} is the direction to the waypoint, and θ_{tbot} is the `turtlebot3` heading:

$$\omega_{\text{cmd}} = \text{sat}\big(k_{\omega 1} \cdot (\theta_{\text{ref}} - \theta_{\text{tbot}}), \omega_{\text{max}}\big),$$

3. When the robot is closer than d_{max} to the waypoint, the yaw rate control law changes to align the robot's heading with the desired waypoint orientation. The desired direction is θ_{wpoint}, and the control law becomes:

$$\omega_{\text{cmd}} = \text{sat}\big(k_{\omega 2} \cdot (\theta_{\text{wpoint}} - \theta_{\text{tbot}}), \omega_{\text{max}}\big),$$

Similar to velocity control, ω_{cmd} is set to zero when $|\theta_{\text{wpoint}} - \theta_{\text{tbot}}| < \theta_{\text{max}}$, with $\theta_{\text{max}} = 0.05\,\text{rad}$.

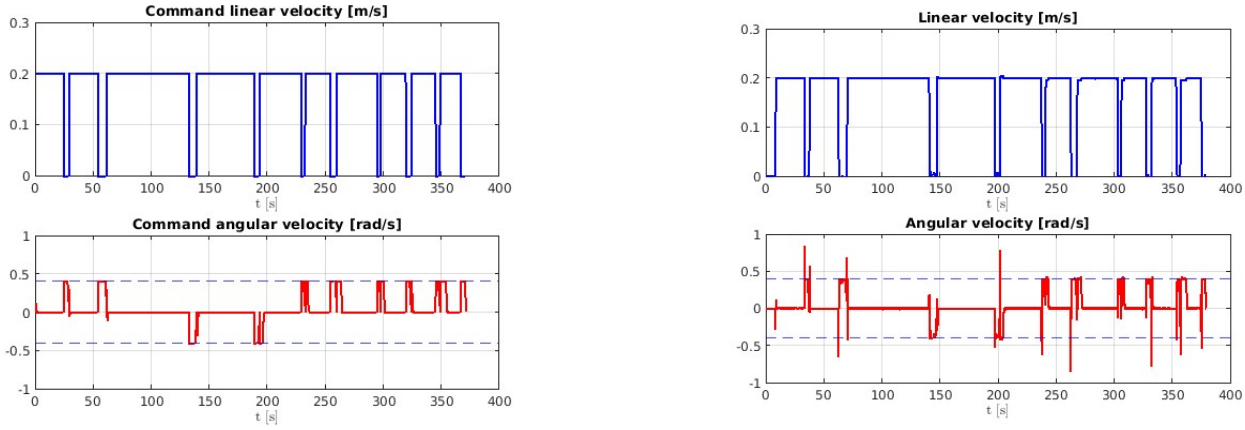


Figure 11: *command and measured velocities - basic feedback control*

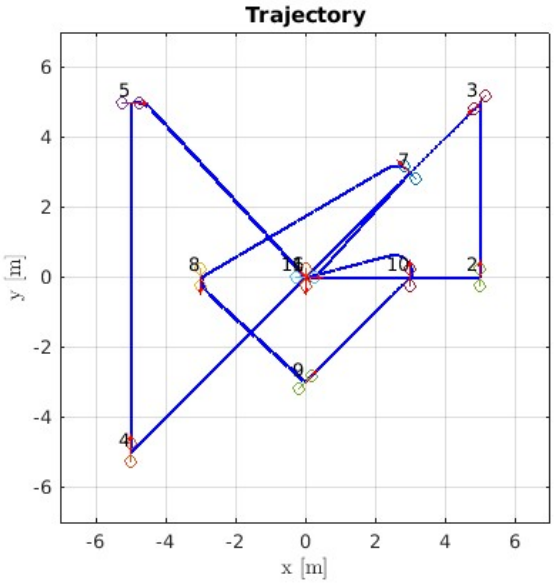


Figure 12: *trajectory - basic feedback control*

The vehicle determines that it has reached a waypoint when both its linear speed and yaw rate are zero. It then selects the next waypoint and moves towards it. In this implementation, waypoints are directly provided by MatLab as a matrix, from which the system selects the next target.

The measured velocity closely replicates the command velocity (*figure* 11), resulting in a smooth and accurate trajectory. This allows the robot to successfully navigate through all the waypoints without significant deviations (*figure* 12).

Since the task is relatively simple, inertial effects do not significantly impact the system's performance. Moreover, unlike in other situations, the strategy of stopping at each waypoint is effective here. However, in more complex scenarios, such as those explored in Project II, these aspects may become critical and require further consideration.

2.2 Extra A

Keeping the same control logic, waypoints are provided by an external publisher. The chosen solution involves a Python node named *publisher_Assignment_II*, which publishes the entire waypoint list as individual *PoseWithCovariance* messages. This node initializes a ROS publisher that sends one message every two seconds. For each waypoint, a *PoseWithCovariance* object is created, with the following fields populated in the `covariance` array:

- `covariance[0]`: x coordinate
- `covariance[1]`: y coordinate
- `covariance[2]`: θ (heading)

Additionally, two metadata fields are included:

- `covariance[3]`: the waypoint index
- `covariance[4]`: a flag set to 1 if it is the last waypoint, or 0 otherwise

```python
#!/usr/bin/env python
import rospy
from geometry_msgs.msg import PoseWithCovariance
import math

waypoints = [
    [0, 0, 0],
    [0, 0, 0], # automatically ignores first two messages

    [0, 0, 0],
    [5, 0, math.pi / 2],
    [5, 5, 5 / 4 * math.pi],
    [-5, -5, math.pi / 2],
    [-5, 5, 0],
    [0, 0, 0],
    [3, 3, 3 / 4 * math.pi],
```

```
    [-3, 0, 3 / 2 * math.pi],
    [0, -3, math.pi / 4],
    [3, 0, math.pi / 2],
    [0, 0, 3 / 2 * math.pi]
]

def publisher_pose():
    pub = rospy.Publisher('geometry_msgs/PoseWithCovariance', PoseWithCovariance,
                          queue_size=50)
    rospy.init_node('publisher_Assignment_II', anonymous=False)
    rate = rospy.Rate(0.5)

    number_of_points = len(waypoints)

    if not rospy.is_shutdown():
        for i, point in enumerate(waypoints):
            msg = PoseWithCovariance()
            msg.covariance[0] = point[0]
            msg.covariance[1] = point[1]
            msg.covariance[2] = point[2]
            msg.covariance[3] = i + 1
            if i == number_of_points - 1:  # check if it's the last point
                msg.covariance[4] = 1
            else:
                msg.covariance[4] = 0

            pub.publish(msg)
            rate.sleep()

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

Metadata are used in the Simulink block shown in *figure* 13. A subscriber reads the *PoseWithCovariance* messages in real time and passes them to a MatLab function, which stores them in a *waypoints* matrix along with metadata, provided the point is not already recorded. When the fifth element of the message is 1, the *done* flag becomes true, the robot moves, and the waypoints are then selected in the same way as described in *subsection* 2.1. The only difference is that Simulink knows the length of the waypoints list through `covariance[4]` and not in advance, allowing sequences of different lengths to be published.

In *figures* 14 and 15, the speeds and trajectory are obviously the same as in *subsection* 2.1, with the only difference that velocities do not start immediately but with some delay, due to both the time required by the Python publisher and the one needed to manually synchronize Simulink with the Python node.

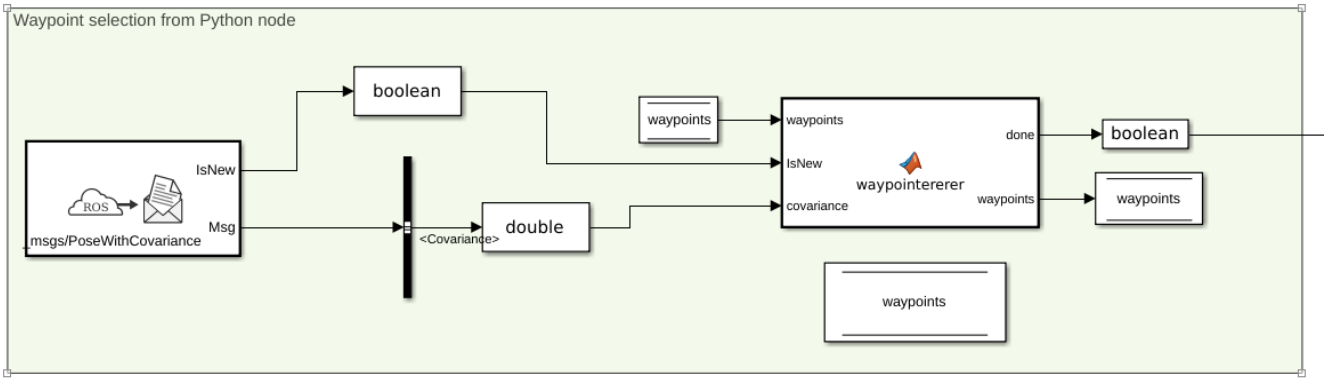


Figure 13: *waypoints selector from PoseWithCovariance*

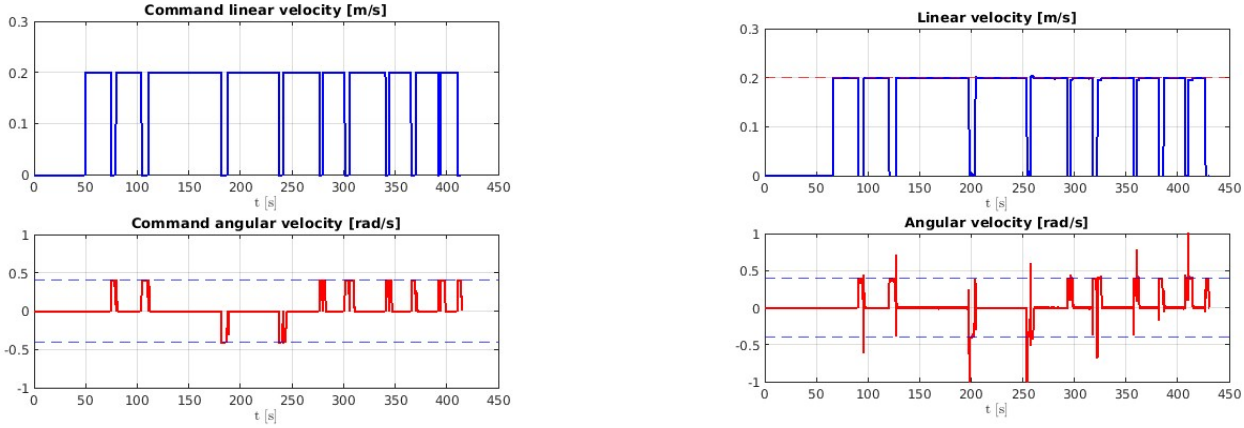


Figure 14: *command and measured velocities - extra A*

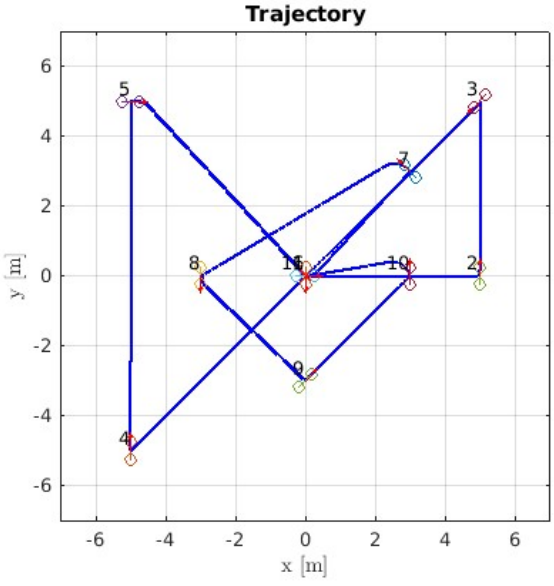
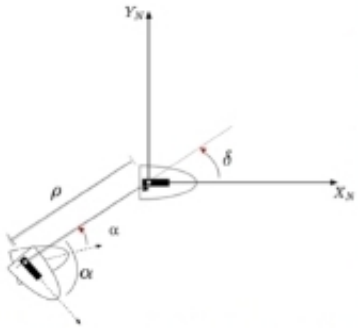


Figure 15: *trajectory - extra A*

2.3 Extra B

A nonlinear feedback control logic is implemented, inspired by a closed-loop posture regulation algorithm (Aicardi Michele et al., "Closed-loop steering of unicycle-like vehicles via Lyapunov techniques," *IEEE robotics and automation magazine*, 1995). Defining the following coordinate transformation:



$$\begin{cases} \rho = \Delta d \\ \alpha = \theta_{\text{ref}} - \theta_{\text{tbot}} \\ \delta = \theta_{\text{ref}} - \theta_{\text{wpoint}} \end{cases}$$

Figure 16: *Coordinate transformation (X_N aligned to the waypoint)*

All angles are between $-\pi$ and π. It must be noted that, since in the paper the goal is to stabilize the robot to the origin $(0, 0, 0)$ and thus $\delta = \theta_{\text{ref}}$, the parameters' equations have to be modified to reach $(x, y, \theta_{\text{wpoint}})$.

Consequently, the closed-loop control law is:

$$v = k_1 \cdot \rho \cdot \cos(\alpha) \tag{a}$$

$$\omega = k_2 \cdot \alpha + k_1 \cdot \frac{\sin(\alpha) \cdot \cos(\alpha)}{\alpha + 10^{-5}} \cdot (\alpha + k_3 \cdot \delta) \tag{b}$$

In *equation* (b), 10^{-5} is added to α for numerical stability. Moreover, since when $\rho < d_{max}$ the robot is very close to the waypoint and thus the computation of α could be affected by errors or instabilities, the term $\frac{\sin(\alpha)\cdot\cos(\alpha)}{\alpha}$ is directly replaced with its asymptotic value of 1 (valid for $\alpha \to 0$). The equation then becomes:

$$\omega = k_1 \cdot k_3 \cdot \delta \tag{b*}$$

As visible in *figure* 18, the trajectories are more complex, and the robot approaches the waypoint more smoothly. As in *subsections* 2.1 and 2.2, the linear velocity $v \approx v_{\max}$ for almost the entire path.

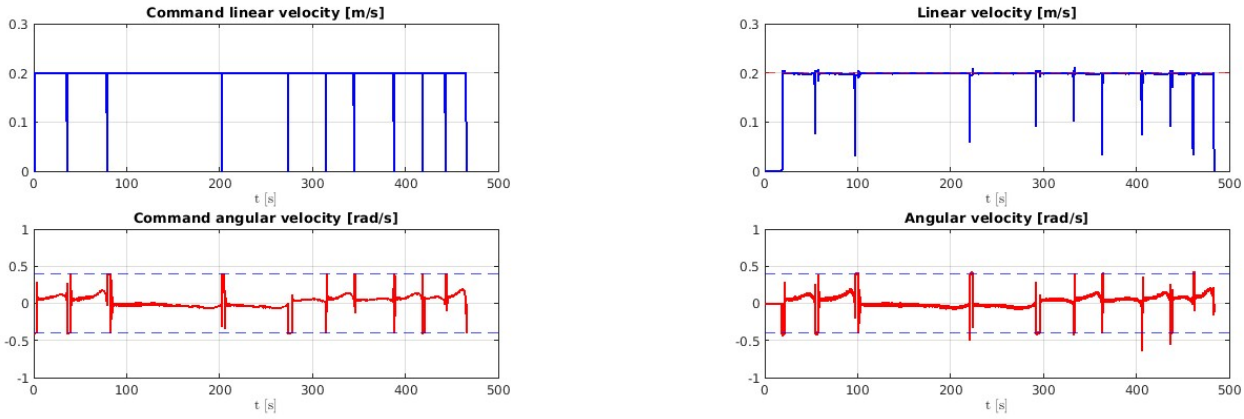


Figure 17: *command and measured velocities - extra B*

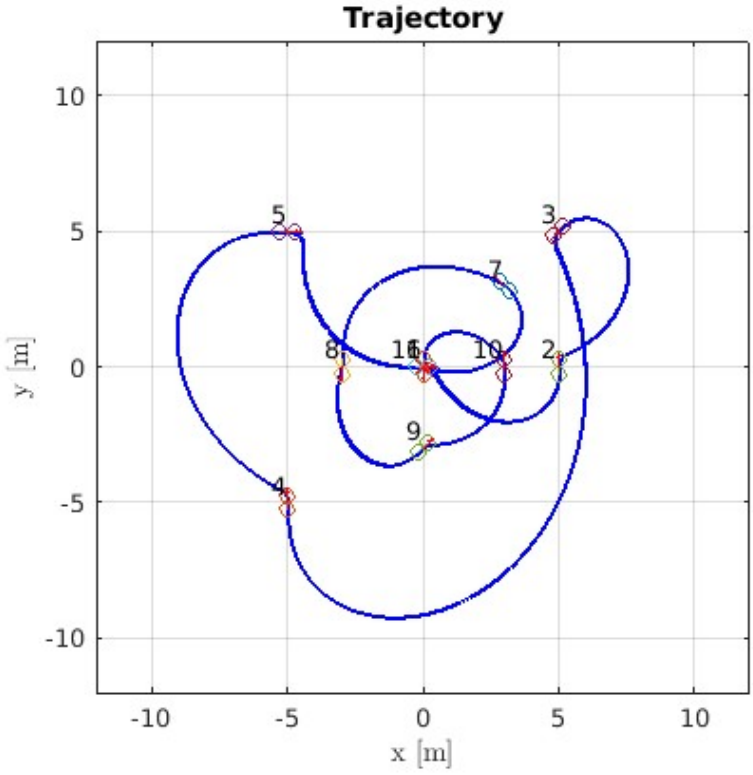


Figure 18: *trajectory - extra B*

3 Assignment III

The first assignment involves modifying an empty Gazebo world by adding some objects, using then the sensors of the `turtlebot3 waffle_pi` robot to track a red sphere (image processing through the camera), and employing a feedback control system to make the robot collide with it.

3.1 Environment setup and ball tracking

Environment setup. First, we need to spawn the required objects:

- *Red sphere*: radius of 0.1 m, located 6 m to the left of the origin. RGB: $(200, 48, 48)$.
- *Purple sphere*: radius of 0.2 m, located 6 m in front of the origin. RGB: $(200, 0, 103)$.

Additionally, to increase the complexity of the task, we add two more spheres:

- *Blue sphere*: radius of 0.4 m, located 6 m behind the origin. RGB: $(0, 0, 255)$.
- *Farther red sphere*: radius of 0.1 m (same size as the target item), located 8 m to the right of the origin. RGB: $(200, 48, 48)$.

```
% Communication object
gazebo = ExampleHelperGazeboCommunicator;

% Adding objects
redSphere = ExampleHelperGazeboModel("Ball");
red_color = [200, 48, 48, 255] * 1/255;
sphereLink1 = addLink(redSphere, "sphere", 0.1, "color", red_color);
spawnModel(gazebo, redSphere, [0, 6, 0]);

purpleSphere = ExampleHelperGazeboModel("Ball");
purple_color = [200, 0, 103, 255] * 1/255;
sphereLink2 = addLink(purpleSphere, "sphere", 0.2, "color", purple_color);
spawnModel(gazebo, purpleSphere, [6, 0, 0]);

blueSphere = ExampleHelperGazeboModel("Ball");
blue_color = [0, 0, 255, 255] * 1/255;
sphereLink3 = addLink(blueSphere, "sphere", 0.4, "color", blue_color);
spawnModel(gazebo, blueSphere, [-6, 0, 0]);

redSphereTest = ExampleHelperGazeboModel("Ball");
green_color = [200, 48, 48, 255] * 1/255;
sphereLink4 = addLink(redSphereTest, "sphere", 0.1, "color", red_color);
spawnModel(gazebo, redSphereTest, [0, -8, 0]);
```

An image of the environment setup is shown in *figure* 19.

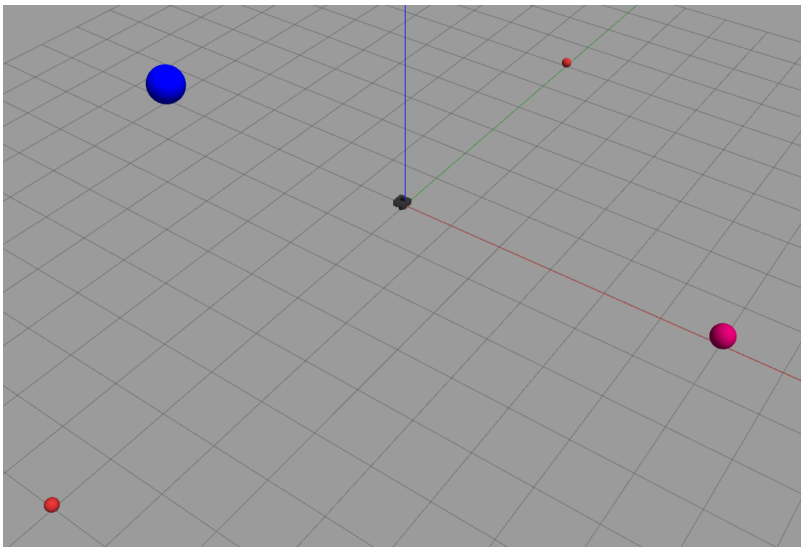


Figure 19: *environment setup*

Simulink is used to communicate with ROS by subscribing to the camera data to perform blob analysis and to identify the closest red sphere. Odometry data are also subscribed to, enabling feedback control for linear speed and yaw rate and allowing the plotting of the obtained trajectory. Information from the `/cmd_vel` topic is exchanged to control the robot's movement and to detect when the ball is hit (based on the discrepancy between `/cmd_vel` commands and `/odom` speed values).

To correctly detect the target ball, raw camera data are converted from RGB to HSV, selecting the appropriate values as follows: hue range from 0 to 0.1, and saturation between 0.6 and 0.8 (*figure* 20). This approach ensures that only red values are selected. However, since purple is relatively close to red in terms of hue, relying solely on hue could lead to misleading conclusions.

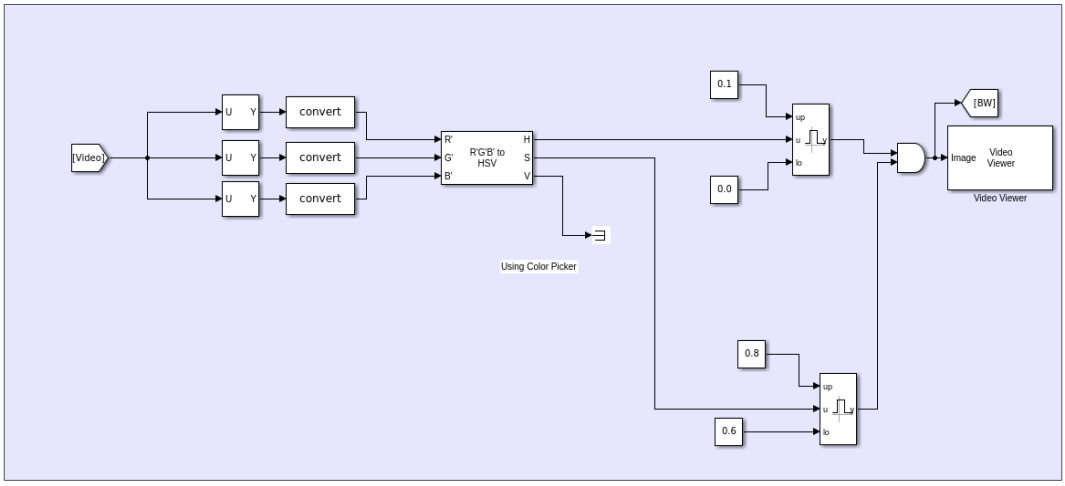


Figure 20: *RGB to HSV*

Ball tracking. The processing logic is as follows: blob analysis is performed on the binary (BW) image. To decide which ball to approach, the robot first executes a 360° spin to identify the largest red blob and then moves toward it. Since cameras lack depth information, the blob area

is used as the criterion for selecting the correct sphere. By performing the initial spin, the robot can discard the farther red sphere and choose the correct one (which, being initially to the left of the robot, is not visible to the camera at the start).

3.2 Feedback control

An articulated Simulink block enables controlling the robot following the sequence described in *algorithm* 1.

Algorithm 1 Red ball tracking feedback control

Step 1: Identify the largest red blob
Spin at maximum yaw rate
for each detected blob **do**
  Compute the area of the blob
  if blob area is the largest so far **then**
    Save blob direction (θ_{blob}) and area
  end if
end for
if robot is spinning and orientation is close to 0° **then**
  Completed 360° scan
end if

Step 2: Align with the blob
Maintain maximum linear velocity
Rotate with yaw_rate $= k \cdot (\theta_{\text{blob}} - \theta_{\text{robot}})$
Once aligned, move towards the red sphere

Step 3: Control to the target and crash detection
Maintain maximum linear velocity
Adjust the yaw rate to keep the red blob centered in the image (dimensions: 480×640), such that:

$$\text{yaw\_rate} = k \cdot (320 - x_{\text{blob}})$$

if `/cmd_vel` linear velocity is not null and `/odom` linear speed is null **then**
  Detected collision
end if

Results are shown in *figures* 21, 22, 23, and 24. Note that the velocities are constrained to $|v| < 0.2$ m/s and $|\omega| < 0.4$ rad/s.

As expected, the command linear velocity is initially null when the `turtlebot3 waffle_pi` is spinning to align with the correct direction. It then reaches the maximum allowed value. The command velocity never returns to zero afterward, as the simulation stops when the corresponding `/odom` linear speed drops. On the contrary, the angular velocity starts at 0.1 rad/s (a quarter of the maximum value, as defined in Simulink's control logic), quickly reaches the maximum, and remains constant (except for spiky noise in the measurements). It eventually decreases when aligning to the correct orientation, achieving a smooth adjustment thanks to a proportional (P) controller. This smooth behavior is reflected in the final trajectory.

The trajectory itself is straightforward: in *figure* 23, the positions of the spheres are shown along with their relative sizes and the robot's path. The blob center position, shown in *figure* 24, is considered null when no blob is detected. Consequently, the y-coordinate remains consistently at 240 pixels when one of the red spheres is detected, and zero otherwise. The x-coordinate, on the other hand, moves within the image (from 0 to 640 pixels) as the spheres are detected during scanning. During *Step 2* and *Step 3*, the x-coordinate increases steadily from 0 to 320 pixels, showing that the robot is turning counter-clockwise, and then stabilizes at 320 pixels, indicating that it is moving straight.

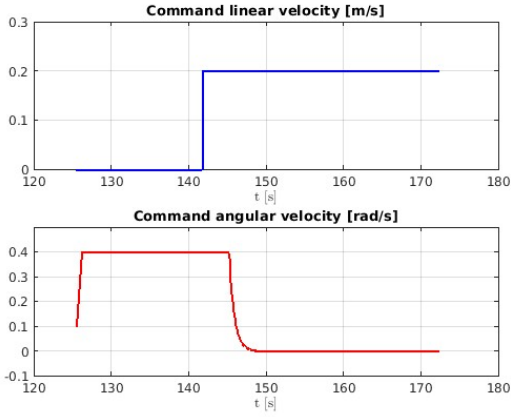


Figure 21: *command velocity*

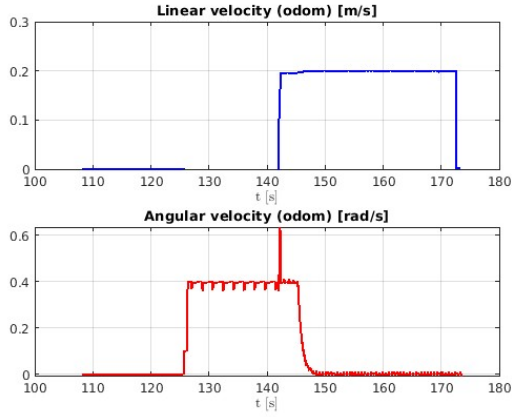


Figure 22: *measured velocity*

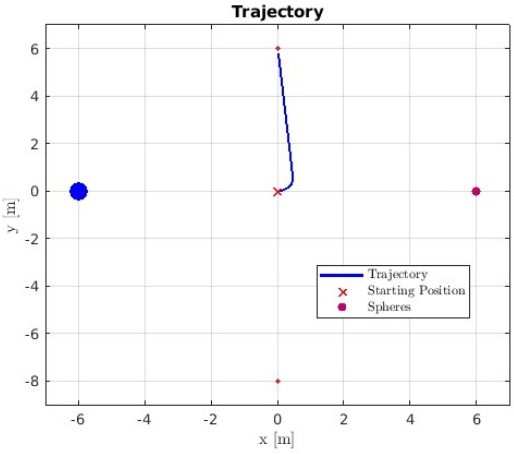


Figure 23: *trajectory*

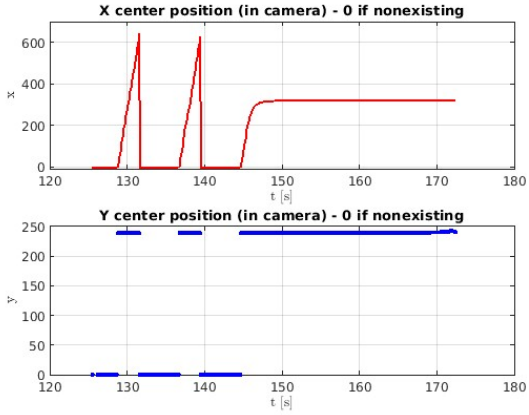


Figure 24: *blob center position*

4 Assignment IV

The fourth assignment involves implementing two optimal path-searching algorithms for motion planning in MatLab. In this context, motion planning refers to determining the sequence of actions required to guide a robot from an initial state to a goal state while avoiding obstacles and minimizing the distance traveled. Assuming ideally perfect prior knowledge of the environment, two label-correcting, grid-based algorithms are employed: Dijkstra and its improved version, A*. The search for the optimal path is conducted in a discrete graph $G = (V, E)$, represented in a highly inefficient way by an edge matrix.

Specifically, starting from a provided MatLab implementation of the Dijkstra algorithm, restricted to horizontal and vertical movements, this is used and modified to implement four variants:

1. *DVanilla*: standard Dijkstra algorithm, restricted to horizontal and vertical movements.
2. *DNav*: Dijkstra algorithm, extended to allow diagonal movements.
3. *AVanilla*: A* algorithm, restricted to horizontal and vertical movements.
4. *ANav*: A* algorithm, extended to allow diagonal movements.

The resulting algorithms are applied to four different maps, created from provided images (including predefined start and goal positions).

4.1 Map processing and edge matrix

Dijkstra is a label-correcting graph search algorithm that finds the optimal path in a given graph from the start to the goal, minimizing the total distance traveled. A* is an improved variant that analyzes fewer nodes by incorporating the *cost-to-go* heuristic into the algorithm. In this assignment, the graph is derived directly from the provided map, discretized into a grid where cells represent the nodes, and the allowed connections (based on obstacles and whether diagonal movements are permitted) define the edges. A MatLab function is implemented to process the map and build the corresponding edge matrix.

Map processing. Four 300x300 `.png` maps are rescaled into a 30x30 binary grid (*figure* 25). Green and red points represent the start and goal positions, respectively, and they must also be rescaled during processing.

Following the convention of *occupancy maps*, likely generated by SLAM algorithms, pixel colors represent probabilities: white areas correspond to free space, black areas to occupied space, and gray areas to uncertain regions, which are conservatively treated as occupied. This is coherent with the problem, as the robot cannot assume whether gray areas - inside objects - are free or not. Map processing in MatLab involves the following steps:

1. HSV conversion: The start and goal positions are identified based on their hue values (start as green with $0.2 < H < 0.5$; goal as red with $H < 0.1$ or $H > 0.9$). Once identified, they are converted to white on the map, and their coordinates in the rescaled maps are computed through the mean of their original positions.
2. Resizing: The map is resized using the `imresize` function.

3. Binarization: The map is converted to greyscale and binarized using the `imbinarize` function with a threshold of 0.99, converting to black anything that is not completely white.

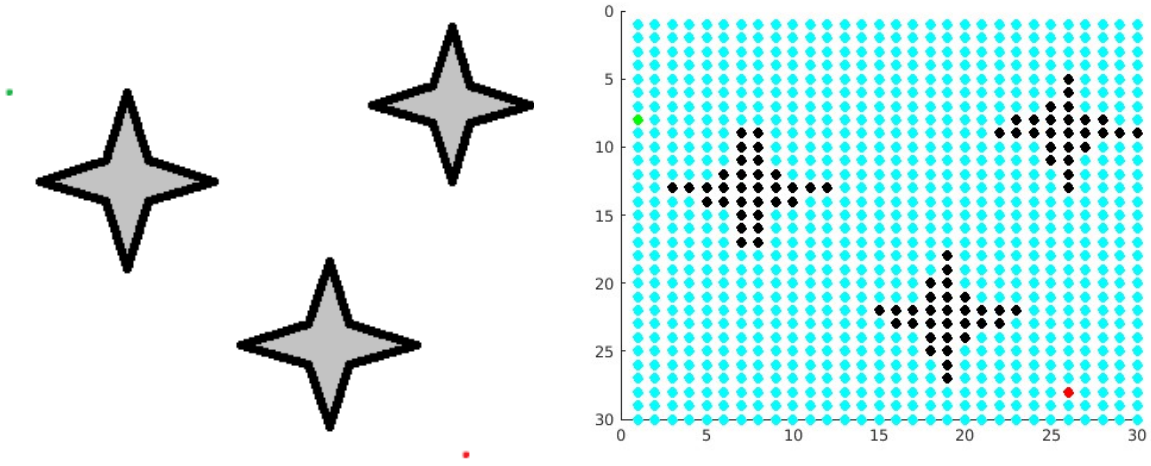


Figure 25: *map processing - image 2*

Edge matrix. To represent the edges connecting nodes in the graph, a possible approach is to construct an edge matrix G containing the connection information. Given a 30x30 map, the total number of nodes is $30^2 = 900$, resulting in a 900×900 matrix. Each element $G(i, j)$ answers the question: "Is node i connected to node j?". The values of $G(i, j)$ are defined as follows:

$$G(i,j) = \begin{cases} 0 & \text{if } i = j, \\ 1 & \text{if nodes } i \text{ and } j \text{ are free and connected,} \\ -1 & \text{otherwise (nodes not connected or either } i \text{ or } j \text{ is occupied).} \end{cases}$$

Since connectivity is transitive in this case, the matrix G is symmetric. In the MatLab implementation, G is defined based on a logical variable *diag*, which is set to `true` if diagonal movements are allowed and `false` otherwise. The definition follows accordingly:

```
G=-1*ones(size(BW,1)*size(BW,2)); % Initialize edge matrix

for ii=1:size(BW,1)
    for jj=1:size(BW,2)
        if BW(ii,jj)==1
            % Self connection
            G((ii-1)*size(BW,1)+jj,(ii-1)*size(BW,1)+jj)=0;

            % 4 directions
            if ii+1>0&&ii+1<=size(BW,1)&&BW(ii+1,jj)==1 % down
                G((ii-1)*size(BW,1)+jj,(ii)*size(BW,1)+jj)=1;
            end
            if ii-1>0&&ii-1<=size(BW,1)&&BW(ii-1,jj)==1 % up
```

```
                G((ii-1)*size(BW,1)+jj,(ii-2)*size(BW,1)+jj)=1;
            end
            if jj+1>0&&jj+1<=size(BW,2)&&BW(ii,jj+1)==1 % right
                G((ii-1)*size(BW,1)+jj,(ii-1)*size(BW,1)+jj+1)=1;
            end
            if jj-1>0&&jj-1<=size(BW,2)&&BW(ii,jj-1)==1 % left
                G((ii-1)*size(BW,1)+jj,(ii-1)*size(BW,1)+jj-1)=1;
            end

            % Diagonals
            if ii+1<=size(BW,1)&&jj+1<=size(BW,2)&&BW(ii+1,jj+1)==1&&diag==true
                G((ii-1)*size(BW,2)+jj,(ii)*size(BW,2)+jj+1)=1;
            end
            if ii-1>0&&jj+1<=size(BW,2)&&BW(ii-1,jj+1)==1&&diag==true
                G((ii-1)*size(BW,2)+jj,(ii-2)*size(BW,2)+jj+1)=1;
            end
            if ii+1<=size(BW,1)&&jj-1>0&&BW(ii+1,jj-1)==1&&diag==true
                G((ii-1)*size(BW,2)+jj,(ii)*size(BW,2)+jj-1)=1;
            end
            if ii-1>0&&jj-1>0&&BW(ii-1,jj-1)==1&&diag==true
                G((ii-1)*size(BW,2)+jj,(ii-2)*size(BW,2)+jj-1)=1;
            end
        end
    end
end
```

Matrix G is initialized as if all nodes were occupied, with all elements set to -1. The algorithm iterates over the binary occupancy grid BW (where 1 represents free space) and updates G as follows:

- **Self-connection:** for each free node (ii, jj), $G(k, k) = 0$, where $k = (ii-1)\cdot\text{size}(BW, 1)+jj$ ("fixing" the row ii while sliding through it with jj).
- **4-directional connectivity:** if a free node has free neighbors in the cardinal directions (down, up, right, left, with respect to *figure* 25), the corresponding entries in G are set to 1.
- **Diagonal connectivity:** if *diag* is `true`, diagonal neighbors (bottom-right, top-right, bottom-left, top-left) are also checked, and the corresponding entries in G are updated to 1.

It is worth noting that the use of a G matrix is highly inefficient in terms of both time and memory, making it impractical for large maps unless significant computational resources are available. Consequently, this approach was revised in Project II.

4.2 Algorithms implementation

Dijkstra and A* algorithms share a similar conceptual framework, differing primarily in how nodes are selected from the frontier queue. In this implementation, node connections are represented by the matrix G, which also encodes information about diagonal movements. Once it is determined

that two nodes are connected, the algorithm only needs to check if the connection is diagonal. Based on this, the cost of reaching a node from its parent is set to 1 for non-diagonal connections and $\sqrt{2}$ for diagonal ones. Alternatively, the cost could be directly encoded in the matrix G, yielding the same outcome.

Algorithm 2 Dijkstra (and A*)

Input: $BWmap$, G, $start$, $goal$
Output: $path$, $length$, $nnodes$
Initialize:

$$\begin{array}{lll} C(q) & \leftarrow \infty & \forall q \neq \text{start}, \\ \text{parent}(q) & \leftarrow \infty & \forall q, \\ \text{status}(q) & \leftarrow -1 & \forall q, \\ \text{act\_node} & \leftarrow start & \end{array}$$

$status(act\_node) \leftarrow 0$
Initialize con_nodes (connected to act_node): $status(con\_nodes) \leftarrow 1$
while any$(status) = 1$ **and** $act\_node \neq goal$ **do**
  for $q \in con\_nodes$ **do**
    if q connected diagonally **then**
      if $C(q) > C(act\_node) + \sqrt{2}$ **then**
        $C(q) \leftarrow C(act\_node) + \sqrt{2}$
        $parent(q) \leftarrow act\_node$
      end if
    else if q not connected diagonally **then**
      if $C(q) > C(act\_node) + 1$ **then**
        $C(q) \leftarrow C(act\_node) + 1$
        $parent(q) \leftarrow act\_node$
      end if
    end if
  end for
  $frontier\_queue \leftarrow \{q \mid \text{status}(q) = 1\}$
  $act\_node \leftarrow NODE\_EXTRACTION(frontier\_queue)$
  Find con_nodes now
  $status(act\_node) \leftarrow 0$
  $status(con\_nodes) \leftarrow 1$ **if** $status(con\_nodes) \neq 0$
end while
$path \leftarrow BUILD\_PATH(parent)$
$length \leftarrow C(goal)$
$nnodes \leftarrow \sum_{q|\text{status}(q)=0} 1$
return $path$, $nnodes$, $length$

Let us refer to *algorithm* 2. At the initialization stage, three vectors are created, each with a length equal to the number of nodes in the map:

- $C(q)$: cost-of-arrival from the start (such that $C(start) \leftarrow 0$, while every other cost is initially set to infinity). It represents the minimum distance from the start to each point.

- $parent(q)$: for each node q, it stores the parent node of q, i.e., the node from which q is reached with the minimum cost.
- $status(q)$: can take three values:

$$status(q) = \begin{cases} 0 & \text{if } q \text{ has been visited, i.e. all its children have been checked,} \\ 1 & \text{if } q \text{ is to be visited } (frontier\_queue), \\ -1 & \text{otherwise (not visited).} \end{cases}$$

The start node is initialized as the first active node, while its children (con_nodes) are added to the $frontier\_queue$. Iteratively, all the children of the active node are checked: if their cost, passing through act_nodes, decreases, it is updated along with their parent node. Once all children have been considered, act_node is removed from the $frontier\_queue$ (i.e. its status is set to 0).

A critical point in the algorithm is the selection of the next node from the frontier queue: $NODE\_EXTRACTION(frontier\_queue)$, which differs between the two algorithms. This step determines the next act_node, whose con_nodes will be processed in the following iteration.

The iterations stop when there are no more nodes in the frontier queue (indicating no solution) or when the goal node is reached as the active node.

At the end, if a solution exists, it can be reconstructed by tracing backward using the $parent$ vector ($BUILD\_PATH(parent)$), computing the length as the goal cost, and finally evaluating the total number of analyzed nodes $nnodes$ as all points whose status is visited.

Dijkstra. Dijkstra orders nodes in the frontier queue by optimal cost-to-arrival. The corresponding $NODE\_EXTRACTION$ pseudocode is presented in *algorithm* 3.

Algorithm 3 NODE_EXTRACTION - Dijkstra

1: **Input:** $frontier\_queue$ (and $C(q)$)
2: **Output:** act_node
3: act_node $\leftarrow \arg\min_{q \in \text{frontier\_queue}} C(q)$
4: **return** act_node

A*. A* orders nodes in the frontier queue by summing the cost-to-arrival to an underestimate of the cost-to-go, $h(q, goal)$. This acts as a metric with respect to the goal. In this implementation, the Euclidean distance has been chosen for $h(q, goal)$; however, nothing prevents the use of alternative approaches, such as the Manhattan distance. Trials with this alternative metric have been conducted, showing no performance improvement, and it has therefore been discarded.

Using a cost-to-go for node selection means that the algorithm prioritizes, with respect to the goal, the node most likely to lead to the optimal path. As a result, A*, despite converging to the optimal solution (or to an equivalently optimal one, i.e., with the same length) just like Dijkstra, analyzes fewer nodes, satisfying the inequality:

$$nnodes(\text{Dijkstra}) \geq nnodes(\text{A*}).$$

Moreover, other costs remain unchanged, as both algorithms guarantee optimality. The difference lies in the node selection process, where the sum $C + h$ is (only) used for choosing the next act_node. The corresponding $NODE\_EXTRACTION$ pseudocode is presented in *Algorithm* 4.

Algorithm 4 NODE_EXTRACTION - A*

1: **Input:** $frontier\_queue$ (and $C(q)$, and heuristic h)
2: **Output:** act_node
3: act_node $\leftarrow \arg\min_{q \in \text{frontier\_queue}} \left[C(q) + h(q, goal)\right]$
4: **return** act_node

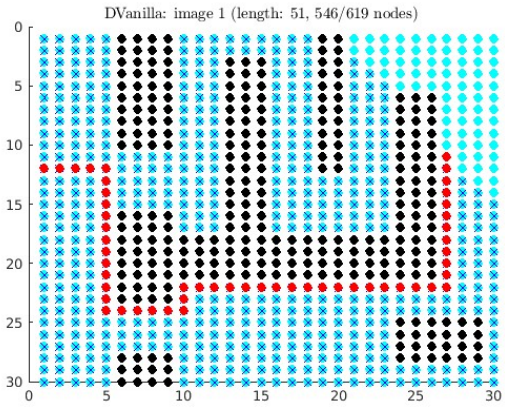
4.3 Results

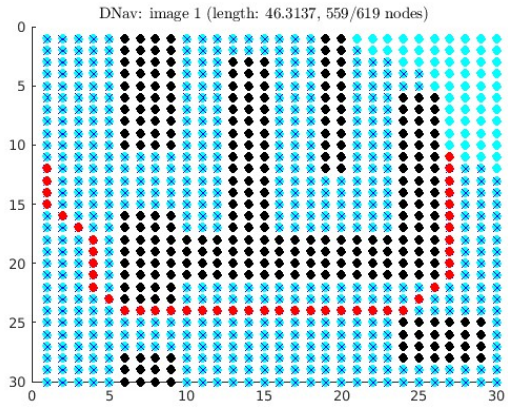
All four algorithms (*DVanilla*, *DNav*, *AVanilla*, *ANav*) are applied to the four maps derived from the provided images. The grids are 30x30: in *figures* 26, 27, 28, and 29, free nodes are shown in cyan, obstacles in black, visited nodes in blue, and the optimal path in red. Performance metrics include the optimal path length and the percentage of evaluated nodes.

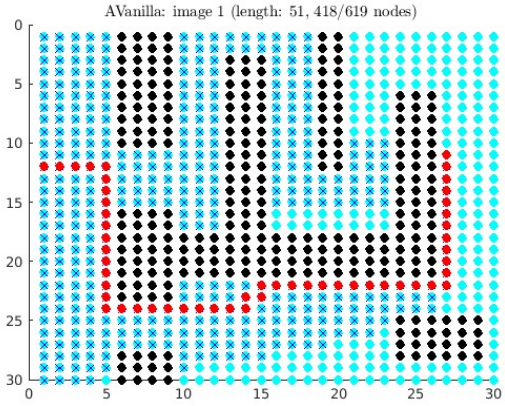
The difference between allowing or not diagonal movements results in sub-optimal solutions in terms of path length, just because the non-diagonal algorithm has fewer degrees of freedom. However, the percentage of analyzed nodes is not consistently higher or lower between the two cases, as this depends on the specific map.

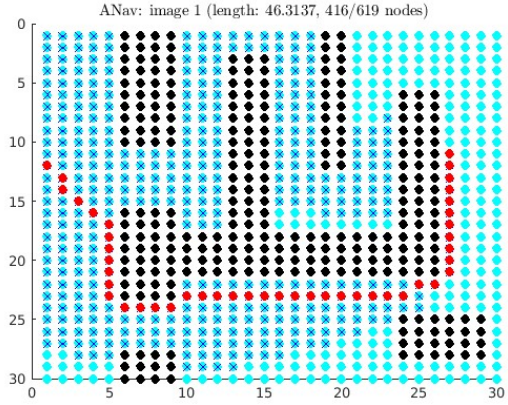
According to theory, the solution remains identical in terms of path length when using Dijkstra or A*, while the number of traversed nodes may differ. This discrepancy arises from the different methods these algorithms use to select the active node.

Thanks to the use of a (non-constant) heuristic, A* consistently converges to the solution with less effort in terms of analyzed nodes. This efficiency is particularly evident in specific, more "open" maps, such as the second one, where the algorithm's "smart" behavior is highlighted. In the third case, Dijkstra's algorithm struggles in terms of efficiency, analyzing 100.00% of the nodes, while A* reduces the computational effort by leveraging the heuristic h (always at the expense of computing it).



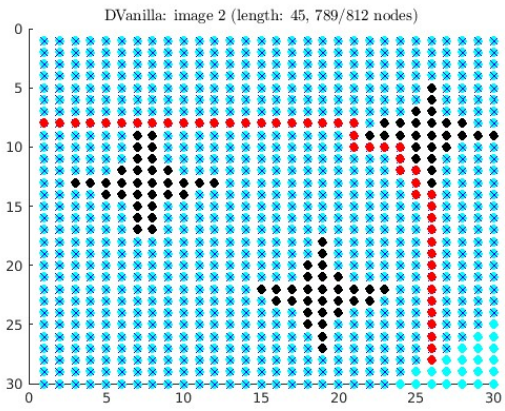


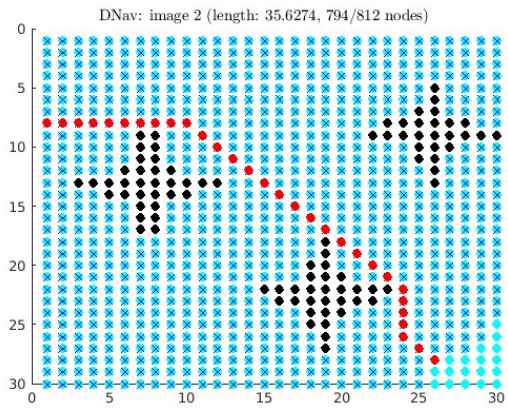


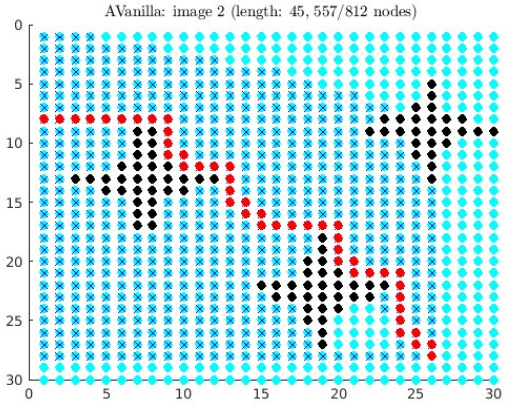


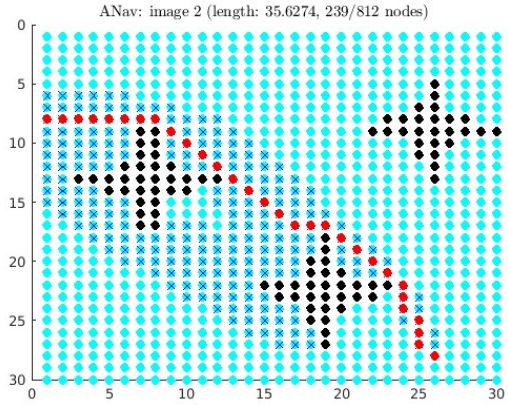
Algorithm	**Path length**	**Evaluated nodes (%)**
DVanilla	51	88.69
DNav	46.3137	90.31
AVanilla	51	67.53
ANav	46.3137	67.21

Figure 26: *map 1*



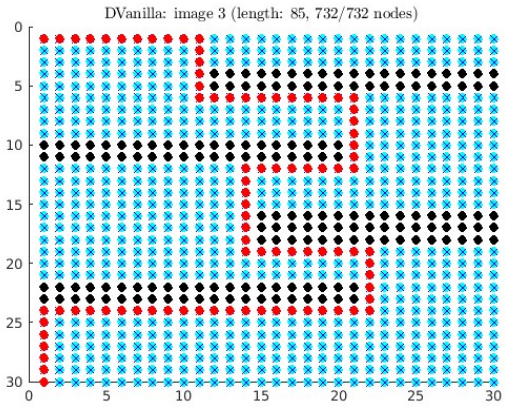


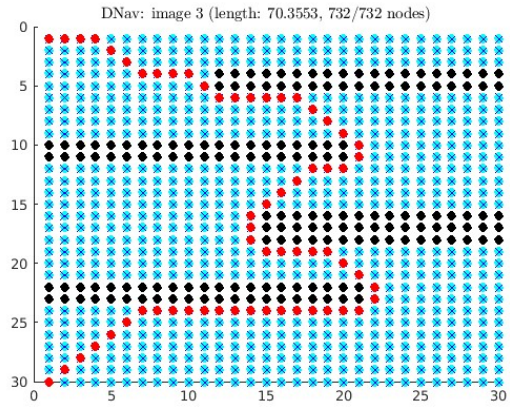


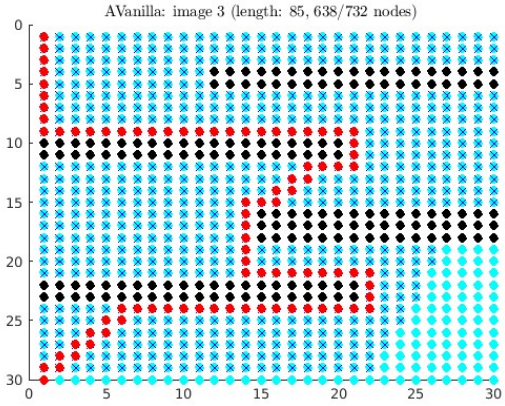


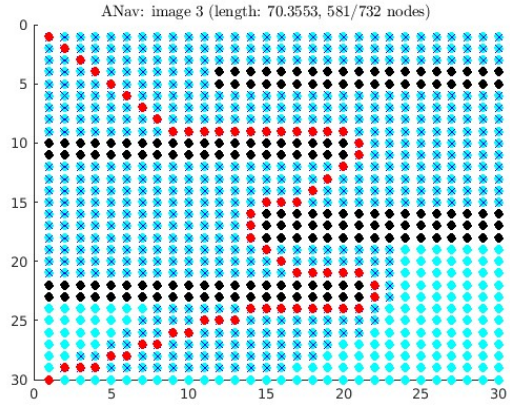
Algorithm	**Path length**	**Evaluated nodes (%)**
DVanilla	45	97.17
DNav	35.6274	97.78
AVanilla	45	68.60
ANav	35.6274	29.43

Figure 27: *map 2*



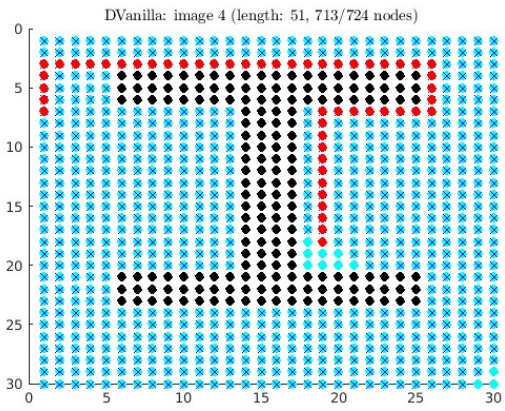


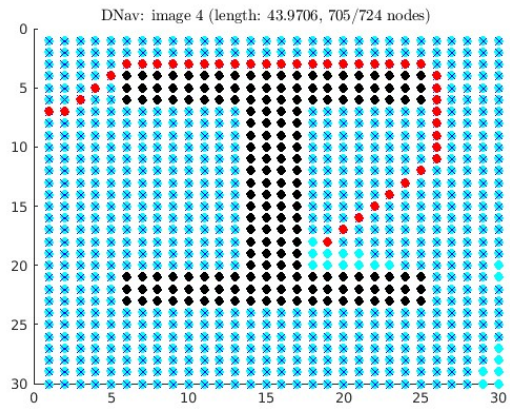


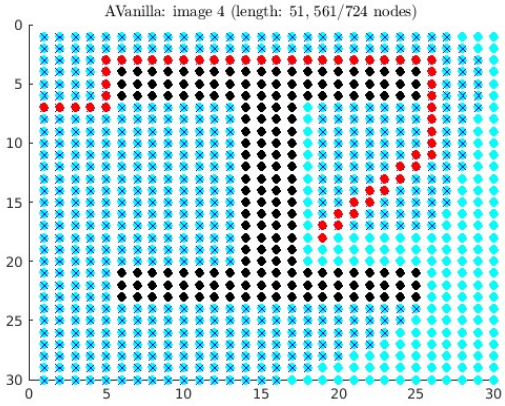


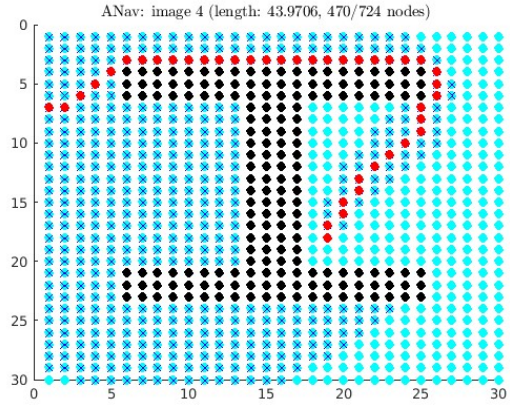
Algorithm	**Path length**	**Evaluated nodes (%)**
DVanilla	85	100.00
DNav	70.3553	100.00
AVanilla	85	87.16
ANav	70.3553	79.37

Figure 28: *map 3*









Algorithm	**Path length**	**Evaluated nodes (%)**
DVanilla	51	98.48
DNav	43.9706	97.38
AVanilla	51	77.49
ANav	43.9706	64.92

Figure 29: *map 4*

5 Assignment V

The fifth assignment involves starting with the finite state model of a car (*vehicle FSM*), analyzing its functionality, and integrating it with the FSM model of a *parking gate control* system. This integration must be performed without altering the original vehicle FSM. The combined models are then executed together, and the results are analyzed and discussed.

5.1 Vehicle FSM

The provided FSM of the vehicle is shown in *figure* 30. Inputs are:

- $x \sim N(-15, 5)$: new car's starting position (reference with respect to the gate).
- *gate_up*: output of the parking gate control FSM in *subsection* 5.2, a logical variable defining if the gate is up (90°) or not.
- *start*: logical variable to define if the simulation has started; in this case, $start = 1$ always.

Outputs (continuously analyzed, like the inputs) are:

- *pos*: the current car's position, starting from x (reset for a new car 10 seconds after the old one has passed).
- *car_waiting*: true if the car is waiting for the gate to raise.
- *car_passed*: true if the car has already passed through the gate; false otherwise.

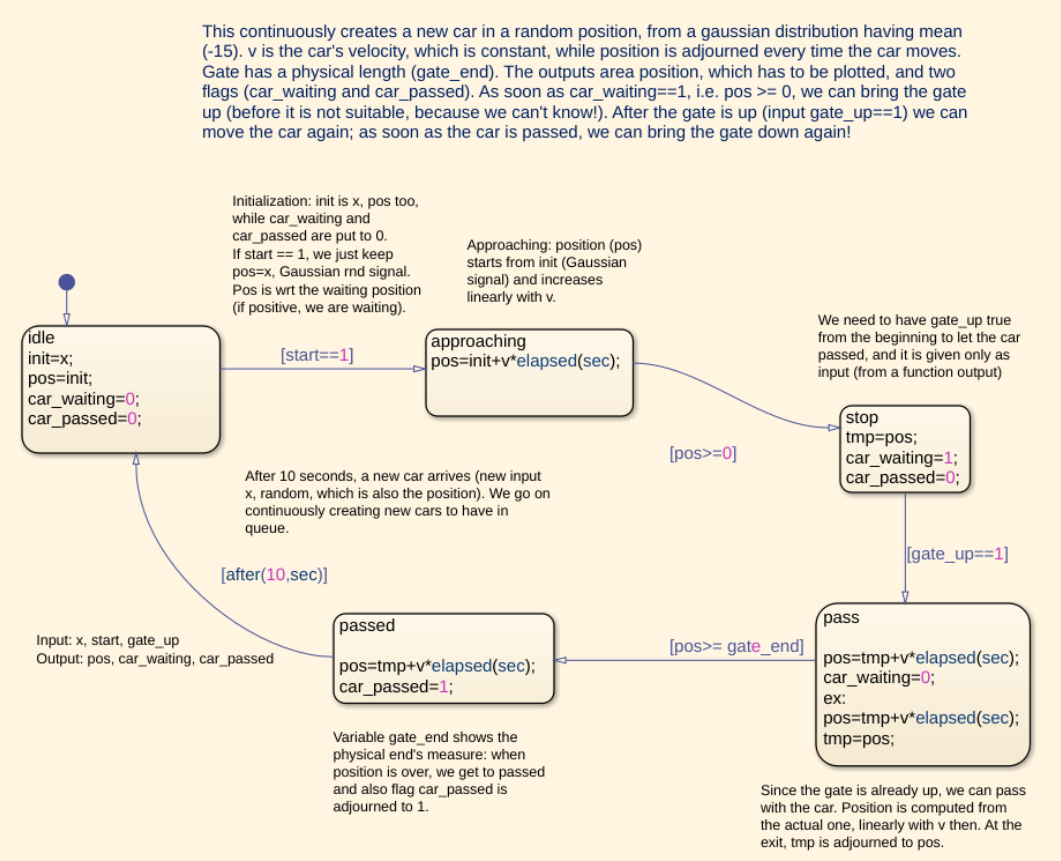


Figure 30: *vehicle FSM*

At the beginning (*idle*), the car is at its initial position x, and it is neither waiting nor has it passed. This allows the vehicle to move at a constant speed v (*approaching*) until it reaches the gate. At this point, it is forced to wait for the gate to raise (*stop*). Once the gate is raised, the car can move again. It continues moving for *gate_end* meters (*pass*), always at a constant velocity. Once it surpasses *gate_end*, the vehicle is considered to have passed (*passed*). After 10 seconds (arbitrarily), a new car is assumed to arrive at position x, generated from a normal distribution. The cycle then starts again, with the new car neither waiting nor having passed yet.

5.2 Parking gate control FSM

The parking gate control FSM is shown in *figure* 31. Its inputs are the outputs from the vehicle FSM (see *subsection* 5.1):

- *car_waiting*
- *car_passed*

The outputs are:

- *gate_up*: indicates whether the gate is up or not, and is provided as input to the vehicle FSM.
- *motion*: represents the angle, in degrees (from 0° to 90°), of the gate during raising and lowering, respectively computed by the functions `custom_raise` and `custom_lower`.
- *speed* and *acceleration*: computed alongside *motion*, they are derived from it. Their explicit computation is preferred to prevent misleading computational errors in derivatives, such as spikes.

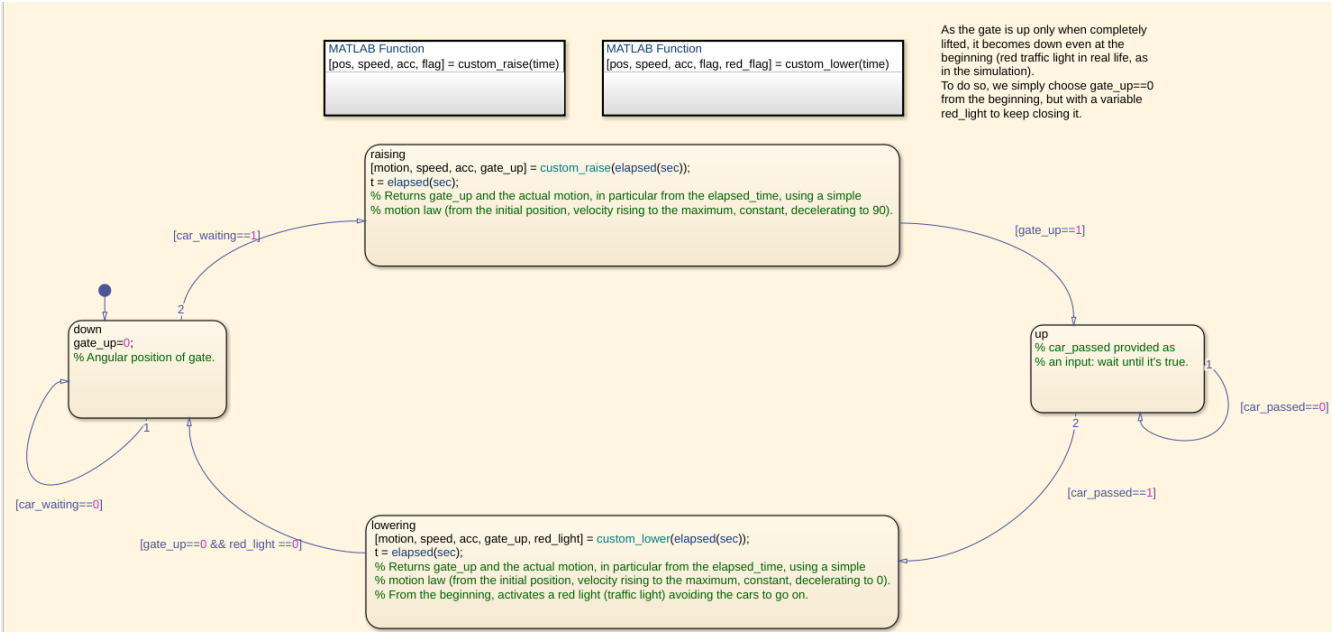


Figure 31: *parking gate FSM*

At the beginning (*down*), the gate is not up and remains in this position as long as there is no car waiting. When a car approaches the gate - and surpasses *pos*, the gate starts opening (*raising*: `custom_raise`).

As soon as $\theta = 90°$, the gate is fully up (*up*), and this information is provided to the vehicle FSM through *gate_up*. It remains in this state until the car has passed (i.e. *pos* = *gate_end*). Once the vehicle is out of view, the gate starts closing (*lowering*: `custom_lower`).

It is important to note that an additional flag variable must be introduced: while *gate_up* $\leftarrow 0$ as soon as the gate starts closing, the next state (*down*) can only be reached when the gate is fully closed. Until then, a *red_light* variable remains true, mimicking the behavior of traffic lights in real gates.

MatLab functions `gate_up` and `gate_down` are defined (5.2.1, 5.2.2), along with the corresponding angle, speed, and acceleration profiles (*figures* 32 and 33). Both motion laws are implemented in a feedforward fashion, ensuring the fastest possible behavior with a trapezoidal speed profile, under the following constraints:

1. Maximum angular speed: $|\omega_{\max}| = 4°/\text{s}$
2. Maximum angular acceleration: $|\alpha_{\max}| = 1°/\text{s}^2$

5.2.1 Function *custom_raise*

```
function [pos, speed, acc, flag] = custom_raise(time)
%#codegen

    if time < 4
        % First part, when we accelerate.
        pos = 1/2 * max_acc_dec * time^2;
        speed = max_acc_dec * time;
        acc = max_acc_dec;
    elseif time < 22.5
        % Second part, constant velocity (starting from 8°, 4s).
        pos = 8 + max_speed * (time - 4);
        speed = max_speed;
        acc = 0;
    else
        % Third part, starting from 82° and decelerating.
        pos = 82 + max_speed * (time - 22.5) - 1/2 * max_acc_dec * (time - 22.5)^2;
        speed = max_speed - max_acc_dec * (time - 22.5);
        acc = -max_acc_dec;
    end

    if time > 26.5
        flag = 1;
        pos = 90;
        speed = 0;
        acc = 0;
```

```
    else
        flag = 0;
    end
end
```

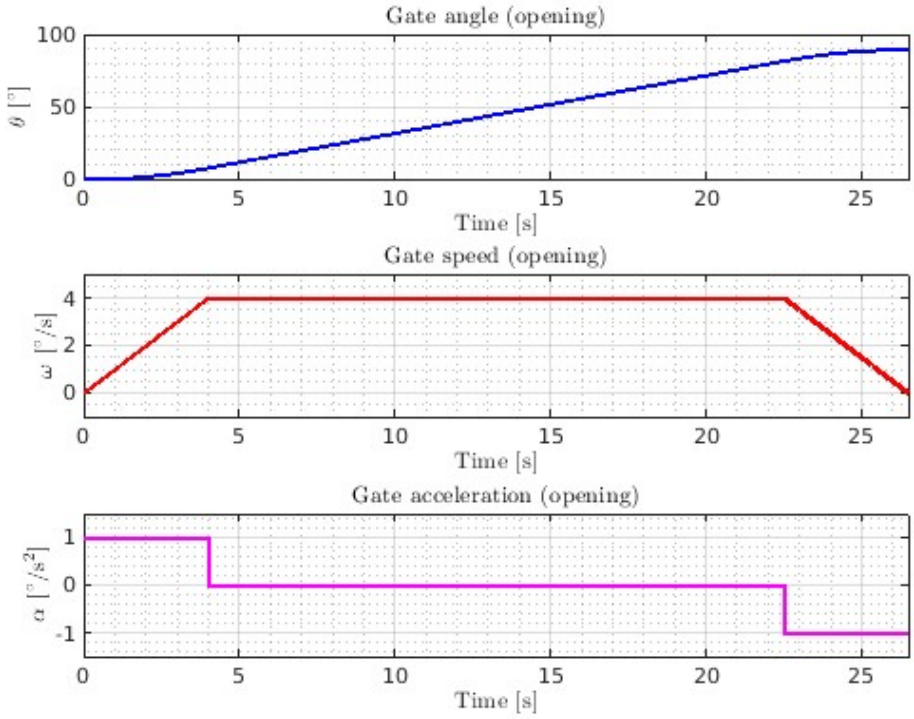


Figure 32: *opening profile*

5.2.2 Function *custom_lower*

```
function [pos, speed, acc, flag, red_flag] = custom_lower(time)
%#codegen

    pos = 90;
    flag = 0;
    red_flag = 1;

    if time < 4
        % First part, when we accelerate.
        pos = 90 - 1/2 * max_acc_dec * time^2;
        speed = -max_acc_dec * time;
        acc = -max_acc_dec;
    elseif time < 22.5
        % Second part, constant velocity (starting from 82°, 4s).
```

```
        pos = 90 - 8 - max_speed * (time - 4);
        speed = -max_speed;
        acc = 0;
    else
        % Third part, starting from 82° and decelerating.
        pos = 8 - max_speed * (time - 22.5) + 1/2 * max_acc_dec * (time - 22.5)^2;
        speed = -max_speed + max_acc_dec * (time - 22.5);
        acc = max_acc_dec;
    end

    if time > 26.5
        red_flag = 0;
        pos = 0;
        speed = 0;
        acc = 0;
    end
end
```

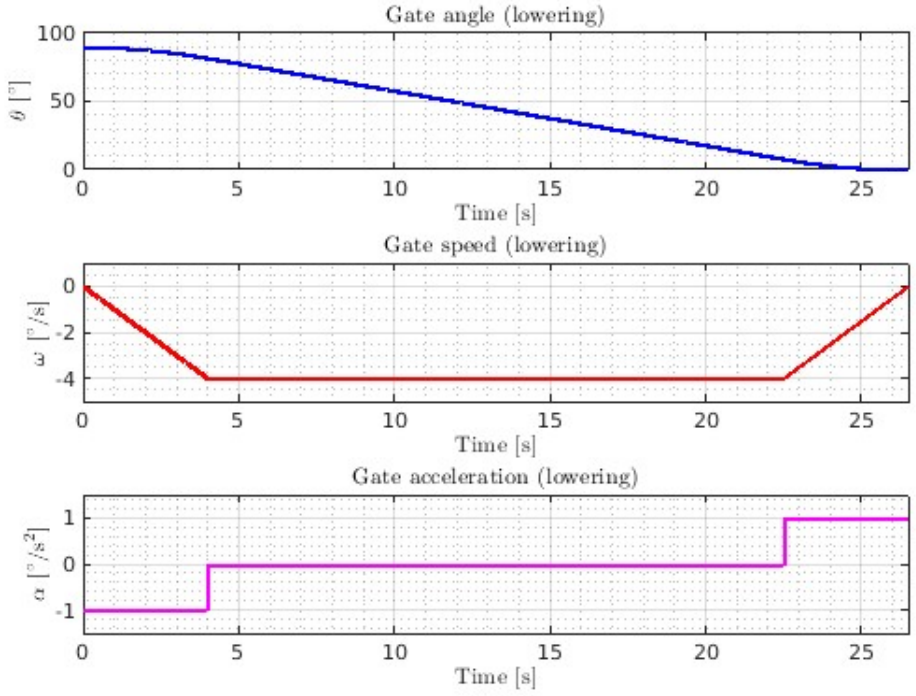


Figure 33: *lowering profile*

5.3 Results

In *figure* 34, the positions of both cars and the gate motion are compared over a 200-second timespan. Cyan dashed lines indicate when *car_waiting* ← 1 (i.e. when the car has arrived at the gate), while orange dashed lines represent when *car_passed* ← 1 (i.e. after *gate_end*).

The car position, as expected, linearly spans values from a random starting point (centered at -15) up to zero, at which point a cyan line appears. At the corresponding time instant, the gate starts to raise for the first car. However, for subsequent cars, the behavior is different: when a car is waiting, the gate has not fully lowered yet. It finishes lowering before raising again.

It could be pointed out that this behavior is not the most efficient solution, as the presence of the car could directly trigger the gate raising. This implementation represents only one possible approach to the problem.

Coherently, speed increases when $car\_waiting \leftarrow 1$, while acceleration exhibits a step behavior, reaching $1°/s^2$.

Once the gate is fully raised, the car passes, and the gate remains up (and stationary) until the car reaches *gate_end*. Then, the gate lowers — following the `custom_lower` profile — and the cycle begins anew.

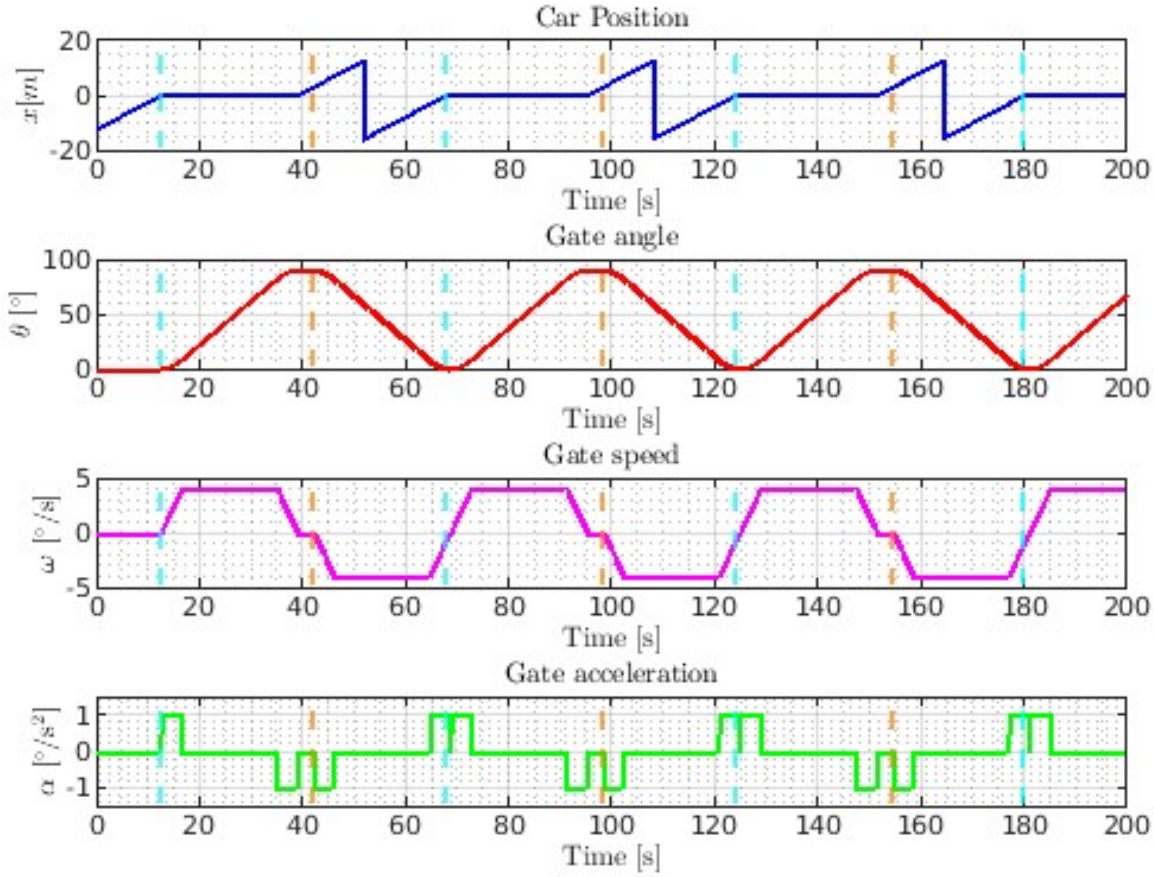


Figure 34: *vehicles positions vs gate motion*