Autonomous Vehicles
Assignment I: Use of ROS bags

Tommaso Bocchietti 10740309

A.Y. 2024/25

POLITECNICO
MILANO 1863

Contents

List of Figures

1 Introduction

The aim of this work is to gain insight into the use of `rosbags` for the analysis of data collected by autonomous vehicles.
The following sections provide a detailed description of the requests associated with this assignment, the approach taken to fulfill them, and the discussion of the results obtained.

Tools As for the tools used, `ROS1` (Robot Operating System) is employed as the main framework for data collection, while their analysis is performed using `MATLAB`. Notice that with the current setup used by the author, `MATLAB 2024a` is running in Windows 10, while `ROS1` is running in the `WSL2` (Windows Subsystem for Linux) environment, specifically with the `Ubuntu 20.04` distribution.

2 Assignment Part A

In this section, we provide a brief overview of the requests associated with the first part of the assignment, along with a description of the approach taken to fulfill them. Discussion of the results obtained is also included.

2.1 Request

Starting from the data collected in the provided `rosbag` file, the goal of this first part of assignment is to evalute at each time instant, the following quantities:

- **Minimum distance** of the vehicle from the obstacles in the environment;
- **Estimate of `/cmd_vel` command** sent to the vehicle during the simulation.

We also recal that the `rosbag` file was obtained by running a `Turtlebot3` robot of the model `burger` in a simulated environment, specifically the `turtlebot3_world` provided by the `turtlebot3_gazebo` package.

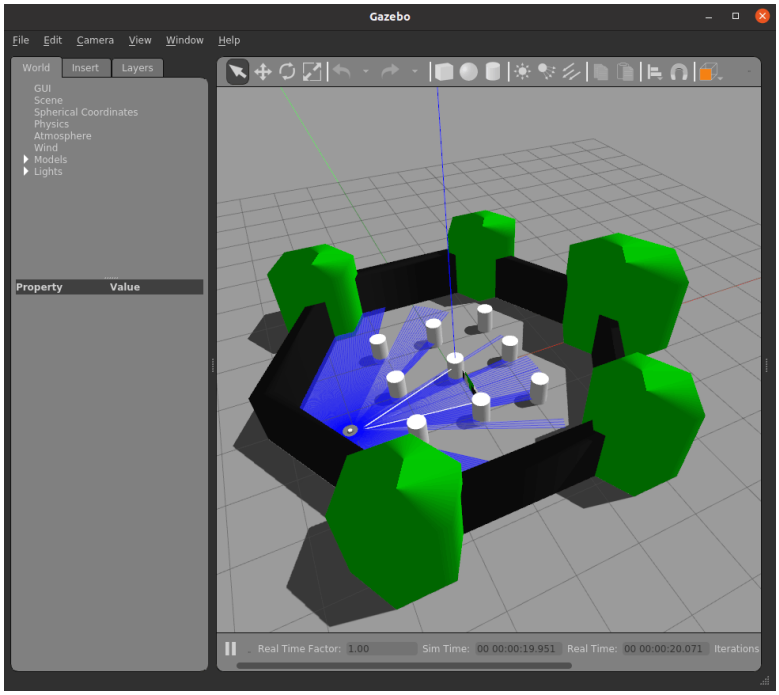

Figure 1: Gazebo world used for the simulation. Credit to `https://emanual.robotis.com/`

2.2 Analysis

At first, the `rosbag` file is loaded into `MATLAB` using the `rosbag` function, which allows to read and manipulate `rosbag` files in a convenient way. A preliminary analysis of the data contained in the `rosbag` file is performed showing the following topics:

	NumMessages	MessageType
/clock	110,710	rosgraph_msgs/Clock
/imu	110,240	sensor_msgs/Imu
/odom	3,316	nav_msgs/Odometry
/scan	552	sensor_msgs/LaserScan
/tf	3,316	tf2_msgs/TFMessage

Table 1: Topics contained in the `rosbag` file.

2.2.1 Minimum distance from obstacles

Given the presence of the `/scan` topic, which contains the laser scan data, we can use it to compute the minimum distance from obstacles in the environment. The `scan` message contains a field called `ranges`, which is an array of distances measured by the laser scanner at different angles. The minimum distance can be computed by taking the minimum value of this array, which represents the closest obstacle detected by the laser scanner at that time instant. The following code snippet shows how to extract the `ranges` field from the `/scan` topic and compute the minimum distance at each time instant:

```
scan = select(bag, 'Topic', '/scan');
scan_msgs = readMessages(scan, 'DataFormat', 'struct');
scan_time = scan.MessageList.Time - scan.StartTime;
scan_ranges = cell2mat(cellfun(@(msg) msg.Ranges(:)', scan_msgs, 'UniformOutput', false)
    );
min(scan_ranges, [], 2)
```

Listing 1: Extracting the `ranges` field from the `/scan` topic and computing the minimum distance at each time step.

One can also plot the minimum distance over time, as shown in Figure 2.

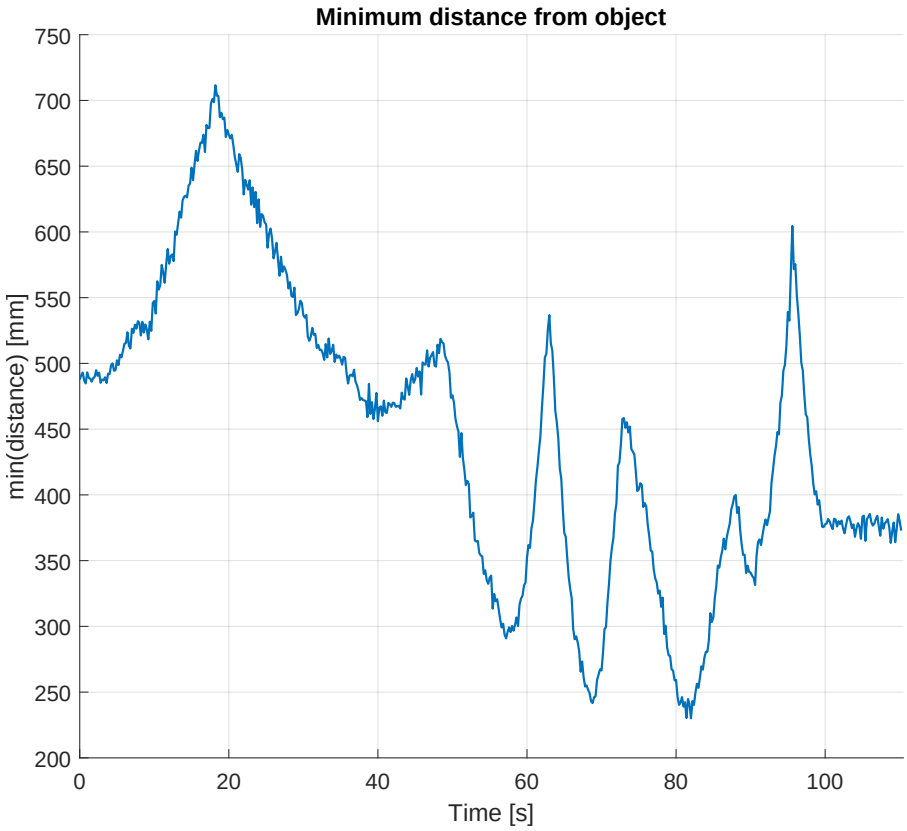

Figure 2: Minimum distance from obstacles over time.

It's important to notice that, being the data coming from a simulated environment, we can expect the accuracy of the laser scanner to be almost perfect (unless the simulation environment add noise on purpose), while on a real vehicle we could face issues like sensor noise or hit of minimum and/or maximum range of the laser scanner.

This could lead to a less accurate estimation of the minimum distance from obstacles, which could be a problem for the robot navigation and obstacle avoidance.

2.2.2 Estimate of `/cmd_vel` command

The `/cmd_vel` topic usually contains the velocity commands sent to the robot, which are typically represented as linear and angular velocities.
As an exercise, given that in the provided `rosbag` file the `/cmd_vel` topic has been removed, we can estimate the linear and angular velocities of the robot using the `/odom` topic, which contains the odometry data. The `odom` message contains the position and orientation of the robot in the world frame, as measured by the odometry system.
Again, with a similar faschion as done for the `/scan` topic, we can extract the information from the `/odom` topic, save them in a matrix form and plot them.

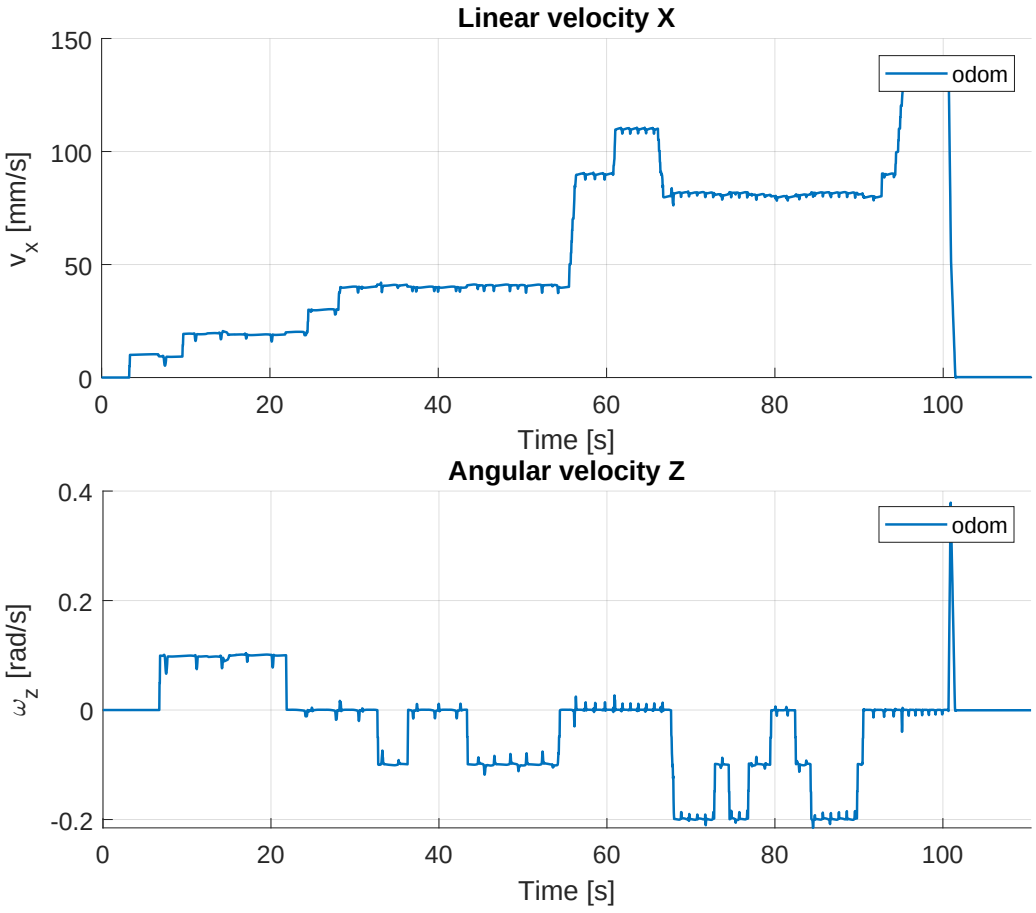

Figure 3: Velocities of the robot over time, purerly based on the odometry data.

Notice that one could have also possibly used the `/imu` topic to estimate both position and velocities of the robot (via single and double integration of the acceleration data). However, due to the presence of noise in the IMU data and cumulative errors due to integration, this approach is definitely not recommended as it easily results in drifted output estimation.
On the other hand, a sensor fusion approach based for example on the Kalman filter could have been used to combine the two sources of data (IMU and odometry) ultimately increasing the overall accuracy.

3 Assignment Part B

In this section, we provide a brief overview of the requests associated with the second part of the assignment, along with a description of the approach taken to fulfill them. Discussion of the results obtained is also included in the following sections.

3.1 Request

Given that in the previous section we have estimated the control commands sent to the robot during the original simulation, in this second part we are asked to evaluate the accuracy of such commands.

3.2 Analysis

In order to evaluate the accuracy of the control commands sent to the robot during the simulation, we need to rerun a simulation by our own, using the same world and the same robot model, and then compare both the trajectory and the velocities with the ones saved in the original `rosbag` file.
We recall that our working environment is setup so that `ROS1` is running in the `WSL2` (Windows Subsystem for Linux) environment, specifically in `Ubuntu 20.04`, while `MATLAB 2024a` is running in Windows 10. Based on these considerations, we can run the simulation in at least three different ways:

- Create a publisher script in `MATLAB` that sends the commands to `ROS1` running in `WSL2`;
- Create a publisher in `Simulink` that sends the commands to `ROS1` running in `WSL2`;
- Perform the analysis directly in `WSL2` using the native commands of `rosbag` to automatically replay the estimated commands.

At first, we tried to create a publisher script in `MATLAB` that was sending the commands over the bridged network, pacing the sender based on the original timing of the `/odom` topic. However, we encountered some major issues with the timing of the commands, which were sent at almost unpredictable intervals, leading to a completely different behavior of the robot in the simulation.
In order to avoid or at least reduce the issues related to the timing of the commands, we decided to directly move to the third option, which is to perform the analysis directly in `WSL2` using the native commands of `rosbag` to automatically replay the estimated commands. At first, the `/odom` topic has been ported to a new `rosbag` file using a simple `python` script, which is able to read the original `rosbag` file and write the `/odom` topic to a new `rosbag` file while also reshaping the data to match the type of the `/cmd_vel` topic. In particular, the following relationship table has been implemented:

/odom	**/cmd_vel**
msg.Twist.Twist.Linear.X	msg.Linear.X
msg.Twist.Twist.Angular.Z	msg.Angular.Z

Table 2: Relationship between the `/odom` and `/cmd_vel` topics.

Once the content of the `/odom` topic has been saved as `/cmd_vel` topic in a new `rosbag` file, we can use the `rosbag play` command to replay the commands in the simulation.
In Figure 4 we can see the comparison of the trajectories performed by the original simulation and the one performed by the replayed commands.

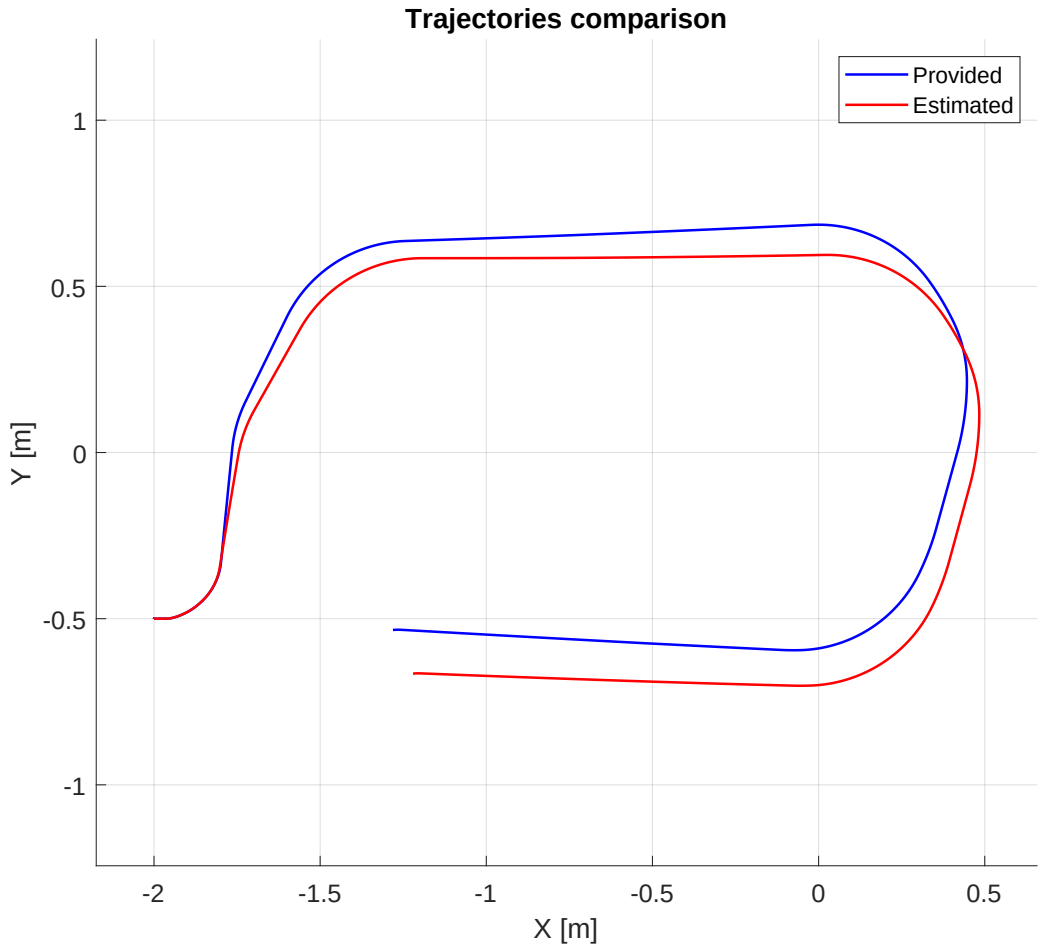

Figure 4: Comparison of the trajectories performed by the original simulation and the one performed by the replayed commands.

One can clearly see that the two trajectories are not completely overlapping.
Looking at the values of the actual velocities of the robot during the simulation, we can see a minor discrepancy between the original simulation and the one performed by the replayed commands, similarly to the behaviour observer for the trajectories. Figure 5 shows the comparison of the velocities performed by the original simulation and the one performed by the replayed commands.

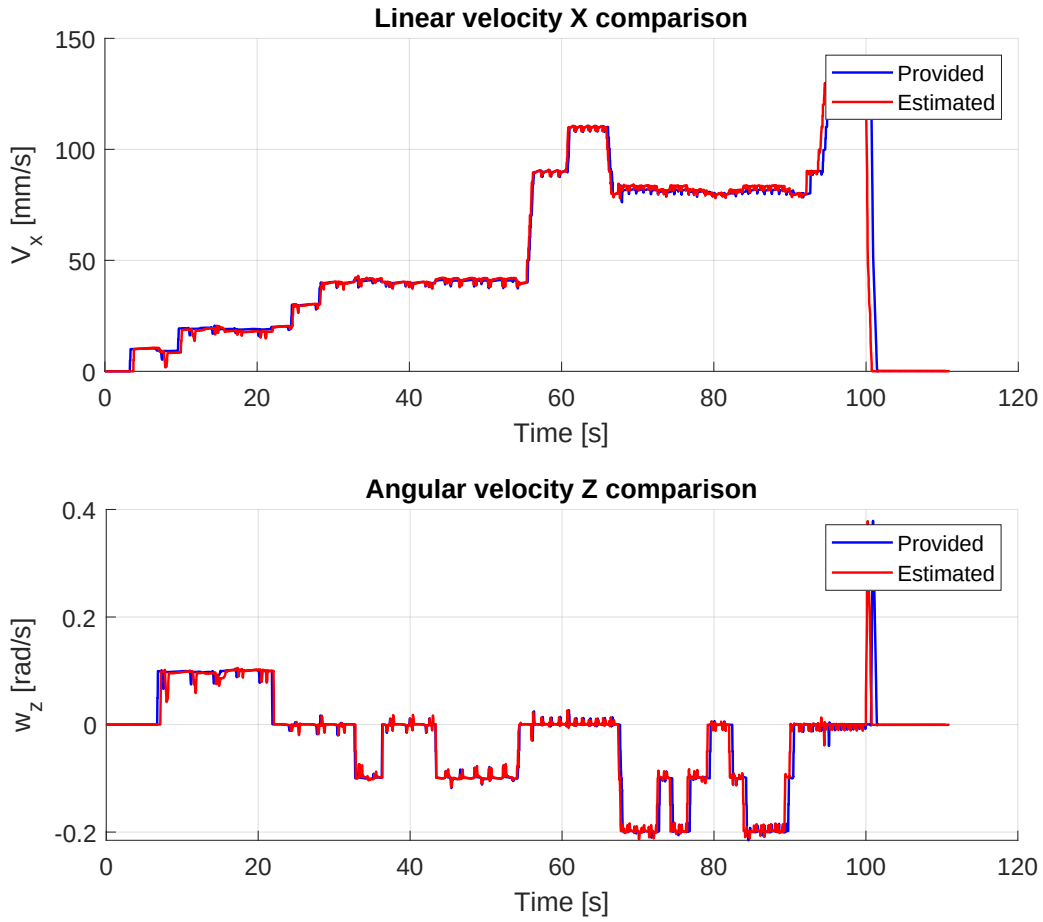

Figure 5: Comparison of the velocities performed by the original simulation and the one performed by the replayed commands.

Again, some minor issues related to timing can be observed. For some reason that the author is not able to determine, the `rosbag play` command is not able to replay the commands at the exact same pace as the original simulation, leading to a slight shift in the velocities.
What is also interesting to notice, are the juggling of the velocities along the whole simulation. These are, in fact, not desired behaviors and most probably caused by the simulated environment itself.

4 Conclusions

In this short report, we have presented the results of the first assignment of the course on Autonomous Vehicles. We have shown how to analyze data collected by a simulated robot in a Gazebo environment, using the `rosbag` command to store the data. We have also demonstrated how to estimate the control commands sent to the robot during the simulation, and how to evaluate their accuracy by comparing trajectories and odometer readings.
Results obtained show that the estimated commands are representative of the original ones, but not completely overlapping due to poor timing performance caused (probably) by the `WSL2` environment that can't fully afford the real-time performance required by the simulation.
Further investigations and testing could be done by switching to a native `Linux` environment, which could provide better performance and more accurate results.

References

[1] Stanford Artificial Intelligence Laboratory et al. Robotic operating system.

Autonomous Vehicles

Assignment II: Feedback control of a differential drive robot

Tommaso Bocchietti 10740309

A.Y. 2024/25

POLITECNICO

MILANO 1863

Contents

List of Figures

1 Introduction

The aim of this work is to gain insight into the development of a feedback control loop for an autonomous vehicle, specifically a `Turtlebot3` robot of the model `burger`. Two different control strategies have been implemented, namely a simple proportional controller and a non-linear controller based on Lyapunov stability theory. As for the simulation environment, the `Turtlebot3` robot was simulated in a Gazebo world, specifically the `turtlebot3_world` provided by the `turtlebot3_gazebo` package.
The following sections provide a detailed description of the requests associated with this assignment, the approach taken to fulfill them, and the discussion of the results obtained.

Tools As for the tools used, `ROS1` (Robot Operating System) is employed as the main framework for communication between the different components of the system. `Simulink` is used as the main agent in the loop, sending control commands to the vehicles based on the telemetry data received from the vehicle itself. `MATLAB` instead is used to perform the analysis of the data collected during the simulation, and to visualize the results. Notice that with the current setup used by the author, `MATLAB 2024a` and `Simulink` are running in Windows 10, while `ROS1` is running in the `WSL2` (Windows Subsystem for Linux) environment, specifically in the `Ubuntu 22.04` distribution.

2 System overview

In this section, we provide a brief overview of the requests associated with this assignment, along with a description of the approach taken to fulfill them. In the successive two sections instead, we describe in detail the control strategies implemented, and the analysis of the results obtained.

2.1 Request

For this assignment, we are asked to implement a feedback control strategy that is able to drive a `Turtlebot3` robot of the model `burger` through some predefined waypoints in the environment.
In particular, the waypoints are specified as a list of 3D coordinates in the world frame, where the first two coordinates represent the position of the vehicle in the plane, and the third coordinate represents the orientation of the vehicle in the world frame.
Also, along with this core request, we are asked to implement two control strategy such as:

- A simple proportional controller, subjected to a set of constraint;
- A more advanced controller (no particular constraints on its choice).

2.2 Waypoints

Before proceeding with the actual implementation of the control system, we show in Figure 1 the waypoints used for this assignment. These waypoints are provided along with the text of the assignment.

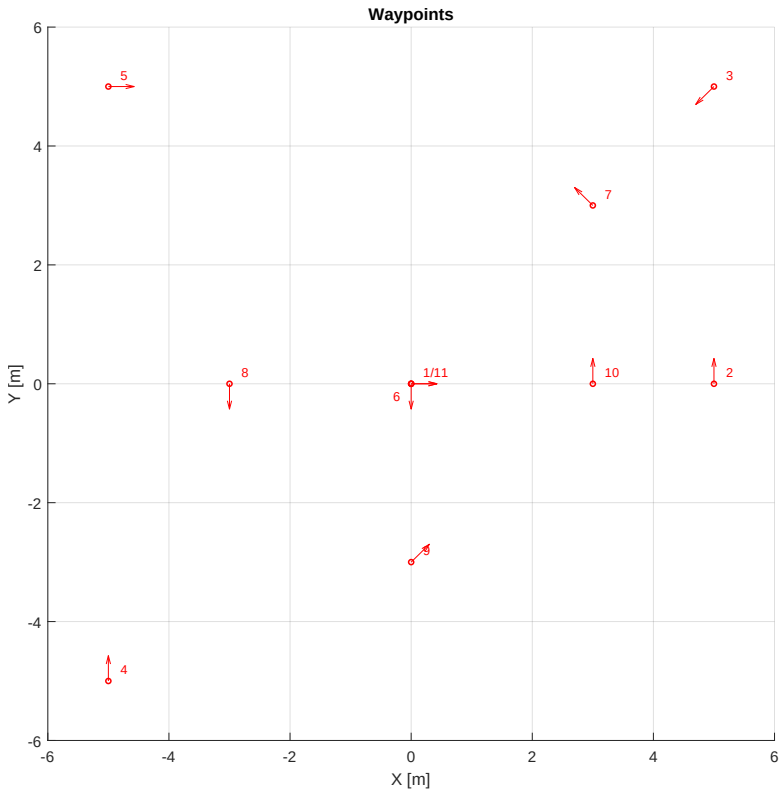

Figure 1: Waypoints used for the simulation.

In Figure 1, we can see the waypoints used for the simulation, which are represented as red dots in the world frame. The arrow associated to each waypoint represents the orientation of the vehicle at that waypoint, which is given by the third coordinate of the waypoint list.

2.3 Simulink model

As already mentioned in the previous section, the logic of the feedback control loop is implemented in `Simulink`, which is a graphical programming environment for modeling, simulating and analyzing dynamic systems. In Figure 2 we can see the Simulink model used for this assignment.

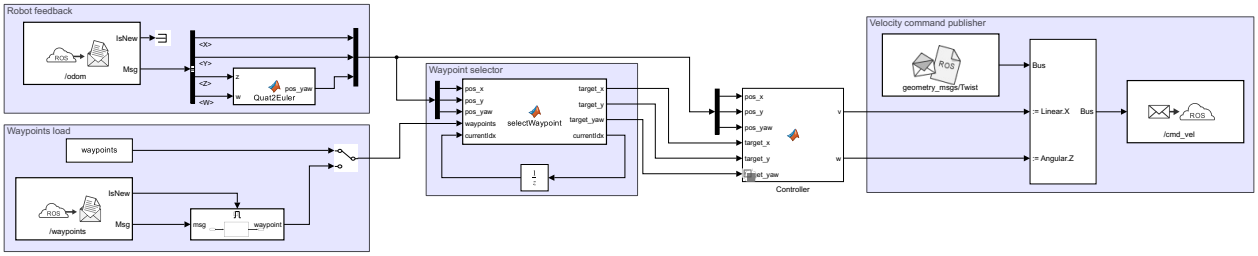

Figure 2: Simulink model used for the implementation of the feedback control loop.

One can easily recognize the different components of the system thanks to the colored areas used to group the different blocks. The main components of the system are:

- Sensors readings: these blocks are used to subscribe to the `/odom` topic and to extract the data needed for the control system;
- Waypoint loader: this block is used to load the waypoints from either a workspace variable or a custom topic `/waypoints` on which the list of waypoints is published;
- Waypoint selector: this block is used to select the current waypoint from the list of waypoints, based on the current position of the vehicle and its history;

- Controller: this block is responsible for the logic of the feedback control loop. It computes the control commands to be sent to the vehicle based on the current position and orientation of the vehicle, and the current waypoint to be reached. The controller is implemented as a MATLAB function block, which allows for a more flexible implementation of the control logic. More details on the implementation of the controller are provided in the next sections;
- Command publisher: this block is used to publish the control commands to the `/cmd_vel` topic, which is used by the vehicle to receive the control commands.

Notice that the commands sent to the vehicle are comprehensive of both linear and angular velocities. In fact, a differential drive model is used internally by the robot to compute the wheel velocities based on the linear and angular velocities provided by the control system.
Inside the controller block, we can find the two different control strategies implemented for this assignment, namely a simple proportional controller and a more advanced controller based on Lyapunov analysis. See Figure 3 for an inside view of the controller block.

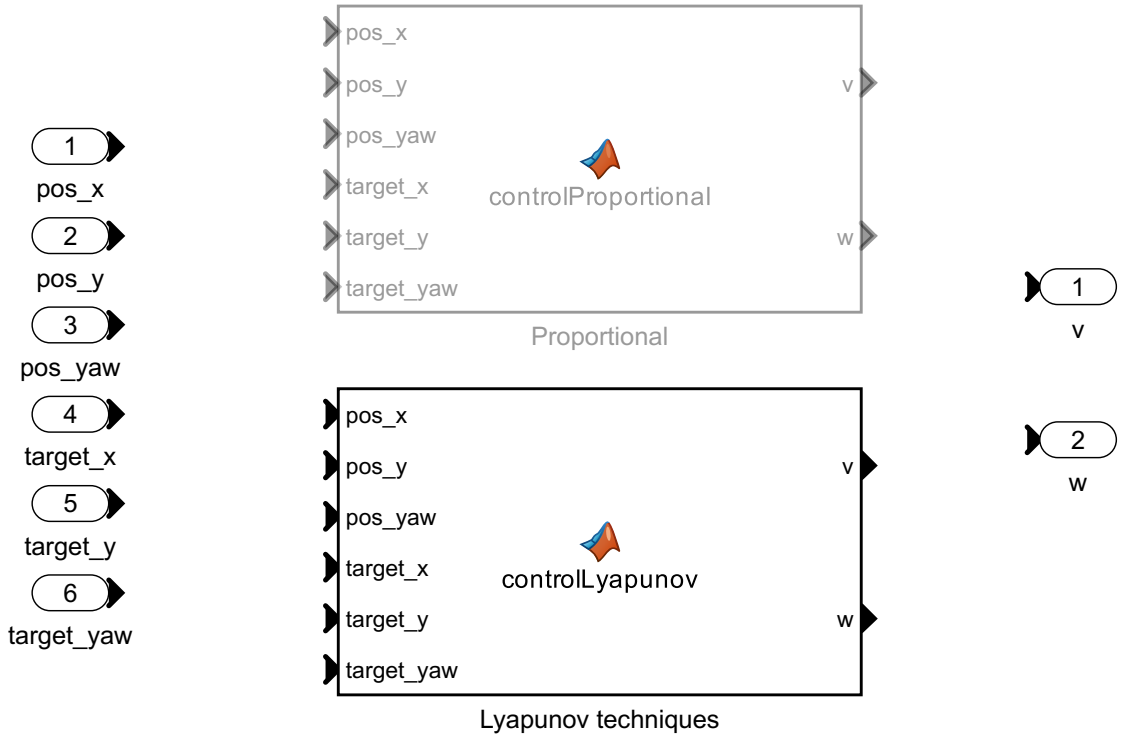

Figure 3: Inside view of the controller block.

3 Proportional controller

As explained in the previous section, one of the requests associated with this assignment is to implement a simple proportional controller for the `Turtlebot3` robot.
For this task, we have opted for a simple two-stage proportional controller. In particular, based on the distance of the robot from the current target waypoint, we apply a control input so that:

- If the robot is far from the target waypoint, we force its direction to be aligned with the target position, and we apply a linear velocity proportional to the distance from the target;
- Instead, when the robot is close to the target waypoint, we apply a control input that forces the robot to rotate in place, so that it can align its direction with the direction required by the target waypoint.

Some control constraints are also applied. In particular, we limit the maximum linear velocity to 0.2 [m/s] and the maximum angular velocity to 0.4 [rad/s]. Moreover, we consider the robot to be close to the target waypoint when the Euclidean distance from the target waypoint is less than 0.05 [m].

3.1 Implementation

The code for the implementation of the controller is provided in the Listing 1.

```
function [v, w] = controlProportional(pos_x, pos_y, pos_yaw, target_x, target_y,
    target_yaw)
% This controller align the robot with the target waypoint and guide it to
% reach it. Once within the threshold zone, correct the yaw of the robot.
% It uses simple Proportional controller for both actions.

bound = @(value, limit) max(min(value, limit), -limit);

% Gains
K_v = 1.2;
K_w = 1.5;
max_linear_speed = 0.2;
max_angular_speed = 0.4;

dx = target_x - pos_x;
dy = target_y - pos_y;
dist_to_target = hypot(dx, dy);
angle_to_target = atan2(dy, dx);

% Control logic
if dist_to_target > 0.05
    v = K_v * dist_to_target;
    w = K_w * wrapToPi(angle_to_target - pos_yaw);
else
    v = 0;
    w = K_w * wrapToPi(target_yaw - pos_yaw);
end

v = bound(v, max_linear_speed);
w = bound(w, max_angular_speed);

end
```

Listing 1: Code for the implementation of the proportional controller.

3.2 Results

Given the above implementation of the proportional controller, we can now analyze the results obtained during the simulation.
The results of the obtained trajectory are shown in Figure 4, where we report the waypoints highlighted in red and the trajectory of the robot drawn in black.

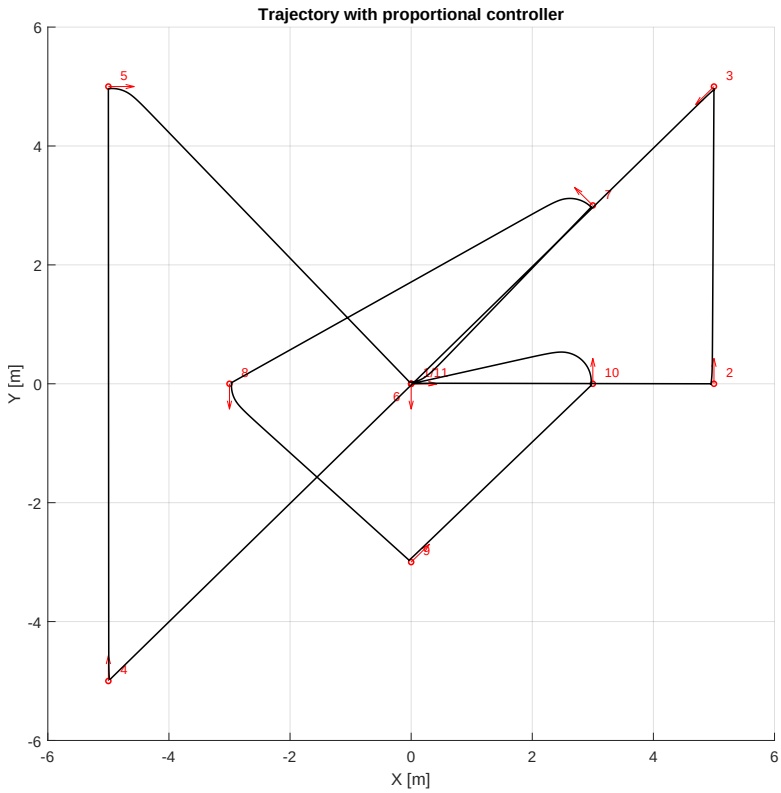

Figure 4: Trajectory of the robot during the simulation with the proportional controller.

One can easily observe that all the waypoints have been reached successfully.
When it comes to the velocity profiles, both in linear and angular dimension, it's easy to visualize the constraints applied to the controller. The linear velocity is limited to 0.2 [m/s], while the angular velocity is limited to 0.4 [rad/s]. This results in a trapezoidal profile for the velocities, as shown in Figure 5.

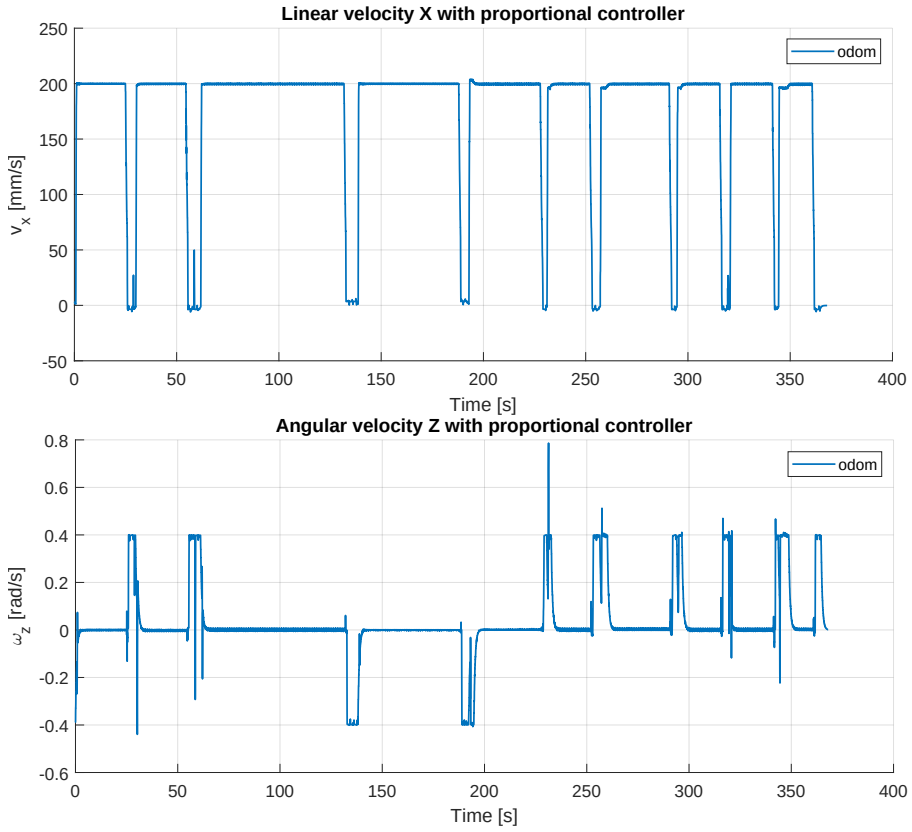

Figure 5: Velocity profiles of the robot during the simulation with the proportional controller.

One can also observe some severe (but luckily not too frequent) peaks in the velocities profiles that violate the constraints imposed. Similarly to the behavior observer in previous assignment, this fact find its explanation in poor performances of the machine on which the simulation is run. The simulation is run on a virtual machine with limited computational power, which results in some momentary oscillations in the robot's dynamics which are well captured by its telemetries. We think that a switch to a native environment of `Ubuntu 20.04` on a `Linux` kernel machine would solve this issue.

4 Lyapunov controller

In this section, we proceed with the implementation of a Lyapunov-based controller for the already defined system.
Lyapunov control theory is a powerful tool for designing controllers that proofs the stability of non-linear systems. The main idea is to find a Lyapunov function associated with the system and prove its convergence to the desired equilibrium point. Notice that global asymptotic stability is guaranteed only if the Lyapunov function is positive definite and radially unbounded.
Once the Lyapunov function is defined and its properties are verified, we can derive a control law that ensures the system's stability. Again, many methods from non-linear control theory exists such as the backstepping method, the feedback linearization method, or again passivation-based methods.
A final important remark here is that the control law derived from Lyapunov theory is not unique and generally speaking is also much more complex than the one derived from traditional linear control theory. Even if stability is guaranteed, the performance of the system may not be optimal and a proper tuning of the controller is not always straightforward.

4.1 Lyapunov control law

Based on the work of [1], we can derive a Lyapunov control law for a generic unicycle-like system. Given that the low level firmware of the `Turtlebot3` robot allows us to send control inputs in the form of linear and angular velocities, we can consider the turtlebot as a unicycle-like system and apply the Lyapunov control law to it.

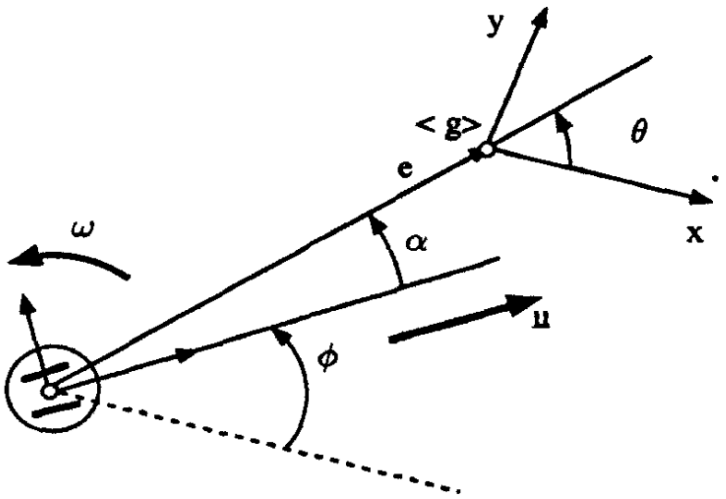

Figure 6: Schematic representation of the unicycle-like system and the reference frame.

Based on the notation used in Figure 6, we can define the following Lyapunov function:

$$v = k_1 \cos(\alpha) e \tag{1}$$

$$w = k_2 \alpha + k_1 \frac{\sin(\alpha)\cos(\alpha)}{\alpha} (\alpha + k_3 \theta) \tag{2}$$

Where k_1, k_2, and k_3 are positive constants, e is the Euclidean distance from the target waypoint, α is the angle between the robot heading and the target waypoint, and θ is the angle between the vector connecting the robot to the target waypoint and the waypoint direction (see Figure 6).

It turns out that with a proper remapping of the control inputs, we can obtain a Lyapunov function that is positive definite and radially unbounded, hence ensuring the global asymptotic stability of the system.

4.2 Implementation

The code for the implementation of the Lyapunov controller is provided in the Listing 2.

```
function [v, w] = controlLyapunov(pos_x, pos_y, pos_yaw, target_x, target_y, target_yaw)
% This controller implements a Lyapunov-based control law for the unicycle-like
% system. It uses a Lyapunov function to derive the control inputs for the robot.
% The controller is designed to ensure global asymptotic stability of the system.
% Inspired by: Aicardi et al. (1995)

bound = @(value, limit) max(min(value, limit), -limit);

% Limits
max_linear_speed = 0.8;
max_angular_speed = 1.6;

% Gains
K_rho   = 1.0;
K_alpha = 1.5;
K_delta = 1.0;

dx = target_x - pos_x;
dy = target_y - pos_y;
rho = hypot(dx, dy);
alpha = wrapToPi(atan2(dy, dx) - pos_yaw);
theta = wrapToPi(atan2(dy, dx) - target_yaw);

% Control law (from Aicardi paper)
v = K_rho * rho * cos(alpha);
w = K_alpha * alpha + K_rho * (sin(alpha) * cos(alpha)) + K_rho * (sin(alpha) * cos(
    alpha)) * K_delta * theta / max(abs(alpha), 1e-3) * sign(alpha);

v = bound(v, max_linear_speed);
w = bound(w, max_angular_speed);

end
```

Listing 2: Code for the implementation of the Lyapunov controller.

4.3 Results

The results of the obtained trajectory are shown in Figure 7, where we can see the waypoints highlighted in red and the trajectory of the robot drawn in black.

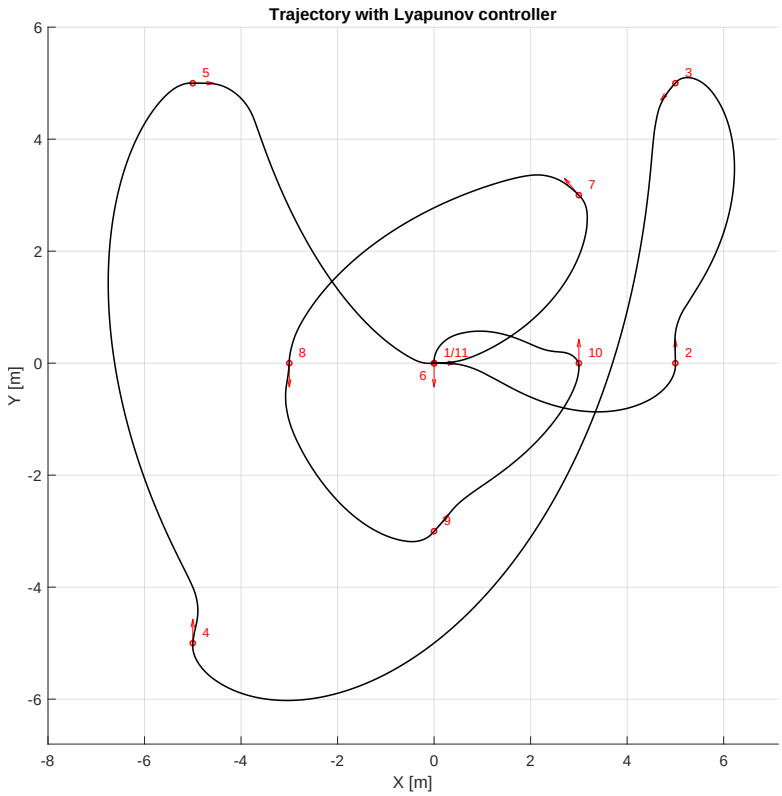

Figure 7: Trajectory of the robot during the simulation with the Lyapunov controller.

With respect to the proportional controller, we can see that the Lyapunov controller is able to smoothly go through the waypoints and to reach them avoiding stop-and-go behavior. The if-else logic is not needed any more given that the control law is able to smoothly connect the waypoints weighting the timing and amplitude for steering and velocity commands based on the 3 parameters k_1, k_2, and k_3. One could also modify the parameters to obtain a more aggressive or conservative behavior of the robot, with consequent changes in the trajectory.
When it comes to the velocities profile, we observe that linearity is lost and non-linear behavior is present.

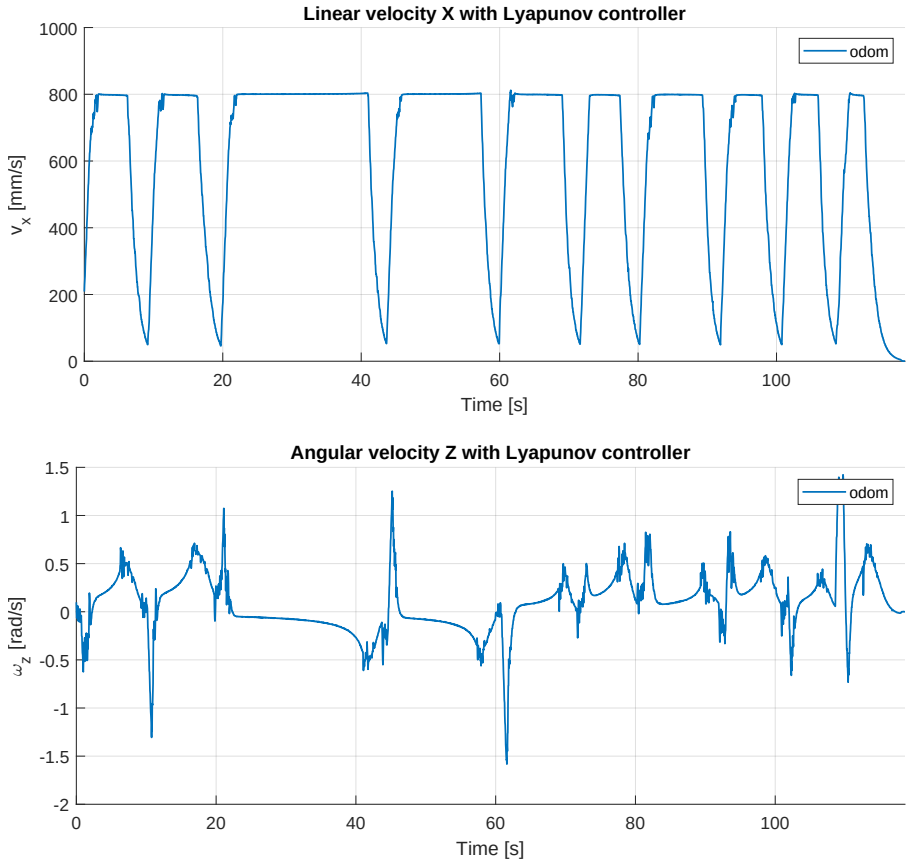

Figure 8: Velocity profiles of the robot during the simulation with the Lyapunov controller.

Notice that with respect to the simulation run with the proportional controller, the linear velocity is now not limited to 0.2[m/s], but it can reach up to 0.8[m/s], and similarly the angular velocity can reach up to 1.6[rad/s]. The author has decided to increase the limits of the velocities in order to allow the robot to reach the waypoints in a shorter time. Nevertheless, the no-slip condition generally required for the unicycle-like system has been checked and verified even with the presence of stronger inertial forces due to the higher velocities.

5 Conclusions

In this short report, we have presented the results of the second assignment of the course on Autonomous Vehicles. We have shown how to implement a feedback control strategy for a mobile robot, specifically the `Turtlebot3` robot, in a simulated environment using `ROS1` and `Gazebo`.
Even if not extensively discussed in this report, the `Simulink` system has allowed to easily load waypoints from different sources (i.e. workspace or ROS topic) and no differences in the results have been observed.
Most importantly, we have shown how to implement a simple proportional controller and a Lyapunov-based controller for the robot. While both have shown to be effective in guiding the robot to the target waypoints, the Lyapunov controller has demonstrated a native smoothness in the trajectory, avoiding stop-and-go behavior.
Some unwanted spikes in the telemetry of the robot have been observed during the simulation, which could be due to the fact that the simulation was run on a `Windows` environment connected to a `WSL` (Windows Subsystem for Linux). Further investigations and testing could be done by switching to a native `Linux` environment, which could provide better performance and more accurate results.

References

[1] M. Aicardi, G. Casalino, A. Bicchi, and A. Balestrino. Closed loop steering of unicycle like vehicles via lyapunov techniques. *IEEE Robotics & Automation Magazine*, 2(1):27–35, 1995.

[2] Stanford Artificial Intelligence Laboratory et al. Robotic operating system.

Autonomous Vehicles

Assignment IV: Algorithms for grid-based motion planning

Tommaso Bocchietti 10740309

A.Y. 2024/25

POLITECNICO
MILANO 1863

Contents

List of Figures

1 Introduction

The aim of this work is to gain insight into the implementation of some of the most basic grid-based planning algorithms, specifically the `Dijkstra` and `A*` algorithms.
The following sections will provide a detailed description of the requests associated with this assignment, the approach taken to fulfill them, and the discussion of the results obtained.

Tools As for the tools used, `MATLAB` is employed as the main platform for implementing the algorithms and performing the analysis of the data collected during the simulation.

2 Grid-based planning

In the field of robotics, a variety of algorithms are available for path planning, each with its own strengths and weaknesses. One of the most ancient and well known class of algorithms is the so called *Grid-based*, which is based on the discretization of the environment into a grid of cells. Once the environment is represented as a grid, the problem of path planning is reduced to a graph search problem, where the cells of the grid are the nodes of the graph and the edges are the connections between adjacent cells.

2.1 Requests

Among the graph search algorithms, two of the most widely known are the `Dijkstra` and its extension `A*` algorithm. The goal of this assignment can be summarized as follows:

- Implement both the `Dijkstra` and `A*` algorithms;
- Allow the possibility of orthogonal and diagonal movements;
- Test the algorithms on a set of predefined maps and compare the performances.

2.2 Graph search algorithms

In general, given a graph $G = (V, E)$, where V is the set of vertices and E is the set of edges, a graph search algorithm is used to find a path from a starting vertex $s \in V$ to a goal vertex $g \in V$. The main idea is that the algorithm explores the graph by expanding nodes and evaluating their neighbors until it finds the goal node or exhausts all possibilities.

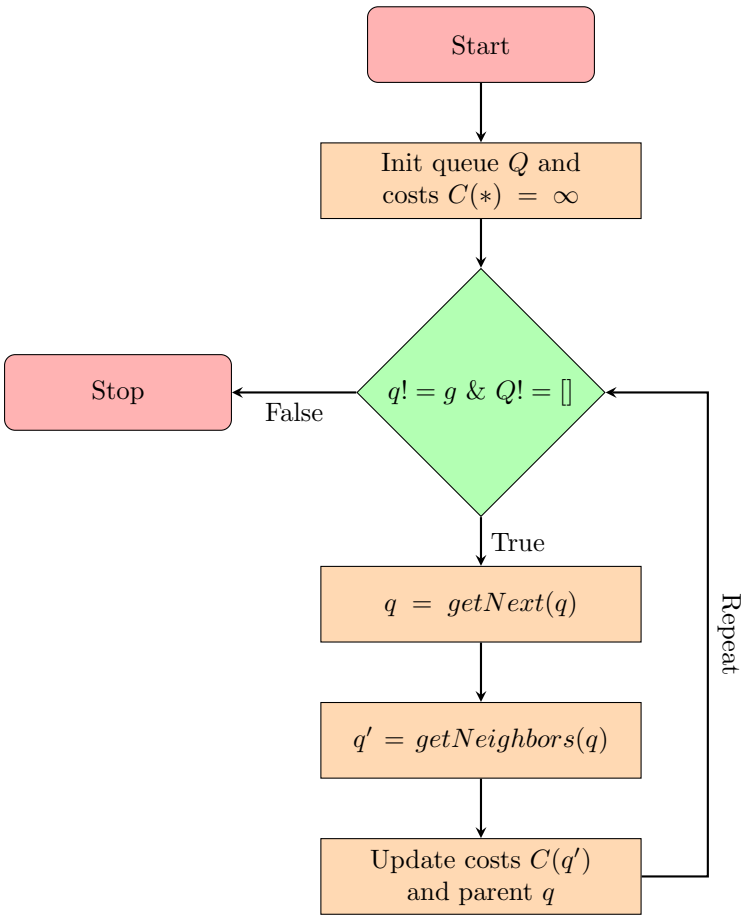

Figure 1: Generic graph search algorithm flowchart

The flowchart in Figure 1 summarizes the main steps of a generic graph search algorithm. The algorithm starts by initializing the open list of nodes and the costs of all nodes to infinity. Then, it enters a loop where based on the previous node visited, its neighbors are evaluated and added to the open list. For each neighbor, the algorithm updates the cost if lower than what already stored and in case updates also the parent node. The loop continues until the open list is empty, or the goal node is reached.
Grid-based algorithms have been proven to be complete and optimal, meaning that they will always find the shortest path if one exists. However, they can be computationally expensive, especially in large environments. There exist many variations built on top of the core algorithm presented in Figure 1. But, what really differentiates them, is the criteria used to select the next node to analyze from the open list. Some of the most common approaches are:

- **Depth-first search (DFS)**: explores as far as possible along each branch before backtracking. It uses a stack to keep track of the nodes to be explored. This approach can easily get stuck in loops.
- **Breadth-first search (BFS)**: explores all the neighbors of a node before moving on to the next level. It uses a queue to keep track of the nodes to be explored.
- **Best-first (a.k.a. Dijkstra)**: selects the node with the lowest cost from the start. This approach is optimal and complete, but can be slow in large environments.
- **A***: an extension of Dijkstra's algorithm that uses a heuristic to estimate the cost from the current node to the goal node. This approach is also optimal and complete, and is often faster than Dijkstra's algorithm in practice, requiring a lower number of nodes to be explored.

In the following sections, we will focus on the Dijkstra and A* algorithms, explaining their mechanisms in choosing the next node to be explored and how they can be implemented to solve the path planning problem in a grid-based environment.

2.2.1 Dijkstra's algorithm

As already mentioned, Dijkstra's algorithm leverages the core structure of the graph search algorithm presented in Figure 1, but with a specific strategy for selecting the next node to explore.

In particular, at each iteration, the algorithm selects the node among the nodes in the open list that has the lowest cost from the start node. The idea is so to select $q' \in Q$ such that:

$$q' = minarg_{q \in Q} C(q) \tag{1}$$

Where $C(q)$ is the cost of reaching node q from the start node s. The cost is calculated as the sum of the cost of the previous node q and the cost of moving from q to q'.
From an implementation point of view, the algorithm can be easily implemented using a priority queue to store the nodes in the open list. The priority queue allows for efficient retrieval of the node with the lowest cost, which is crucial for the performance of the algorithm. Listing 1 shows a possible implementation of the Dijkstra's algorithm. The algorithm takes as input the graph G, the start node s, and the goal node g.

```
function [exist, distances, parents] = dijkstra(G, node_initial, node_final)
% DIJKSTRA Finds the shortest path between two nodes in a graph
% using Dijkstra's algorithm.

N = length(G.nodes);
distances = inf(N, 1);
parents   = NaN(N, 1);
visited   = false(N, 1);

% Initialization
ID_current = node_initial.ID;
distances(ID_current) = 0;

while (ID_current ~= node_final.ID && ~all(visited))

    unvisited = find(~visited);
    [~, ID_current] = min(distances(unvisited));
    ID_current = unvisited(ID_current);

    visited(ID_current) = true;

    % For each neighbor not yet visited
    IDs_neighbors = G.adjacents{ID_current};
    IDs_neighbors = IDs_neighbors(~visited(IDs_neighbors));
    for ID_neighbor = IDs_neighbors'

        distance = ...
            distances(ID_current) + ...
            G.getEdgeByConnection(ID_current, ID_neighbor).weight;

        if distance < distances(ID_neighbor)
            distances(ID_neighbor) = distance;
            parents(ID_neighbor) = ID_current;
        end

    end

end

exist = ~isinf(distances(node_final.ID));

end
```

Listing 1: Code for the implementation of Dijkstra's algorithm.

The algorithm starts by initializing the distances of all nodes to infinity, except for the start node, which is set to zero. Then, it enters a loop where it selects the node with the lowest cost from the open list and explores its neighbors. For each neighbor, it updates the cost if the new cost is lower than the previously stored cost and updates the parent node accordingly. The loop continues until the goal node is reached, or all nodes have been visited. Finally, the algorithm returns the distances and parents of each node, which can be used to reconstruct the shortest path from the start node to the goal node.

2.2.2 A* algorithm

A* algorithm is an extension of Dijkstra's algorithm that uses a heuristic to estimate the cost from the current node to the goal node. The heuristic is a function that provides an estimate of the cost to reach the goal from a given node, and it is used to guide the search towards the goal more efficiently.
In particular, the A* algorithm selects the next node to explore based on the sum of the cost from the start node to the next node and the estimated cost from the next node to the goal node. The selection is done as follows:

$$q' = min\,\mathrm{arg}_{q \in Q}(C(q) + h(q)) \tag{2}$$

Where $h(q)$ is the heuristic function that estimates the cost from node q to the goal node g. The heuristic function can be any function that provides an estimate of the cost, but it is important that it is admissible, meaning that it never overestimates the true cost to reach the goal. For the sake of simplicity, and in particular leveraging the meaningful geometrical interpretation of the grid-based environment, we will use the Euclidean distance as the heuristic function. The A* algorithm can be implemented similarly to Dijkstra's algorithm, with the addition of the heuristic function. Listing 2 shows a possible implementation of the A* algorithm. The algorithm takes in the same input as the Dijkstra's one.

```
function [exist, distances, parents] = astar(G, node_initial, node_final)
% ASTAR Finds the shortest path between two nodes in a graph
% using the A* algorithm.

N = length(G.nodes);
distances = inf(N, 1);
costs     = inf(N, 1);
parents   = NaN(N, 1);
visited   = false(N, 1);

% Initialization
ID_current = node_initial.ID;
distances(ID_current) = 0;
costs(ID_current) = Node.euclideanDistance(node_initial.state, node_final.state);

while (ID_current ~= node_final.ID && ~all(visited))

    unvisited = find(~visited);
    [~, ID_current] = min(costs(unvisited));
    ID_current = unvisited(ID_current);

    visited(ID_current) = true;

    % For each neighbor not yet visited
    IDs_neighbors = G.adjacents{ID_current};
    IDs_neighbors = IDs_neighbors(~visited(IDs_neighbors));
    for ID_neighbor = IDs_neighbors'

        distance = ...
            distances(ID_current) + ...
            G.getEdgeByConnection(ID_current, ID_neighbor).weight;

        if distance < distances(ID_neighbor)

            distances(ID_neighbor) = distance;
            parents(ID_neighbor) = ID_current;

            node_neighbor = G.getNodeByID(ID_neighbor);
            costs(ID_neighbor) = distance + Node.euclideanDistance(node_neighbor.state,
    node_final.state);

        end

```

```
43      end
44
45  end
46
47  exist = ~isinf(distances(node_final.ID));
48
49  end
```

Listing 2: Code for the implementation of A* algorithm.

One can clearly notice that the structure of the A* algorithm is basically the same as the Dijkstra's algorithm, except for the addition of the heuristic function in the cost calculation. Notice also that the heuristic function is only used to update the cost of the nodes in the open list, and it is not used to update the distances of the nodes from the start node.
Given that now the algorithms is somehow guided, one can expect that the A* algorithm will be faster than the Dijkstra one, as fewer nodes will be explored.

3 Algorithm Testing

Now that the algorithm has been implemented, we can proceed presenting at first the test environment that have been used to test them and then the results of the tests.
Figures 2 show the four image maps used to test the algorithm. Even if the images here below are of size 300x300, we preferred to scale down to 30x30 given that each pixel is then converted into a node of the graph and so to avoid huge graphs and long computation times. Notice also that while the white areas are free space, the one marked in black or gray are to be considered as obstacles. Finally, the green dot is the starting point of the vehicle, while the red one is the goal.

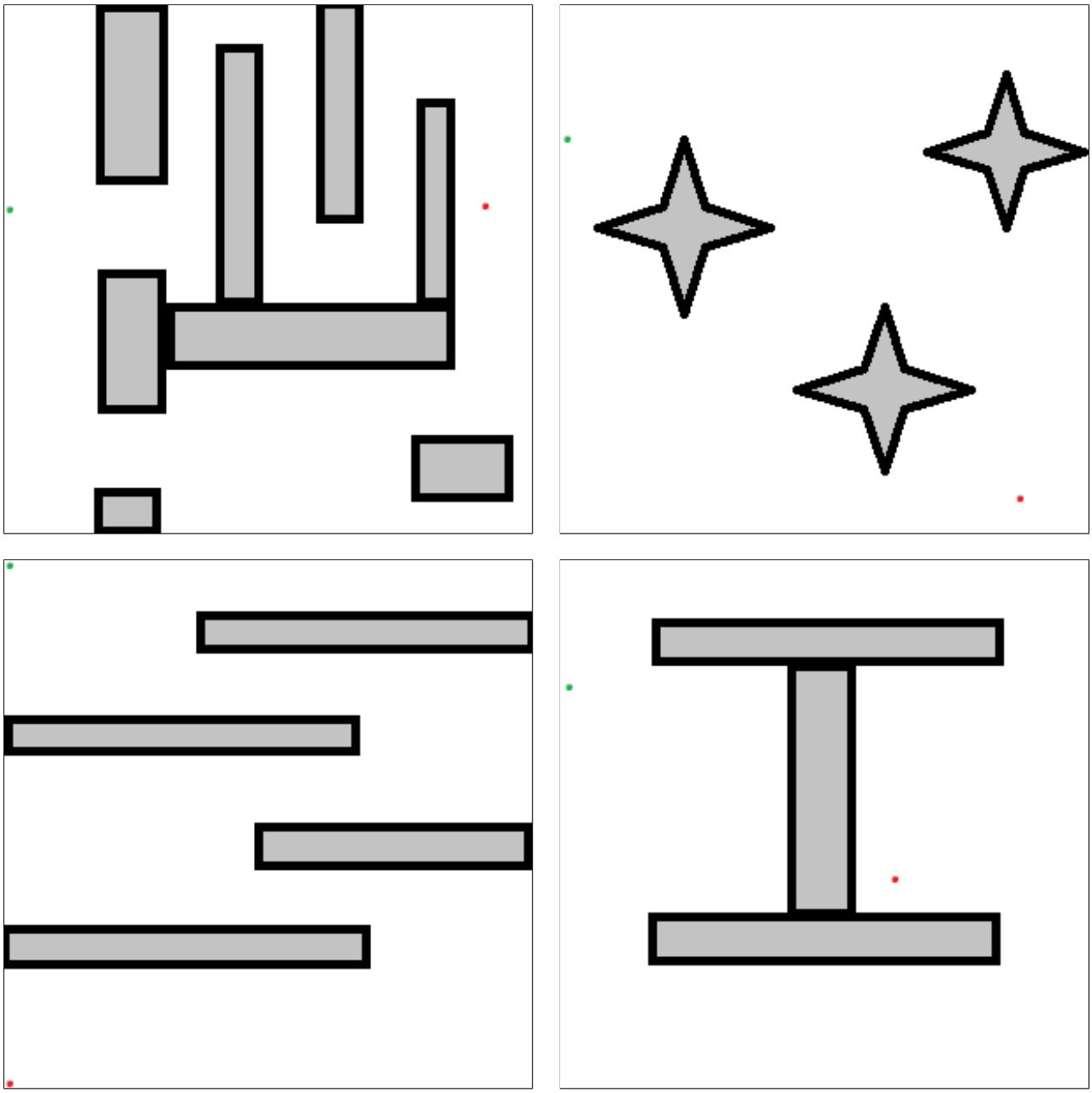

Figure 2: Test maps used for the algorithm testing.

In the following, results obtained running both the A* and the Dijkstra algorithms on each of the four maps are presented.
The plotting of both the (geometrical) connected graph and the best path found is also shown. Notice that each test case has been run allowing or disabling the diagonal movement of the vehicle, which can be achieved during the graph generation phase modifying the connection of the nodes.
Tables presenting the main metrics of the tests are also shown. The metrics are the following:

- **Elapsed time**: here only the time needed to run the searching algorithm is considered (i.e. we are excluding both the generation of the graph and the plotting of the results);
- **Nodes analyzed**: number of nodes analyzed (visited) before reaching the goal;
- **Path length**: length of the path found by the algorithm. This also correspond to the path cost, given that the cost of each edge is equal to its geometrical length.

3.1 Map 1

Results obtained running the A* and Dijkstra algorithms on Map 1 are shown in Figures 3 and Table 1. About the figures, the upper row shows the graph generated running the Dijkstra algorithm while the lower one shows

the graph generated running the A* algorithm. The left column shows the graph generated allowing only orthogonal movements, while the right one shows the graph generated allowing also diagonal movements. The path found by the algorithm is shown in red, while the graph is shown in black.

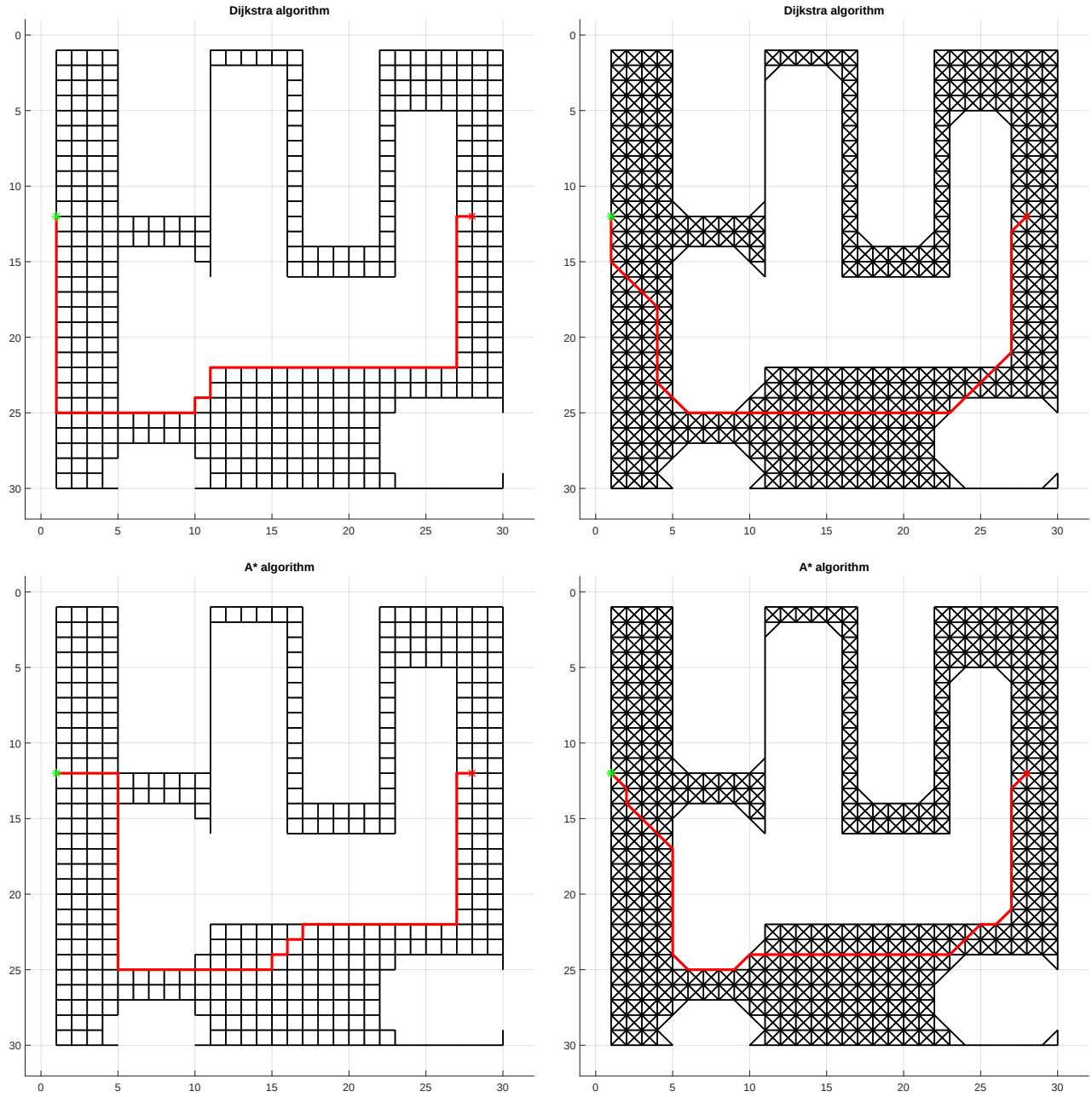

Figure 3: Map 1: Graph and path found by the algorithms.

Algorithm	Diagonal	Elapsed time (ms)	Nodes analyzed	Path length (m)
Dijkstra	No	624	456	53
	Yes	1592	453	47
A*	No	339	378	53
	Yes	1227	339	47

Table 1: Map 1: Results of the tests.

From the results presented in Table 1 and in Figure 3, we can already notice that while the cost of the path found by the two algorithms is equivalent, the number of nodes considered by the A* algorithm is significantly lower than the one considered by the Dijkstra algorithm. This behavior is expected due to the use of the heuristic function to guide the search of the A* algorithm. The advantage in terms of computational time is also evident, with the A* algorithm outperforming the Dijkstra algorithm.

3.2 Map 2

Results obtained running the A* and Dijkstra algorithms on Map 2 are shown in Figures 4 and Table 2. The same layout of the previous map is used for the figures and the same metrics are used for the table.

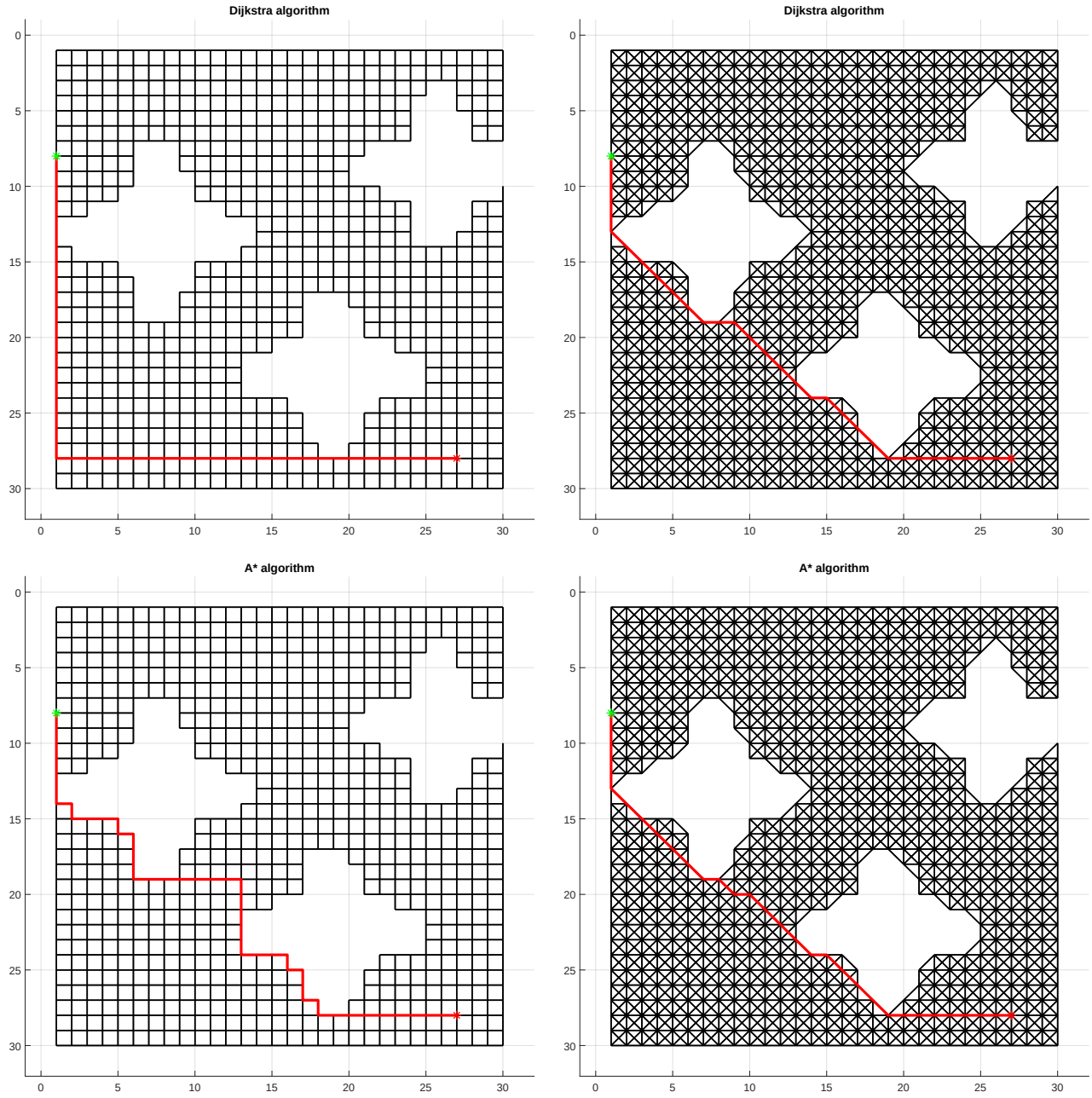

Figure 4: Map 2: Graph and path found by the algorithms.

Algorithm	Diagonal	Elapsed time (ms)	Nodes analyzed	Path length (m)
Dijkstra	No	1009	731	46
	Yes	3963	742	37
A*	No	732	490	46
	Yes	1434	233	37

Table 2: Map 2: Results of the tests.

Similarly to the previous map, the results presented in Table 2 and in Figure 4 show that while the cost of the path found by the two algorithms is equivalent, the number of nodes considered by the A* algorithm is significantly lower than the one considered by the Dijkstra algorithm. Similar conclusions can be drawn in terms of computational time and performance of the two algorithms.

As an additional note, in this case we can also appreciate that the number of analyzed nodes by A* is significantly lower than the one analyzed by Dijkstra, particularly in case of diagonal movements. This is, however, expected. Looking at the final path, we observe that is almost equivalent to the diagonal connecting ideally the start and goal points. This means that the heuristic function used by A* is even more effective in this case, given that very few obstacles are present along the optimal path designed by the heuristic function itself and so fewer death ends are encountered. In other words, if the optimal path turns out to be a straight line connecting the start and goal points, the A* algorithm would analyze a number of nodes equal to the number of nodes along the straight line, while the Dijkstra algorithm would still analyze a much larger number of nodes given that no information is used to guide the search.

3.3 Map 3

Results obtained running the A* and Dijkstra algorithms on Map 3 are shown in Figures 5 and table. The same layout of the previous map is used for the figures and the same metrics are used for the table.

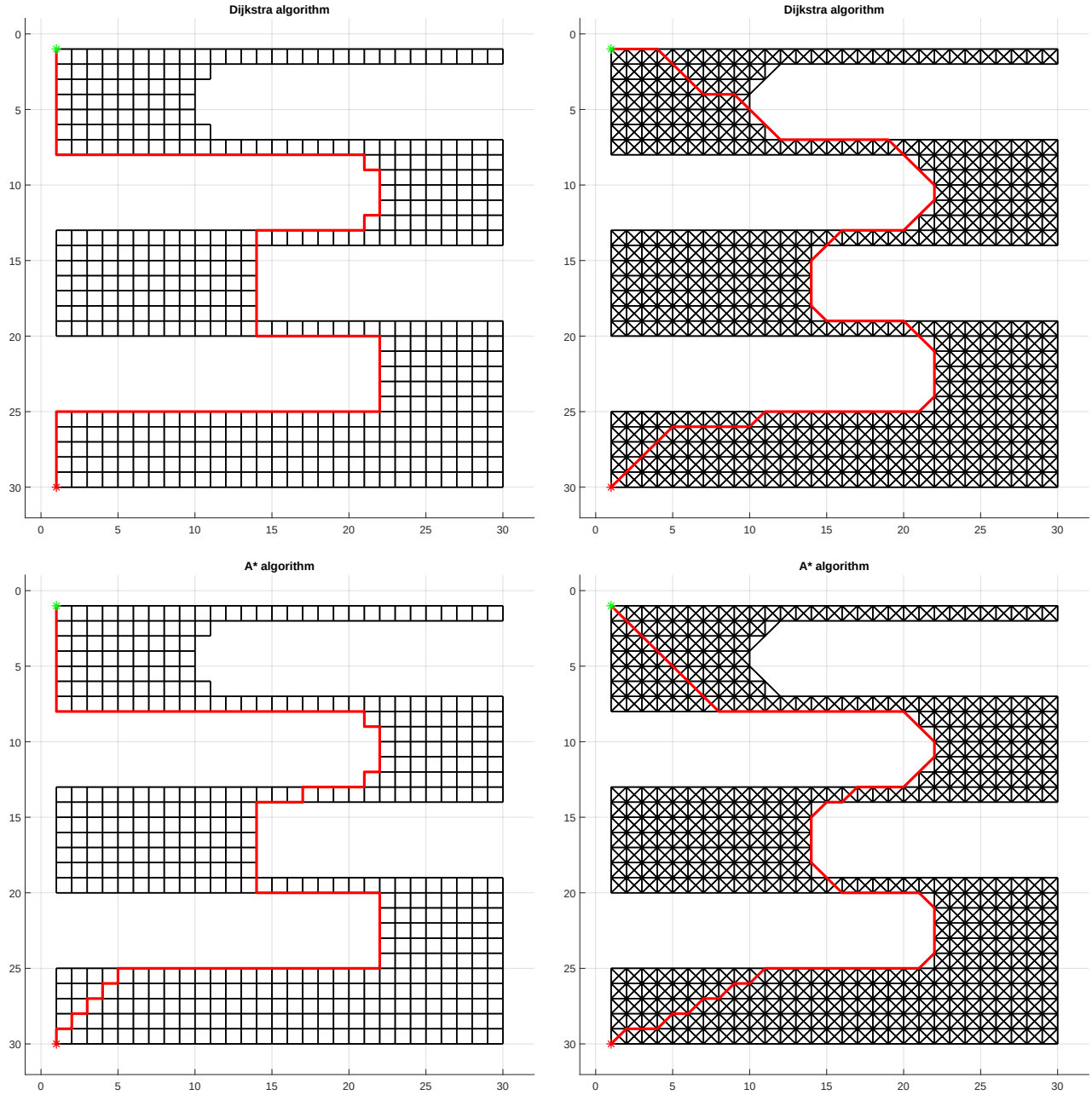

Figure 5: Map 3: Graph and path found by the algorithms.

Algorithm	Diagonal	Elapsed time (ms)	Nodes analyzed	Path length (m)
Dijkstra	No	567	592	87
	Yes	2461	592	74
A*	No	496	497	87
	Yes	1780	452	74

Table 3: Map 3: Results of the tests.

Again, similar considerations as before can be drawn from the results presented in Table 3 and in Figure 5.

3.4 Map 4

Results obtained running the A* and Dijkstra algorithms on Map 4 are shown in Figures 6 and table. The same layout of the previous map is used for the figures and the same metrics are used for the table.

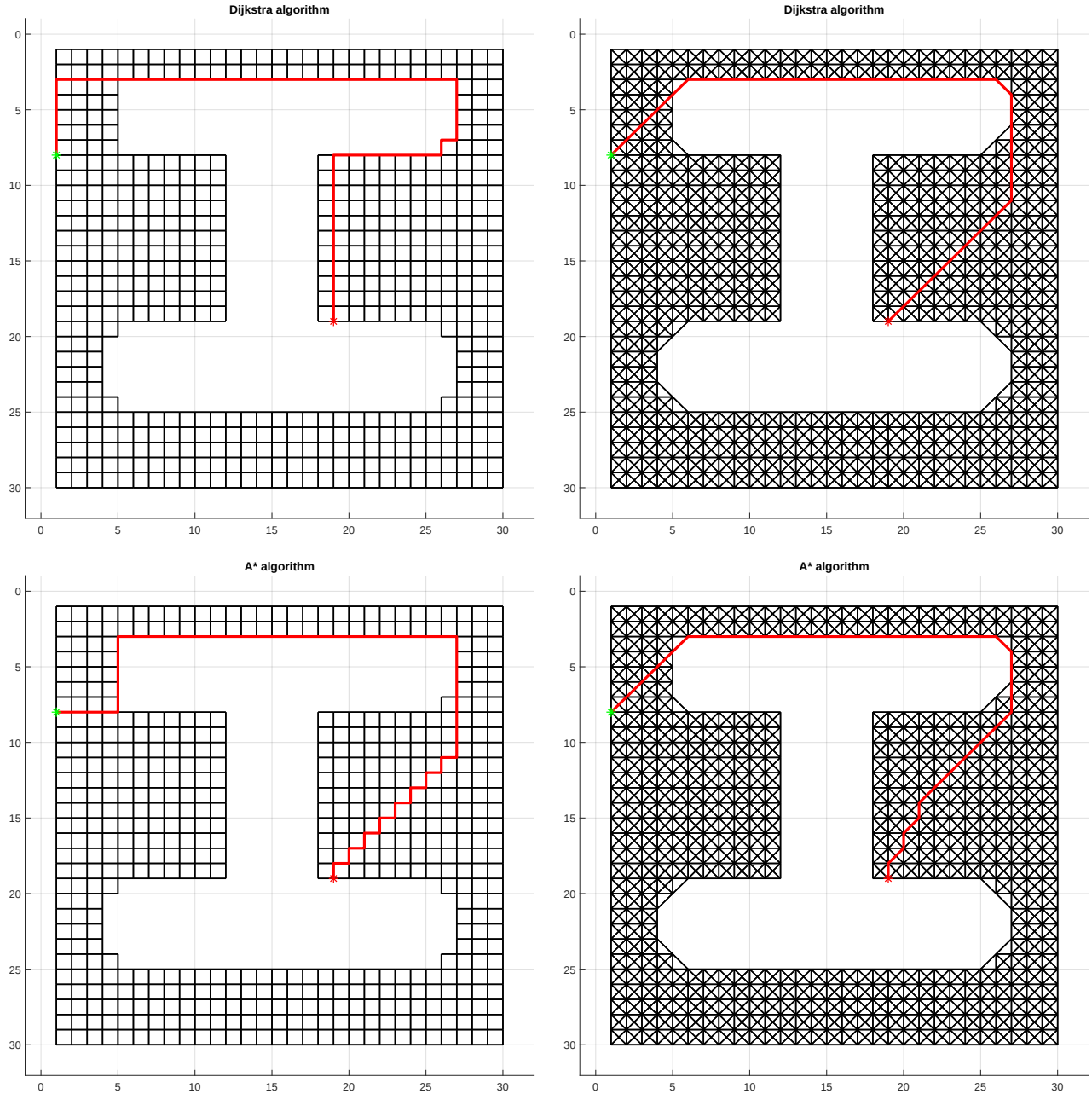

Figure 6: Map 4: Graph and path found by the algorithms.

Algorithm	Diagonal	Elapsed time (ms)	Nodes analyzed	Path length (m)
Dijkstra	No	725	650	55
	Yes	2799	650	47
A*	No	640	562	55
	Yes	2060	450	47

Table 4: Map 4: Results of the tests.

Once again, even from the tests on Map 4, similar conclusions as before can be drawn from the results presented in Table 4 and in Figure 6.

4 Conclusions

In this short report, we have presented the results of the fourth assignment of the course on Autonomous Vehicles. We have briefly explained the grid-based algorithm for path planning, and deepened into the implementation of both the Dijkstra and A* algorithms.
These algorithms have been tested on four different scenarios, with different obstacles and connectivity configurations (i.e., allowing or not diagonal movements). The results have shown that the A* algorithm is generally faster than the Dijkstra algorithm as he uses heuristics to guide the search towards the goal allowing for a more efficient exploration of the search space. This is also reflected in the number of nodes explored, which is significantly lower for the A* algorithm compared to the Dijkstra algorithm.
As also proved by the theroetical analysis performed for this grid-based algorithm, both algorithms are complete and optimal, meaning that they will always find the shortest path if one exists. Obtained results have reflected this theoretical analysis, as both algorithms have been able to find the optimal path in all tested scenarios.
One of the main limitations of the grid-based algorithm is that they can be computationally expensive, especially in large or hiperdimensional spaces. A quick solution to this problem is to use a coarser grid, which can reduce the number of nodes and edges in the graph, but at the cost of accuracy and possibly missing the optimal path, or even failing to find a path at all.
Because of the above limitation, the grid-based algorithm is not suitable for all applications, especially in dynamic environments where the obstacles can change over time. In these cases, more advanced algorithms based on combinatorial techniques or sampling-based methods, such as Rapidly-exploring Random Trees (RRT) or Probabilistic Roadmaps (PRM), may be more appropriate.

A MATLAB Graph implementation

During the implementation of the Dijkstra and A* algorithms, a needs for a solid and efficient graph representation arose. In order to achieve this, a set of MATLAB classes was created to represent the graph, its nodes and its edges.
In the following, the complete code used for this purpose is shown. One can refer back to the graph search algorithm implementation to understand how these classes are used.

```
classdef Graph < handle
    % GRAPH Class for representing a graph structure with nodes and weighted edges.
    % Provides methods for adding and retrieving nodes, edges, and adjacency data.

    properties
        nodes Node
        edges Edge
        adjacents cell
    end

    methods
        function obj = Graph()
            obj.nodes = Node.empty;
            obj.edges = Edge.empty;
            obj.adjacents = {};
        end

        function ID = addNode(obj, state)
            ID = length(obj.nodes) + 1;
            obj.nodes(ID, 1) = Node(ID, state);
            obj.adjacents{ID, 1} = [];
        end

        function ID = addEdge(obj, ID_A, ID_B, weight)
            ID = length(obj.edges) + 1;
            obj.edges(ID, 1) = Edge(ID, ID_A, ID_B, weight);
            obj.adjacents{ID_A, 1}(end+1, 1) = ID_B;
        end

        function node = getNodeByID(obj, ID)
            node = obj.nodes(ID);
        end

        function edge = getEdgeByID(obj, ID)
            edge = obj.edges(ID);
        end


        function nodes = getNodesByState(obj, state)
            nodes = Node.empty;
            for node = obj.nodes'
                if Node.compareStates(node.state, state)
                    nodes(end+1) = node;
                end
            end
        end


        function edge = getEdgeByConnection(obj, ID_A, ID_B)
            for edge = obj.edges([obj.edges.ID_A] == ID_A)'
                if (edge.ID_B == ID_B)
                    return
                end
```

```
            end
        end


        function adjacents = getAdjacents(obj, node_ID)
            adjacents = obj.adjacents{node_ID};
        end
    end

end
```

Listing 3: Graph class used to represent a weighted and directed/undirected graph.

```
classdef Edge
    % EDGE Represents a weighted connection between two nodes in a graph.
    % Stores IDs of connected nodes and the edge weight.

    properties
        ID double
        ID_A double
        ID_B double
        weight double
    end

    methods
        function obj = Edge(ID, ID_A, ID_B, weight)
            obj.ID = ID;
            obj.ID_A = ID_A;
            obj.ID_B = ID_B;
            obj.weight = weight;
        end

        function isConnection = isConnectionTo(obj, ID_A, ID_B)
            isConnection = obj.ID_A == ID_A & obj.ID_B == ID_B;
        end
    end

end
```

Listing 4: Edge class used to represent a weighted and directed edge in the graph.

```
classdef Node
    % NODE Represents a graph node with a unique ID and a state vector.
    % Includes utility methods for comparison and distance calculation.

    properties
        ID double
        state double
    end

    methods
        function obj = Node(ID, state)
            obj.ID = ID;
            obj.state = state;
        end

        function isEqual = eq(node_A, node_B)
            isEqual = Node.compareStates(node_A.state, node_B.state);
        end
    end

```

```
    methods (Static)
        function dist = euclideanDistance(state_A, state_B)
            dist = norm(state_A - state_B);
        end

        function equal = compareStates(state_A, state_B)
            equal = isequal(state_A, state_B);
        end
    end

end
```

Listing 5: Node class used to represent a node in the graph.

As a final note, a code profiler was used to test the performance of the graph implementation. The main bottleneck was found in the `getEdgeByConnection` method of the `Graph` class, as shown in Figure 7.

Generated 27-Apr-2025 23:28:19 using performance time.

Function Name	Calls	Total Time (s)	Self Time* (s)	Total Time Plot (self time / total time)
Assignment_4	1	7.505	0.010	
dijkstra	1	6.593	0.083	
Graph>Graph.getEdgeByConnection	2665	6.510	6.510	
imgToGraph	1	0.673	0.140	
Graph>Graph.addEdge	5398	0.325	0.292	
reconstructSequence	1	0.143	0.143	
Graph>Graph.getNodeByID	6154	0.118	0.118	
Graph>Graph.getNodesByState	2	0.044	0.040	

Figure 7: Profiler output for the graph implementation.

The author of this report is aware that the graph implementation can be improved in terms of performance by using perhaps a caching mechanism or a more efficient data structure. However, this was not the main focus of the assignment and further investigation is left as future work.

Autonomous Vehicles
Assignment V: Finite State Machines

Tommaso Bocchietti 10740309

A.Y. 2024/25

POLITECNICO
MILANO 1863

Contents

List of Figures

1 Introduction

The aim of this work is to gain insight into the modelling of systems via Finite State Machines (FSMs) and to understand the principles behind their implementation in the context of autonomous vehicles.
At first, a brief overview of the concepts of FSMs comprehensive of simple examples is provided. Then, the focus shifts to the modelling of a simple application, namely a parking gate system, which is designed using FSMs.

Tools As for the tools used, `Simulink` and in particular `Simulink/Stateflows`, is employed as the main tool for implementing the FSM and simulating the system. On the other hand, `MATLAB` is simply used to plot the results of the simulation.

2 Finite State Machines

Finite state machines (FSMs) are a powerful method of modelling systems whose output depends on the entire history of their inputs (as opposed to memoryless systems, where the output depends on the current input only). FSMs are widely used in various fields, including computer science, control systems, and digital circuit design. They can be classified into two main classes: deterministic finite state machines (DFSMs) and non-deterministic finite state machines (NDFSMs).
Mathematically, a finite state machine is defined as a 6-tuple $(S, I, O, \delta, \lambda, s_0)$, where:

- S is a finite set of states.
- I is a finite set of input symbols (input alphabet).
- O is a finite set of output symbols (output alphabet).
- $\delta : S \times I \to S$ is the state transition function, which maps a state and an input symbol to the next state.
- $\lambda : S \times I \to O$ is the output function, which maps a state and an input symbol to an output symbol.
- $s_0 \in S$ is the initial state.

The FSM processes a sequence of input symbols, transitioning between states according to the state transition function δ and producing output symbols according to the output function λ. The behavior of an FSM can be represented using state transition diagrams or state transition tables.

2.1 Turnstile FSM

As an example, we consider a turnstile that allows entry when a coin is inserted and locks when a person pushes through. The FSM for this turnstile can be represented by the state transition diagram in Figure 1 [1] and the state transition table in Table 1 [1].

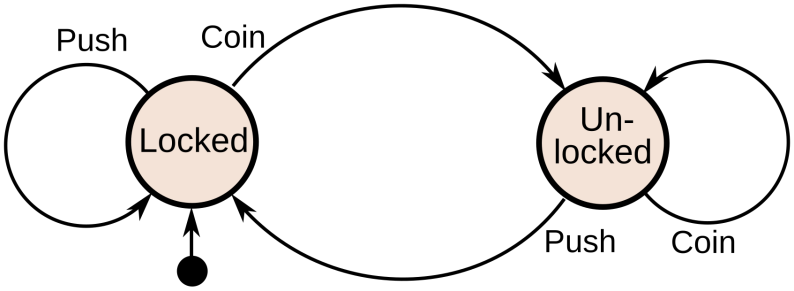

Figure 1: Turnstile FSM State Transition Diagram

Current State	Input	Next State	Output
Locked	Coin	Unlocked	Unlocks the turnstile so that the customer can push through
	Push	Locked	None
Unlocked	Coin	Unlocked	None
	Push	Locked	When the customer has pushed through, locks the turnstile

Table 1: Turnstile FSM Transition Table

The FSM starts in the *Locked* state. When a coin is inserted, it transitions to the *Unlocked* state and allows the customer to push through. If the customer pushes through while in the *Unlocked* state, the FSM transitions back to the *Locked* state. If a coin is inserted while in the *Unlocked* state, the FSM remains in the same state.

3 Parking Gate System

As already mentioned in the previous sections, the focus of this assignment is to model a parking gate system using finite state machines (FSMs).

3.1 Requests

The requests that the parking gate system has to fulfil are the following:

- Analyze the provided *Vehicle FSM* and understand how it works;
- Define the *raise* and *lower* functions of the gate respecting $w_{max} = |4|[^\circ/s]$ and $\dot{w}_{max} = |1|[^\circ/s^2]$;
- Define the FSM to model the parking gate control system;
- Integrate the *Vehicle FSM* and the parking gate control system FSM;
- Provide results of the simulation of the integrated system;

Notice that despite the suggestion from the professor to start from the already provided *Vechicle FSM*, I decided to start from scratch in order to develop a deepened understanding of the FSMs, modelling also events, slightly more complex logics and composition / decomposition patterns.

3.2 System Overview

The parking gate system is composed of a single Finite State Machine (FSM) that is responsible for modelling at the same time the vehicle and the parking gate system.

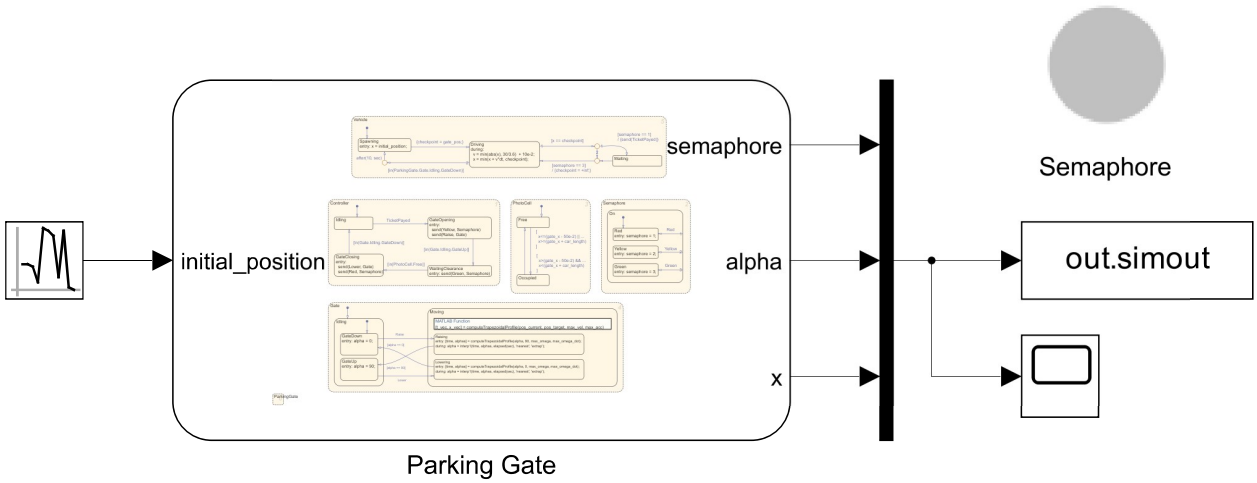

Figure 2: Parking Gate System Overview

Under the hood of the unique FSM, the system is composed of other two parallel FSMs, with precise and well-defined roles and responsibilities. The communication between the two FSMs is done through events and state transitions, which are used to inform each other about the current state of the system and to trigger the appropriate actions.

- The *Vehicle FSM* share the *vehicle_position (x)* with the *Parking Gate FSM* to inform it about the vehicle's position in the parking lot;
- The *Parking Gate FSM* share the *semaphore_state* with the *Vehicle FSM* to inform it about the state of the gate (closed, opening, opened);
- The *Vehicle FSM* also sends the event about the *TicketPayed* to the *Parking Gate FSM* to inform it that the vehicle has paid for the ticket and is ready to leave the parking lot;

Figure 2 shows the top level of the parking gate system, while Figure 3 shows the parking gate system FSM.

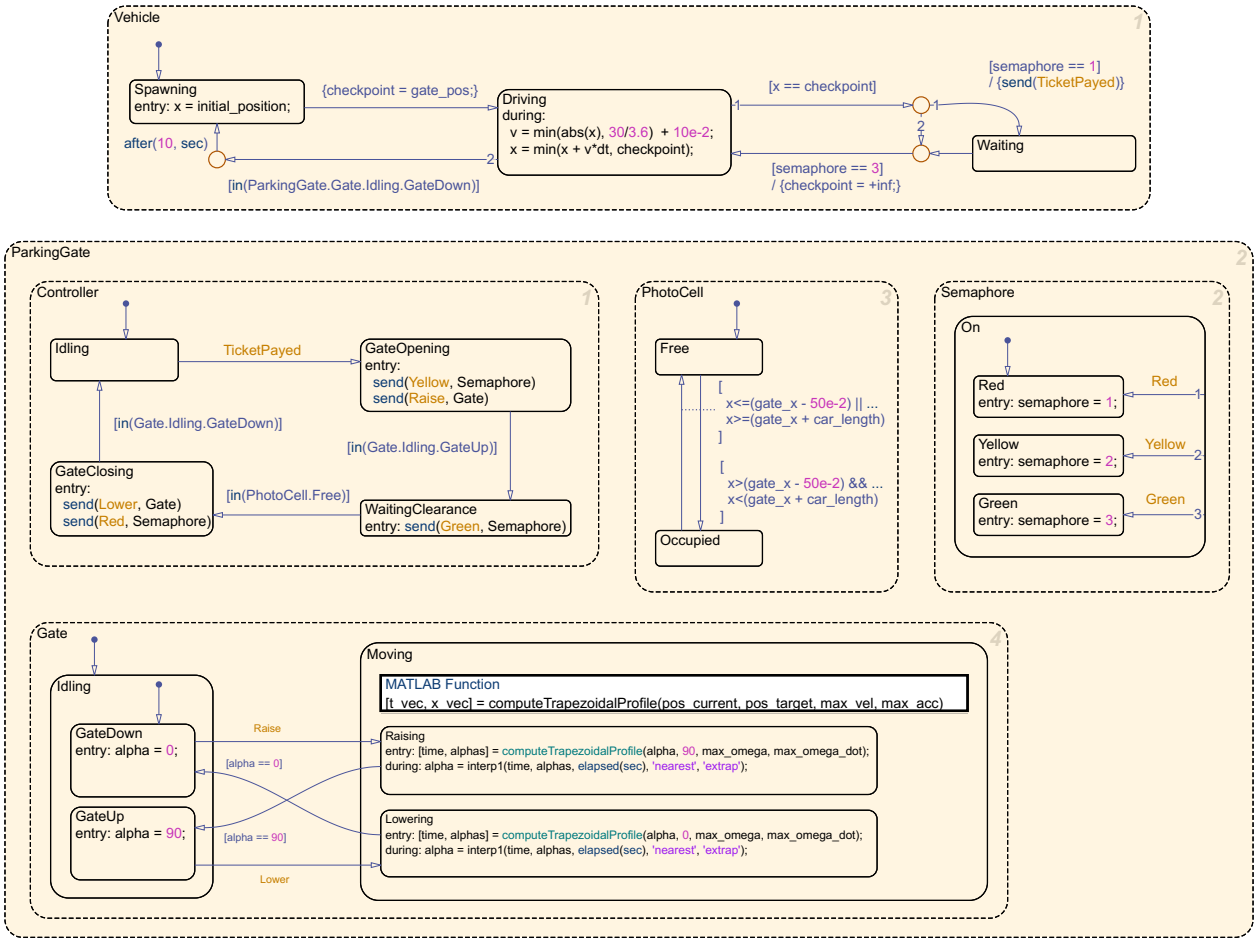

Figure 3: Parking Gate System FSM

3.2.1 Vehicle FSM

The vehicle FSM is responsible for simulating the vehicle's behavior in the parking lot. It has two main states: *Driving* and *Waiting*. At first, the vehicle is initialized and given a random position ahead of the gate. A checkpoint is also set so that it will stop at the gate before proceeding to the next state. Once at the gate, the vehicle looks for the semaphore state: if the semaphore is green, it directly returns to the *Driving* state and proceeds ahead to the next checkpoint which is now set to $+\infty$; instead, if the semaphore is red, it simulates the payment of the ticket by sending the event *TicketPayed* to the parking gate FSM and goes into the *Waiting* state, waiting for the semaphore to turn green. The vehicle proceed until the gate is closed again, when the external environment resets the vehicle to a new random position ahead of the gate to start the process again. Figure 4 shows the vehicle FSM.

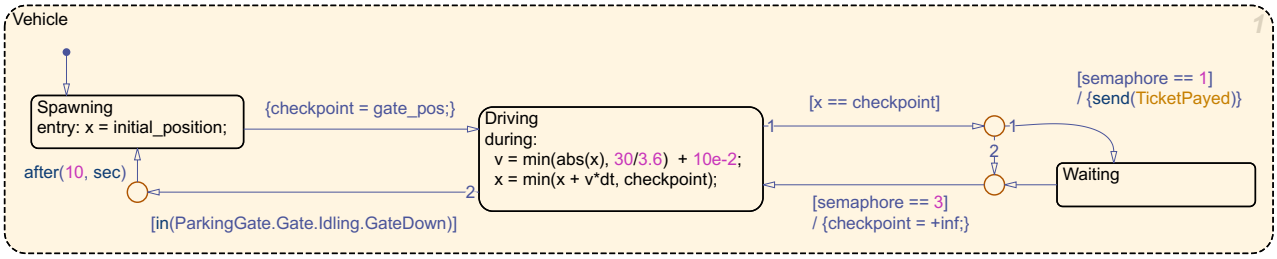

Figure 4: Vehicle FSM

Notice that the vehicle FSM has been modelled adopting the default *OR* logic, meaning that the vehicle can assume only one state at a time (i.e. *Driving* or *Waiting*, or *Spawning* when the vehicle is initialized).

3.2.2 Parking Gate FSM

The parking gate FSM is responsible for controlling at first the gate's opening and closing, and then the semaphore state. It has four main blocks: *Controller*, *Gate*, *Semaphore* and *PhotoCell*.

Controller The entry point of the parking gate FSM is the *Controller* block, which is responsible for managing the overall state of the system and coordinating the other blocks. It has been designed to be event-driven, meaning that communicates with the subcomponents of the system by sending and receiving events. It has four main states: *Idling*, *GateOpening*, *WaitingClearance* and *GateClosing*. At first, the *Controller* block is in the *Idling* state, waiting for the vehicle to arrive and receive the *TicketPayed* event. Once the ticket is paid, the *Controller* block moves to the *GateOpening* state, where it sends the *Raise* event to the *Gate* block and the *Yellow* event to the *Semaphore* block. Once the gate is fully opened, the semaphore turns green and the *Controller* block moves to the *WaitingClearance* state, where it waits for the vehicle to pass through the gate. Notice that the system determines the vehicle's position by using the *PhotoCell* block, which is responsible for detecting the vehicle's presence in front of the gate. Once the vehicle has passed through the gate, the *Controller* block moves to the *GateClosing* state, where it sends the *Lower* event to the *Gate* block and the *Red* event to the *Semaphore* block. The *Controller* block then moves back to the *Idling* state, waiting for the next vehicle to arrive.
Figure 5 shows the *Controller* block of the parking gate FSM.

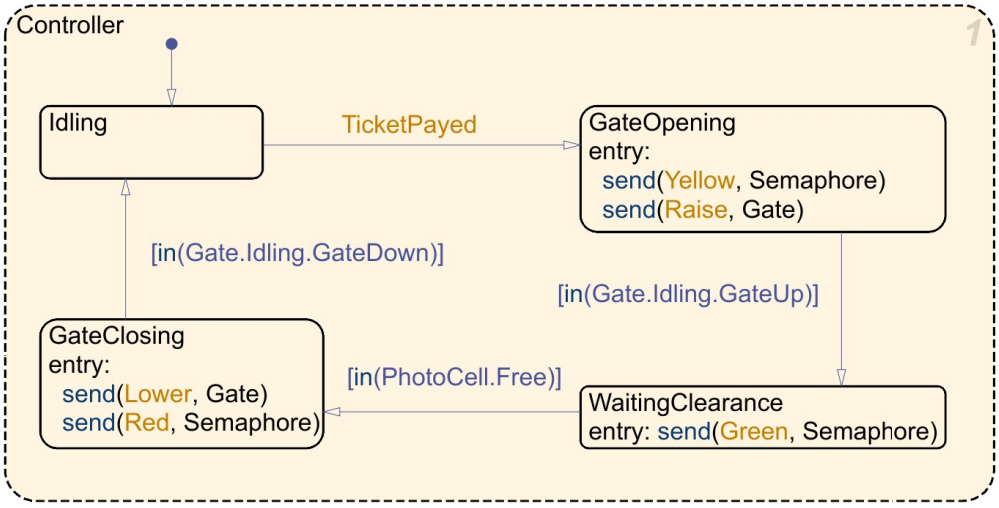

Figure 5: Control Block

Gate As already mentioned, the *Gate* block is responsible for controlling the gate's position and ensuring that it opens and closes correctly. It's designed based on two hierarchical states: *Idling* and *Moving*, which then are further divided into *GateUp* and *GateDown* states, and *Raising* and *Lowering* states, respectively. At first, the *Gate* block is in the *GateDown* state, where the gate is closed and the vehicle is not allowed to pass through. Once the *Raise* event is received from the *Controller* block, the *Gate* block moves to the *Raising* state, where it starts to open the gate. Once the gate is fully opened, the *Gate* block moves to the *GateUp* state waiting for

the *Lower* event from the *Controller* block that will then perform the same steps in reverse order, moving to the *Lowering* state and then to the *GateDown* state once the gate is fully closed.
Figure 6 shows the *Gate* block of the parking gate FSM.

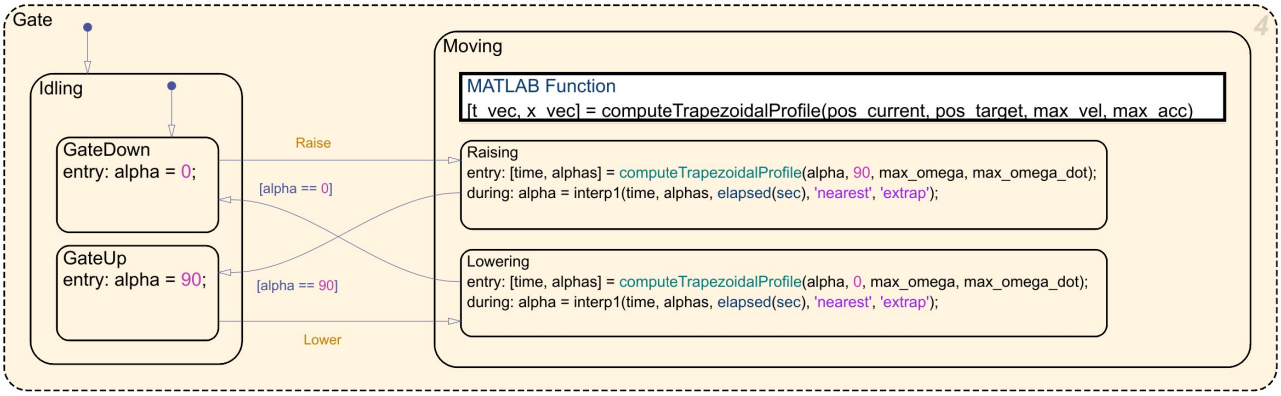

Figure 6: Gate Block

Notice that, in order to respect the constraints $w_{max} = |4|[^\circ/s]$ and $\dot{w}_{max} = |1|[^\circ/s^2]$, the position of the gate is updated based on the trapezoidal motion profile, generated when entering the *Raising* or *Lowering* states. The trapezoidal motion profile is defined by the following equations:

$$w(t) = \begin{cases} \frac{w_{max}}{t_1} t & 0 \leq t < t_1 \\ w_{max} & t_1 \leq t < t_2 \\ w_{max} - \frac{w_{max}}{t_3}(t - T) & t_2 \leq t < T \end{cases} \tag{1}$$

Where t_1 is the time when the gate reaches the maximum speed and t_2 is the time when the gate starts to decelerate. The trapezoidal motion profile is designed to ensure that the gate opens and closes smoothly, without any sudden jerks or stops. Listing 1 shows the implementation of the trapezoidal motion profile as MATLAB function.

```
function [t_vec, x_vec] = computeTrapezoidalProfile(pos_current, pos_target, max_vel,
    max_acc)
% COMPUTE_TRAPEZOIDAL_PROFILE Generates time and angle points for a trapezoidal profile
% At first checks if a trapezoidal profile is feasible or is necessary to
% fallback to a triangular one.

delta_pos = pos_target - pos_current;
time_acc_phase = max_vel / max_acc;
delta_pos_acc_phase = 0.5 * max_acc * time_acc_phase^2;

if 2 * delta_pos_acc_phase < abs(delta_pos)
    % Trapezoidal profile
    t1 = time_acc_phase;
    t2 = (abs(delta_pos) - 2 * delta_pos_acc_phase) / max_vel;
    t3 = time_acc_phase;
else
    % Triangular profile
    max_vel = sqrt(abs(delta_pos) * max_acc);
    t1 = max_vel / max_acc;
    t2 = 0;
    t3 = max_vel / max_acc;
end


% Trapezoidal profile template (HP: s0 = 0, v0 = 0)
acc = max_acc * sign(delta_pos);
vel = max_vel * sign(delta_pos);
profile_template = @(t) ...
    (t <= t1) .*                              (1/2 * acc * t.^2) + ...
```

```
    (t > t1 & t <= (t1 + t2)) .*              (1/2 * acc * t1^2 + vel * (t - t1)) + ...
    (t > (t1 + t2) & t <= (t1 + t2 + t3)) .* (1/2 * acc * t1^2 + vel * t2 + vel * (t - (
    t2 + t1)) - 1/2 * acc * (t - (t1 + t2)).^2) + ...
    (t > (t1 + t2 + t3)) .*                   (1/2 * acc * t1^2 + vel * t2 + vel * t3 -
    1/2 * acc * t3.^2);

% Profile generation
t_vec = linspace(0, t1 + t2 + t3, 100);
x_vec = profile_template(t_vec) + pos_current;

end
```

Listing 1: Trapezoidal Motion Profile

PhotoCell Despite being used only during the *WaitingClearance* state of the *Controller* block, the *PhotoCell* block is responsible for detecting the vehicle's presence in front of the gate. This, in fact, is the only way to understand if the vehicle has passed through the gate and the gate can be closed again safely. In order to continuously sample the vehicle's position, the *PhotoCell* block is designed to not be event-driven, but to move continuously between the two states *Free* and *Occupied* based on the vehicle's position. By doing so, an external agent can continuously monitor the vehicle's position by checking the state of the *PhotoCell* block. In order to determines the presence or not of the vehicle, the *PhotoCell* block uses the knowledge of the vehicle's length and position, the latter of which is shared by the *Vehicle FSM*.
Figure 7 shows the *PhotoCell* block of the parking gate FSM.

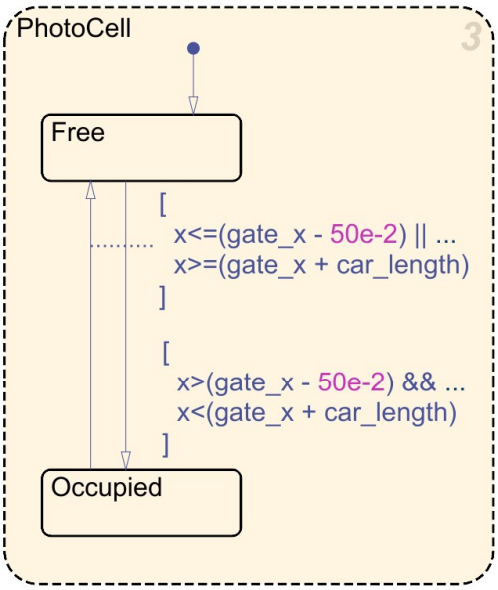

Figure 7: PhotoCell Block

Semaphore Last but not least, the *Semaphore* block is responsible for switching the semaphore state between red, yellow and green based on the events received from the *Controller* block. It has one single state *On* composed of three sub-states representative of the color of the semaphore. Based on the event received, the correct sub-state is chosen and the semaphore state is then shared with the *Vehicle FSM* to inform it about the state of the gate (open, closed, etc.). This is done in order to inform the vehicle about the state of the gate and allow it to proceed or stop based on the semaphore state, which is the only information that the vehicle has about the parking gate system (i.e. the vehicle doesn't know if the gate is open or closed, but only if the semaphore is red or green).
Figure 8 shows the *Semaphore* block of the parking gate FSM.

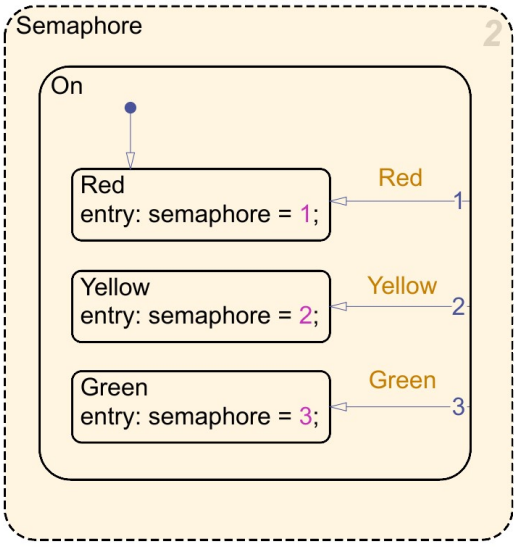

Figure 8: Semaphore Block

3.3 Simulation Results

The simulation has been run for 100s and the vehicle has been reset to a new random position right after the closure of the gate. The following figures show the results of the simulation.

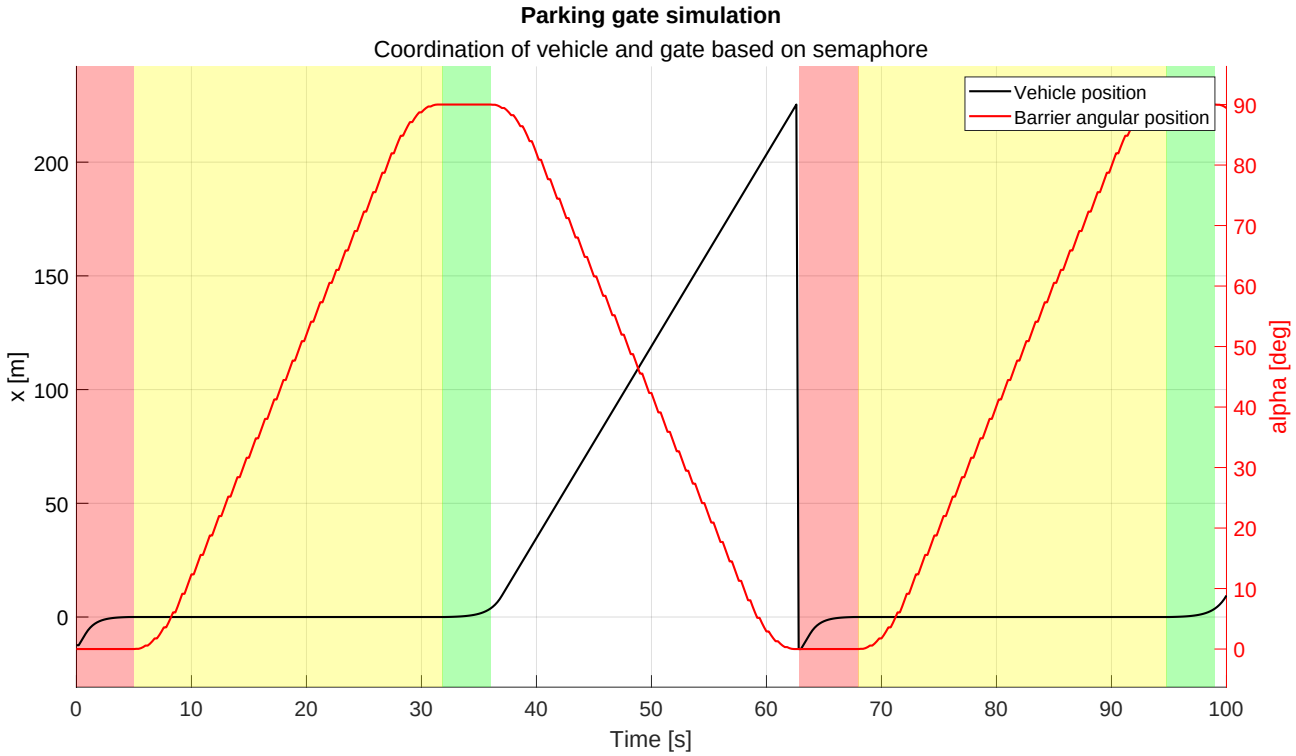

Figure 9: Simulation Results

Figure 9 shows the vehicle position (in black) and the gate angular position (in red) over time. The plot has also been divided into different regions based on the state of the *Semaphore* block (red, yellow and green). Notice that for clarity, the semaphore state has been added to the plot only when actually visible by the vehicle (i.e. when the vehicle is in front of the gate and not yet passed through it).
Instead, a more detailed view of the underlying coordination between the two FSMs is shown in Figure 10, where a visualization of the state transitions and events is shown. The plot report on the vertical axis the time of the simulation and on each vertical dashed line the state of each superstate of the FSMs is shown. One can appreciate how the two FSMs are coordinated and how the events are sent and received between them, along with the transitions between the states.

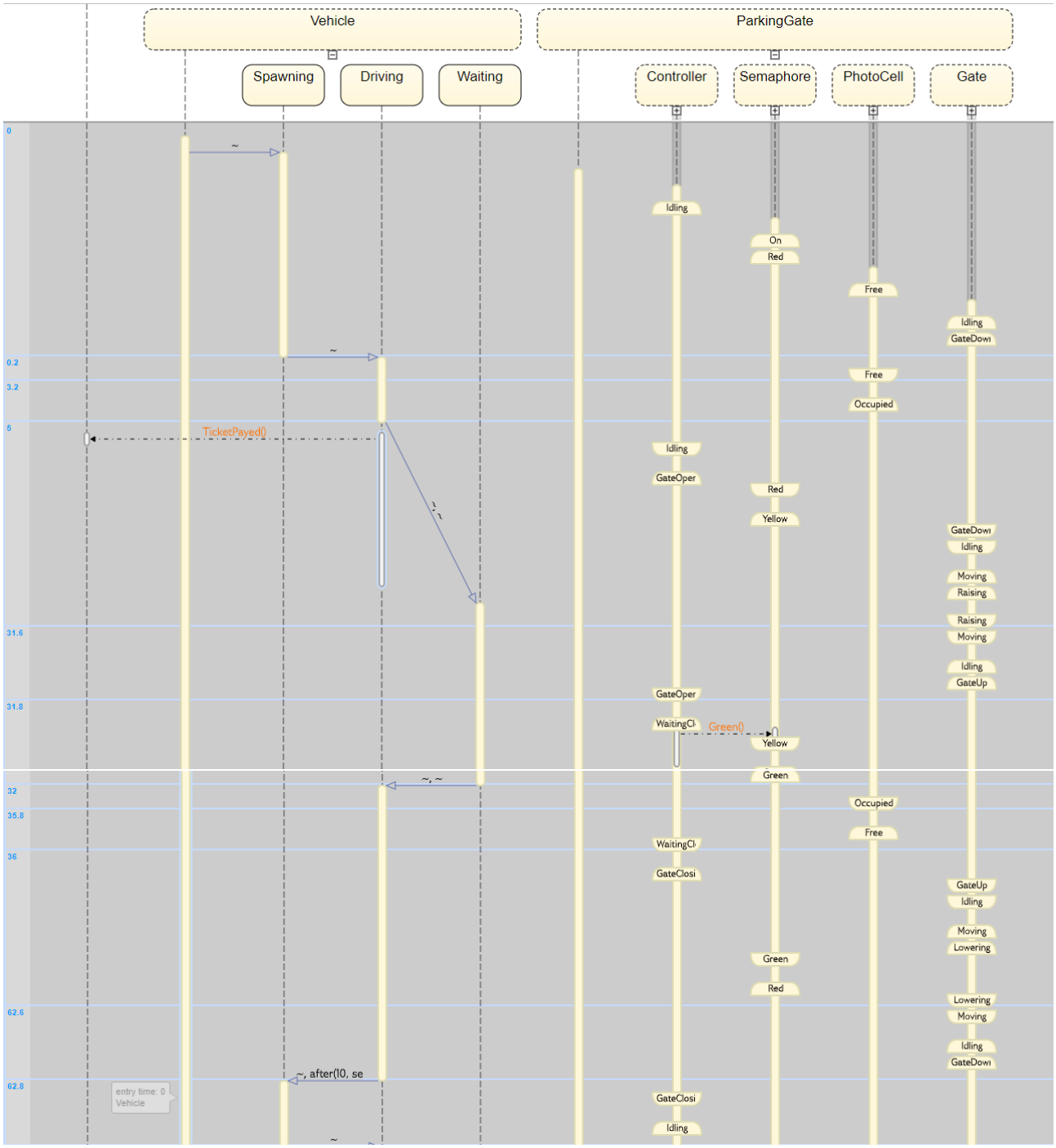

Figure 10: Simulation Results - Detailed View

4 Conclusions

In this short report, we have presented the results of the fifth assignment of the course on Autonomous Vehicles. We have briefly explained what FSMs are and how they can be used to model the behavior of generic systems. We have also presented a possible FMSs implementation about a car parking system, which is able to coordinate the behavior of the car and the parking gate. The results presented have shown the effectiveness of the proposed solution.

The current solution could be improved under many aspects. For example, one could think of implementing features like error handling, or ticket authentication and validation, or even a more complex parking system composed of multiple parking gates and cars that must coordinated over the network.

We believe that FMSs are a power tool that can often replace traditional programming paradigms, allowing to model complex systems in a more intuitive way. Further studies on this topic could be useful to understand the full potential of FMSs and their applications in the field of autonomous vehicles and robotics in general.

References

[1] Wikipedia contributors. Finite-state machine — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Finite-state_machine&oldid=1285525354`, 2025. [Online; accessed 26-April-2025].

A Appendix

The system presented in Section 3.2, leverages one single FSM including under the hood both the vehicle and the parking gate FSMs. However, this solution is not the most elegant one given that the two FSMs are actually two separate entities that communicate with each other via external signals. For this reason, the first solution and implementation of the parking gate system was the one shown in Figure 11.

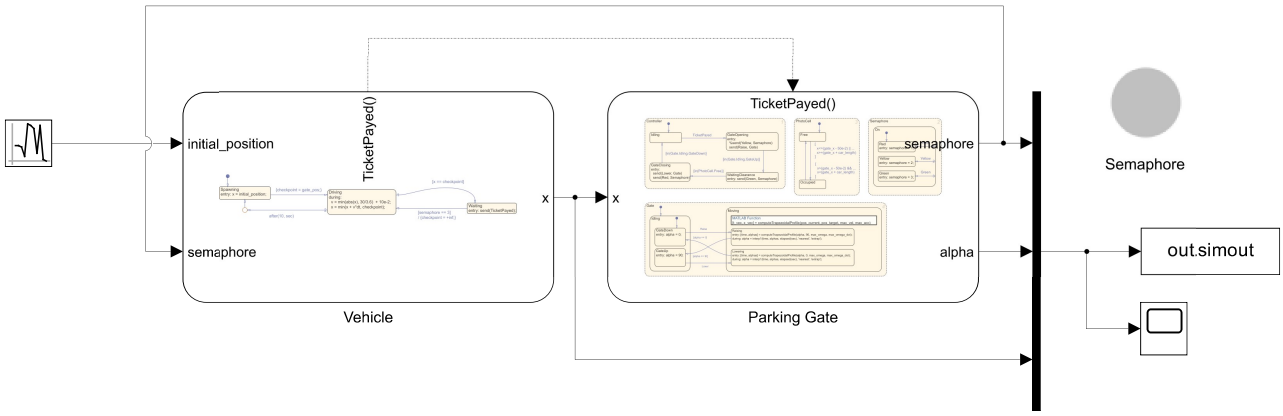

Figure 11: Original Parking Gate System Overview

Notice that the underling components and states are exactly the same as the ones presented in the previous section. Nonetheless, the structure of Figure 11 resolved in unexpected behavior of the system and never generated the expected results.
Even with the help of debugging tools, it was impossible to understand the reason behind the unexpected behavior of the system and the author had to simplify the structure and fallback to the single FSM solution discussed during the document (compromise between elegance and functionality).

Autonomous Vehicles
Implementation of sampled-based motion planning algorithms

Tommaso Bocchietti 10740309

A.Y. 2024/25

POLITECNICO
MILANO 1863

Contents

List of Figures

1 Introduction

Motion planning is a fundamental aspect of autonomous vehicle navigation, enabling them to navigate complex environments while avoiding obstacles and reaching their destinations efficiently.
While in previous work we have focused on the implementation of two grid-based motion planning algorithms, namely Dijkstra and A*, this report aims to implement and analyze sampled-based motion planning algorithms. Specifically, we focus on Rapidly-exploring Random Trees (RRT) and two of its variant, RRT* and a kinematically-learnt version of RRT.
After a brief introduction to the sampled-based motion planning approach, we present the implementation of the algorithms, followed by a discussion of the results obtained by running them on a set of predefined map scenarios. An extensive comparison of the performance between the previously implemented A* algorithm and the RRT* algorithm is also reported, highlighting the strengths and weaknesses of each approach.

Report Structure The report is structured as follows:

- In Section 2, we provide a brief introduction to the sampled-based motion planning algorithms, giving an overview of the single-query and multi-query approaches, and discussing the RRT algorithm and its variants.
- In Section 3, we present the all the three algorithms, RRT, RRT* and RRT-kinematics, explaining the main features of each one and the implementation details.
- In Section 4, we present the results obtained by running the algorithms on a set of predefined map scenarios, and we discuss their performance in terms of computation time, path length and number of nodes sampled.
- In Section 5, we compare the performance of the RRT* algorithm with the A* algorithm, highlighting the strengths and weaknesses of each approach.
- In Section 6, we summarize the main findings of the report and discuss the future work that can be done to improve the algorithms and their performance.

Tools As for the tools used, `MATLAB` is employed as the main platform for implementing the algorithms and performing the analysis of the results.

2 Sampling-based planning

In the field of robotics, a variety of algorithms are available for path planning, each with its own strengths and weaknesses. One of the most ancient and well known class of algorithms is the so called *Grid-based*, which is based on the discretization of the environment into a grid of cells and nodes. As we have already discovered during the previous assignment, this approach is particularly useful for environments with a known structure, such as indoor environments or outdoor environments with well-defined obstacles but is limited in terms of scalability and flexibility.
In contrast, *Sampling-based* algorithms are designed to work in high-dimensional spaces and can handle much more complex environments. For this reason, most of the modern path planning algorithms are based on sampling-based methods, which are able to efficiently explore the configuration space of the robot and find a feasible path from the start to the goal configuration.
Among the many robotics applications that benefit from sampling-based algorithms, we can mention:

- Terrestrial robots: path planning for autonomous vehicles in outdoor environments, such as urban areas or off-road terrains. The ability to handle large environments is crucial for the success of these applications.
- Robotic manipulation: path planning for robotic arms in cluttered environments, where the robot needs to avoid obstacles and reach a specific target position. The ability to handle high-dimensional configuration spaces is essential for these applications.

Sampling-based algorithms are further divided into two main classes: *single-query* and *multi-query* algorithms. Single-query algorithms are designed to find a path from a specific start configuration to a specific goal configuration, while multi-query algorithms are designed to find paths between multiple pairs of start and goal configurations.
In the following sections, we focus exclusively on the single-query algorithms, leaving the multi-query algorithms for future discussions.

2.1 Single-query algorithms

As it was for the grid-based ones, also the single-query sampling-based algorithms follow a structure that is in common to all of them. The main idea here is to start from a given configuration and then randomly sample points in the configuration space, connecting them to form a tree if the condition of collision-free is satisfied. The tree is then incrementally expanded until a path from the start configuration to the goal configuration is found.
The fundamental difference between grid-based and sampling-based algorithms is that the former rely on a discretized representation of the environment, while the latter rely on a continuous representation of the configuration space. This allows sampling-based algorithms to handle complex environments with arbitrary shapes and obstacles, as they do not rely on a fixed grid structure.

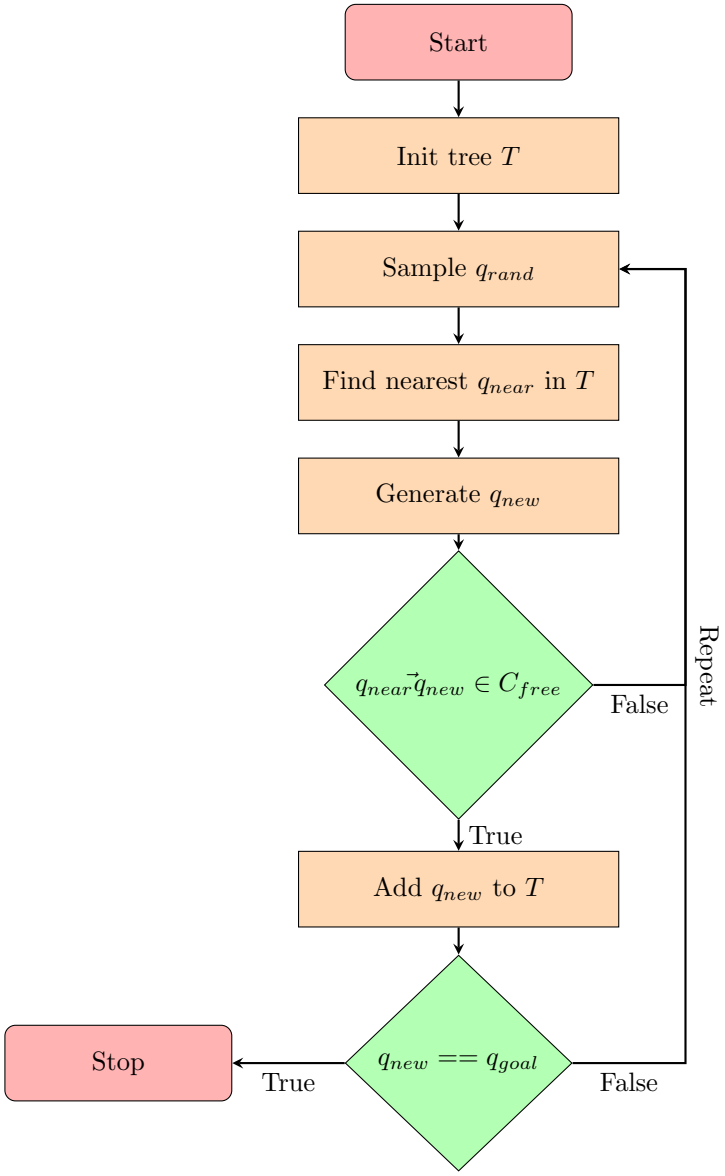

Figure 1: Generic sampling-based algorithm flowchart

The flowchart in Figure 1 summarizes the main steps of a generic sampling-based algorithm. One can notice that tree T is built incrementally at each iteration, starting from the initial configuration and adding new sampled points to the tree if they respect the collision-free condition. As already said, this is in fact the main difference between the sampling-based algorithms and the grid-based algorithms, which rely on a complete discretization of the environment made in advance.
However, the sampling-based algorithms suffers from a major drawback, which is the lack of completeness and optimality. In particular, the algorithms are not guaranteed to find a solution, even if one exists, and the solution found may not be optimal. As we discuss in the following sections, improvements have been made to the core structure to address these issues, but they still remain a challenge in the field of robotics and motion planning.

There exist many variations built on top of the core algorithm presented in Figure 1. The most common ones are:

- **Rapidly-exploring Random Tree (RRT)**: a single-query algorithm that just implements the core algorithm presented in Figure 1. It's the simplest, but also the most suboptimal algorithm among the ones presented in this project.
- **RRT***: an extension of the RRT algorithm that improves the optimality of the solution by rewiring the tree to find shorter paths. It is more computationally expensive than the RRT algorithm, but it is able to refine the solutions in terms of path length.
- **RRT-connect**: a variant of the RRT algorithm that builds two trees, one from the start configuration and one from the goal configuration, and connects them to find a path. It's usually faster than the RRT algorithm to find a solution given that the search space is explored from both ends and also benefits from some form of optimality.
- **RRT-kinematic**: a variant of the RRT algorithm that takes into account the kinematic constraints of the robot/vehicle during the planning process. This is particularly useful for vehicles with non-holonomic constraints, such as terrestrial vehicles or mobile robots. The algorithm generates a tree of feasible trajectories that respect the kinematic constraints of the robot.

The following sections focus on the in-depth analysis and implementation of the RRT, RRT* and RRT-kinematic algorithms. Notice that despite being a popular and widely used algorithm, the RRT-connect algorithm is not going to be implemented in this project due to time constraints.

3 Algorithm implementation

In this section, we present the implementation of the three algorithms we are going to analyze in this report. The algorithms are implemented in `MATLAB` and are designed to handle multiple dimensions problems, meaning that they can be used for both N-D environments. In the following only the core of each algorithm is presented, leaving the code of all the auxiliaries classes such as `Graph`, `Edge`, or `Node` among the files in the repository.

3.1 RRT algorithm

The most basic version of the single-query sampling-based algorithm is the Rapidly-exploring Random Tree (RRT) algorithm. The RRT algorithm is nothing more than the direct implementation of the core algorithm presented in Figure 1.
Listing 1 shows a possible implementation of the RRT algorithm.

```
function [exist, distances, parents] = rrt(G, map, node_initial, node_final)
% RRT Builds a tree using the Rapidly-exploring Random Tree (RRT) algorithm.
%
%   [EXIST, DISTANCES, PARENTS] = RRT(G, MAP, NODE_INITIAL, NODE_FINAL)
%   attempts to grow a tree from NODE_INITIAL to NODE_FINAL using random
%   sampling and a step-limited steering function.
%
%   Inputs:
%       G            - Graph object representing the RRT.
%       MAP          - Binary occupancy grid (1 = free, 0 = obstacle).
%       NODE_INITIAL - Node object with initial state.
%       NODE_FINAL   - Node object with final/goal state.
%
%   Outputs:
%       EXIST     - Boolean flag indicating whether a path to the goal was found.
%       DISTANCES - Vector of cumulative distances from the start to each node.
%       PARENTS   - Vector of parent node IDs for reconstructing the tree.
%
%   The tree is expanded by selecting random points and connecting the closest
%   node in the tree toward them, checking each connection for validity.

q_new = node_initial.state;
distances = zeros(1, 1);
```

```
parents = NaN(1, 1);
iter = 0;

while (Node.euclideanDistance(q_new, node_final.state) > G.step_length && iter < G.
    max_iter)

    % Sampling stage
    q_rand = rand(1, ndims(map)) .* size(map);

    % Finding the nearest node
    IDs_near = G.getNearbyIDs(q_rand);
    ID_near = IDs_near(1);
    q_near = G.getNodeByID(ID_near).state;

    % Steering stage
    q_new = G.steerState(q_near, q_rand, G.step_length);

    % Collision checking and tree update
    if (G.isValidConnection(q_near, q_new, map))

        ID_new = G.addNode(q_new);
        dist = Node.euclideanDistance(q_near, q_new);
        G.addEdge(ID_near, ID_new, dist);

        parents(ID_new, 1) = ID_near;
        distances(ID_new, 1) = distances(ID_near) + dist;

    end

    iter = iter + 1;

end

exist = ~(iter == G.max_iter);

% Connection with final node
if (exist)
    ID_final = G.addNode(node_final.state);
    IDs_near = G.getNearbyIDs(node_final.state);
    ID_near = IDs_near(1);
    q_near = G.getNodeByID(ID_near).state;
    dist = Node.euclideanDistance(q_near, node_final.state);
    G.addEdge(ID_near, ID_final, dist);
    parents(ID_final, 1) = ID_near;
    distances(ID_final, 1) = distances(ID_near) + dist;
end

end
```

Listing 1: Node class used to represent a node in the graph.

The algorithm takes as input the graph G, the map M, the start node s, and the goal node g. It starts by initializing the tree T with the start node and enters a loop where at first a random point/configuration is sampled from the workspace. Then, it finds the nearest node already in the tree and generates a new node by moving in the direction of the sampled point. If the new node is in collision with the obstacles, it's discarded and the algorithm skips to the next iteration. If the new node is valid, it's added to the tree. A final step is to check if the new node is close enough to the goal node, in which case the algorithm terminates and returns the path from the start node to the goal node, otherwise the algorithm continues to sample new points until a maximum number of iterations is reached, or the goal node is reached.
The RRT algorithm is simple and efficient, but lacks optimality in the generated path which is normally jagged and not smooth. The algorithm is also not complete, meaning that it may not find a solution even if one exists in the configuration space due to the maximum number of iterations limit. However, it has been proved that the

RRT algorithm is probabilistically complete, meaning that as the number of iterations increases, the probability of finding a solution approaches 1 (at the cost of longer computation time, of course).

3.2 RRT* algorithm

RRT* is an extension of the RRT algorithm that improves the optimality of the solution by rewiring the tree to find shorter paths. The main idea behind the RRT* algorithm is to keep track of the cost of reaching each node in the tree and to rewire the tree whenever a new node is added. This is done by checking if the new node can be used as a cheaper parent connection for any of the existing nodes in the tree.
Listing 2 shows a possible implementation of the RRT* algorithm.

```
function [exist, distances, parents] = rrtStar(G, map, node_initial, node_final)
% RRTSTAR Builds a tree using the RRT* (optimal Rapidly-exploring Random Tree) algorithm.
%
%   [EXIST, DISTANCES, PARENTS] = RRTSTAR(G, MAP, NODE_INITIAL, NODE_FINAL)
%   attempts to find an optimal path from NODE_INITIAL to NODE_FINAL
%   in the graph G by incrementally building a tree of collision-free paths
%   using random sampling, rewiring, and cost minimization.
%
%   Inputs:
%       G            - Graph object representing the RRT.
%       MAP          - Binary occupancy grid (1 = free, 0 = obstacle).
%       NODE_INITIAL - Node object with initial state.
%       NODE_FINAL   - Node object with final/goal state.
%
%   Outputs:
%       EXIST     - Boolean flag indicating whether a path to the goal was found.
%       DISTANCES - Vector of cumulative distances from the start to each node.
%       PARENTS   - Vector of parent node IDs for reconstructing the tree.
%
%   The algorithm improves upon RRT by locally rewiring nodes to ensure that
%   each node is connected through the lowest-cost path, producing asymptotically
%   optimal solutions.

q_new = node_initial.state;
distances = zeros(1, 1);
parents = NaN(1, 1);
iter = 0;

while (Node.euclideanDistance(q_new, node_final.state) > G.step_length && iter < G.max_iter)

    % Sampling stage
    q_rand = rand(1, ndims(map)) .* size(map);

    % Finding the nearest node
    IDs_near = G.getNearbyIDs(q_rand);
    ID_near = IDs_near(1);
    q_near = G.getNodeByID(ID_near).state;

    % Steering stage
    q_new = G.steerState(q_near, q_rand, G.step_length);

    % Collision checking and tree update
    if (G.isValidConnection(q_near, q_new, map))

        % Node creation
        ID_new = G.addNode(q_new);

        % Neighborhood selection
```

```
        IDs_near = G.getNearbyIDs(q_new, G.rewiring_radius)';
        % IDs_near = G.getNearbyIDs(q_new, G.rewiring_radius*(log(numel(G.nodes)) / numel(G.nodes))^(1/(ndims(map)+1)) )';

        % Parent choice & edge creation
        distances(ID_new, 1) = +inf;
        for ID_near = IDs_near
            q_near = G.getNodeByID(ID_near).state;
            dist = distances(ID_near) + Node.euclideanDistance(q_near, q_new);
            if (dist < distances(ID_new, 1) && G.isValidConnection(q_near, q_new, map))
                distances(ID_new, 1) = dist;
                parents(ID_new, 1) = ID_near;
            end
        end
        node_parent = G.getNodeByID(parents(ID_new));
        G.addEdge(node_parent.ID, ID_new, Node.euclideanDistance(node_parent.state, q_new));

        % Rewiring stage
        for ID_near = IDs_near
            q_near = G.getNodeByID(ID_near).state;
            dist = distances(ID_new) + Node.euclideanDistance(q_near, q_new);
            if (dist < distances(ID_near) && G.isValidConnection(q_new, q_near, map))
                edge = G.getEdgeByConnection(parents(ID_near), ID_near);
                G.edges(edge.ID).ID_A = ID_new;
                G.edges(edge.ID).ID_B = ID_near;
                G.edges(edge.ID).weight = Node.euclideanDistance(q_new, q_near);
                distances(ID_near) = dist;
                parents(ID_near) = ID_new;
            end
        end

    end

    iter = iter + 1;

end

exist = ~(iter == G.max_iter);

% Connection with final node
if (exist)
    ID_final = G.addNode(node_final.state);
    IDs_near = G.getNearbyIDs(node_final.state);
    ID_near = IDs_near(1);
    q_near = G.getNodeByID(ID_near).state;
    dist = Node.euclideanDistance(q_near, node_final.state);
    G.addEdge(ID_near, ID_final, dist);
    parents(ID_final, 1) = ID_near;
    distances(ID_final, 1) = distances(ID_near) + dist;
end

end
```

Listing 2: Code for the implementation of RRT* algorithm.

Again, the core structure of the algorithm is similar to the RRT algorithm, but with a few extra steps. The algorithm starts by initializing the tree T with the start node and enters a loop where it samples a random point in the configuration space. Then, it finds the nearest node in the tree and generates a new node by moving towards the sampled point. If the new node is in collision with the obstacles, it is discarded and the algorithm continues to sample a new point. If the new node is valid, the algorithm at first connect it to the nearest node in the tree, and then checks if any of its neighbors can also be connected to it with a lower cost-to-go. This

analysis, called rewiring, help to construct a more optimal tree with smoother connections and consequently shorter paths. Figure 2 shows an example of the rewiring process.

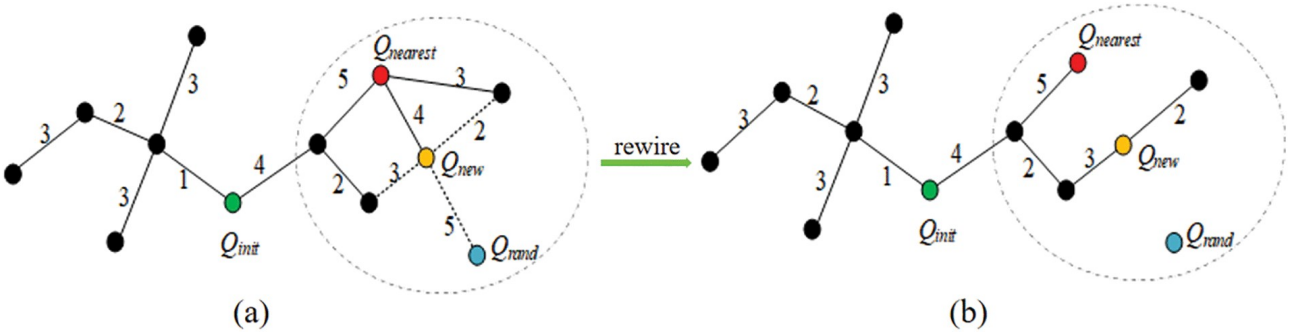

Figure 2: Example of the rewiring process in RRT* algorithm. Credit to [1].

As it is going to be shown in the next section, the RRT* algorithm is more efficient than the RRT algorithm in terms of path length, at the cost negligible increase in computation time.

Optimality convergence of RRT* As a final note, we recall that thanks to [2], it has been proved that the RRT* algorithm converges to the optimal solution as the number of sampled points goes to infinity and the rewiring radius is dynamically adjusted as:

$$r_n = \gamma \left(\frac{\log n}{n} \right)^{\frac{1}{d+1}} \tag{1}$$

Where n is the number of sampled points, d is the dimension of the configuration space, and γ is a constant that depends on d.
Line 50 of the code in Listing 2 shows a possible implementation of this dynamic rewiring radius.

3.3 RRT-kinematic algorithm

Finally, we move to the last algorithm we are going to implement, which is the RRT-kinematic algorithm. The RRT-kinematic algorithm is a variant of the RRT algorithm that takes into account the kinematic constraints of the robot during the planning process. This is particularly useful for vehicles with non-holonomic constraints, such as terrestrial vehicles or mobile robots.
Listing 3 shows a possible implementation of the RRT-kinematic algorithm.

```
1  function [exist, distances, parents] = rrtKinematic(G, map, node_initial, node_final,
       vehicle_model)
2  % RRTKINEMATIC Builds a tree using RRT with vehicle kinematic constraints.
3  %
4  %   [EXIST, DISTANCES, PARENTS] = RRTKINEMATIC(G, MAP, NODE_INITIAL, NODE_FINAL,
       VEHICLE_MODEL)
5  %   attempts to find a path from NODE_INITIAL to NODE_FINAL using a kinematically
6  %   feasible motion model for non-holonomic vehicles (e.g., cars).
7  %
8  %   Inputs:
9  %       G             - Graph object to store the RRT tree.
10 %       MAP           - Binary occupancy map (1 = free, 0 = obstacle).
11 %       NODE_INITIAL  - Starting node with state [x, y, theta].
12 %       NODE_FINAL    - Goal node with target state [x, y, theta].
13 %       VEHICLE_MODEL - Function handle simulating vehicle motion over time.
14 %
15 %   Outputs:
16 %       EXIST     - Boolean indicating whether a valid path was found.
17 %       DISTANCES - Cumulative cost from the start node to each tree node.
18 %       PARENTS   - Parent indices used to reconstruct the tree structure.
19 %
20 %   This version of RRT uses motion primitives derived from a forward vehicle model
21 %   to ensure that the generated paths respect the system kinematic constraints.
```

```
22
23 v_options = linspace(-1, 1, 2);
24 v_options = 1;
25 delta_options = linspace(-pi/6, pi/6, 3);
26
27 q_new = node_initial.state;
28 distances = zeros(1, 1);
29 parents = NaN(1, 1);
30 iter = 0;
31
32 while (Node.euclideanDistance(q_new(1:2), node_final.state(1:2)) > G.step_length && iter
       < G.max_iter)
33
34     % Sampling stage
35     q_rand = [max(rand(1, 2) .* size(map), 1), rand() * 2 * pi];
36
37     % Finding the nearest node
38     IDs_near = G.getNearbyIDs(q_rand);
39     ID_near = IDs_near(1);
40     q_near = G.getNodeByID(ID_near).state;
41
42     % Kinematics constraints
43     best_state = [];
44     min_dist = Inf;
45     for v = v_options
46         for delta = delta_options
47
48             % Forward integration of vehicle motion
49             q_new = vehicle_model(q_near, [v, delta], G.step_length / abs(v));
50
51             % Collision checking and choice of nearest state to sampled one
52             if (G.isValidConnection(q_near, q_new, map))
53                 dist = Node.euclideanDistance(q_new(1:2), q_rand(1:2));
54                 if (dist < min_dist)
55                     best_state = q_new;
56                     min_dist = dist;
57                 end
58             end
59
60         end
61     end
62
63     if isempty(best_state)
64         continue;
65     end
66
67     % Tree update
68     q_new = best_state;
69     ID_new = G.addNode(q_new);
70     dist = Node.euclideanDistance(q_near(1:2), q_new(1:2));
71     G.addEdge(ID_near, ID_new, dist);
72
73     parents(ID_new, 1) = ID_near;
74     distances(ID_new, 1) = distances(ID_near) + dist;
75
76     iter = iter + 1;
77
78 end
79
80 exist = ~(iter == G.max_iter);
81
```

```
% Connection with final node
if (exist)
    ID_final = G.addNode(node_final.state);
    IDs_near = G.getNearbyIDs(node_final.state);
    ID_near = IDs_near(1);
    q_near = G.getNodeByID(ID_near).state;
    dist = Node.euclideanDistance(q_near(1:2), node_final.state(1:2));
    G.addEdge(ID_near, ID_final, dist);
    parents(ID_final, 1) = ID_near;
    distances(ID_final, 1) = distances(ID_near) + dist;
end

end
```

Listing 3: Code for the implementation of RRT-kinematic algorithm.

Again, we recognize the same core structure of the RRT algorithm, but with a few extra steps. In particular, the choice of the new node is not done by simply moving towards the sampled point, but rather by generating a trajectory from the nearest node in the tree to the sampled point leveraging the kinematic model of the robot. In the specific case of the implemented algorithm, we consider dealing with a *Bicycle* model, which is a common model used to represent the motion of a car-like vehicle. Starting from the nearest node in the tree, a random control input is sampled and a trajectory is generated by integrating the kinematic model of the robot over a fixed time horizon. Collision checking is then performed and among all the tested trajectories, the one that is valid and has the lowest cost is selected as the new node to be added to the tree. By doing so, we are sure that the generated path is feasible by the robot, respecting its kinematic constraints.

Bicycle kinematic model Even if the kinematic model of the vehicle is not the main focus of this report, we briefly present it here for completeness. The kinematic model of the vehicle is described by the following equations:

$$\begin{aligned} \dot{x} &= v \cdot \cos(\theta) \\ \dot{y} &= v \cdot \sin(\theta) \\ \dot{\theta} &= \frac{v}{L} \cdot \tan(\delta) \end{aligned} \tag{2}$$

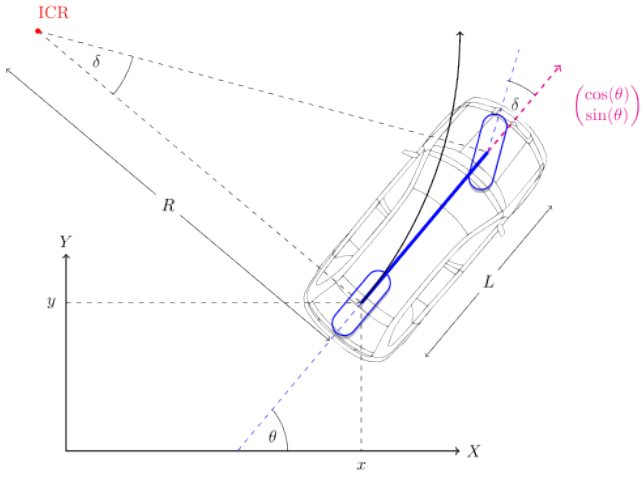

Figure 3: Bicycle kinematic model. Image credit to Thomas Fermi.

Where x and y are the position of the vehicle in the workspace, θ is the heading angle of the vehicle, v is the linear velocity, L is the wheelbase of the vehicle (distance between the front and rear axles), and δ is the steering angle.
One can clearly see that the kinematic model is composed by a set of differential equations. In order to simulate in time based on the control inputs and current state of the vehicle, we need to integrate these equations. Different integration methods can be used, such as Euler integration or Runge-Kutta integration. However, given the simplicity and expected slow dynamics of the model, we can use a simple Euler integration with a fixed time step Δt.
Listing 4 shows a possible implementation of the kinematic model of the vehicle.

```
function q_new = modelBicycle(q, u, T)
% MODELBICYCLE Simulates bicycle model kinematics over time T.
%
%   Q_NEW = MODELBICYCLE(Q, U, T) updates the state of a vehicle using the
```

```
%   kinematic bicycle model given the current state Q, control input U, and
%   time duration T.
%
%   Inputs:
%       Q - Current state vector [x, y, theta], where:
%           x, y   - Position of the vehicle in 2D space
%           theta  - Heading angle (radians)
%
%       U - Control input vector [speed, steering_angle], where:
%           speed           - Forward velocity
%           steering_angle  - Steering angle (radians)
%
%       T - Duration over which to apply the control input (seconds)
%
%   Output:
%       Q_NEW - New state vector [x, y, theta] after time T.

% States
x = q(1);
y = q(2);
theta = q(3);

% Inputs
speed = u(1);
steering_angle = u(2);

% Parameters
wheelbase = 2.0;

time = linspace(0, T, 100);
dt = diff(time(1:2));
for t = time
    x = x + speed * cos(theta) * dt;
    y = y + speed * sin(theta) * dt;
    theta = theta + (speed / wheelbase) * tan(steering_angle) * dt;
end

q_new = [x y theta];

end
```

Listing 4: Code for the kinematic model of the vehicle.

4 Algorithm Testing

We can now proceed to test the algorithms we have implemented in the previous section. In the following, the maps scenarios adopted are presented first, and then the results of the tests are shown.

4.1 Map scenarios

Figures 4 show the four image maps used to test the algorithm. Even if the images here below are of size 300x300, we preferred to scale down to 30x30 in order to have a direct comparison between the results obtained previously with the grid-based algorithms. Notice also that while the white areas are free space, the one marked in black or gray are to be considered as obstacles. Finally, the green dot is the starting point of the vehicle, while the red one is the goal.

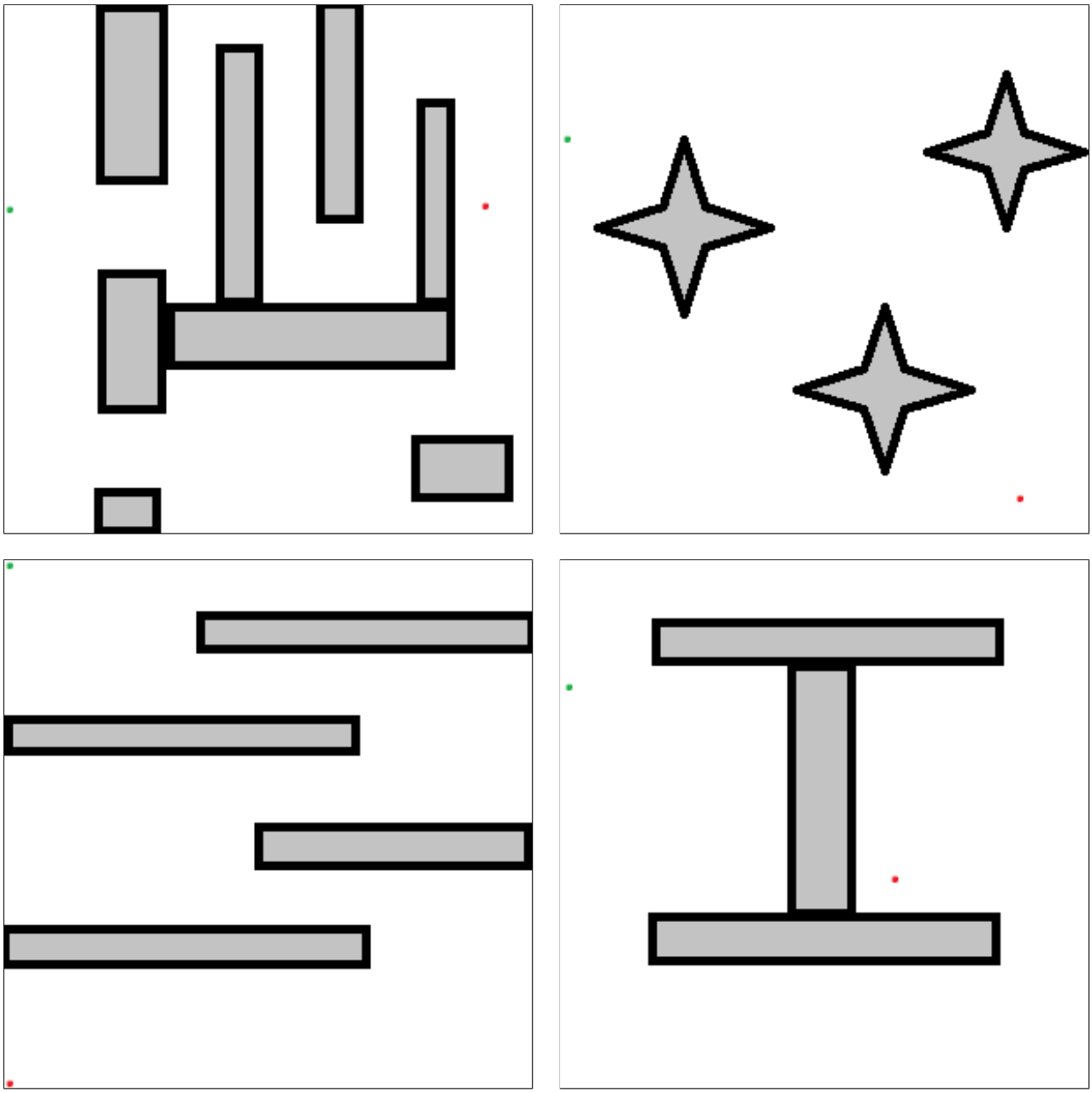

Figure 4: Test maps used for the algorithm testing.

4.2 Results analysis

In the following, results obtained running both RRT, RRT* and RRT kinematic algorithms on each of the scenario are presented. For comparison purposes, the results obtained running the A* algorithm on the same maps are also reported, even if an in-depth analysis of the differences and performances of the two families of algorithms is left to the next section (Section 5).
The plotting of both the (geometrical) connected graph and the best path found is also shown. Tables presenting the main metrics of the tests are also shown. The metrics are the following:

- **Elapsed time**: here only the time needed to run the searching algorithm is considered (i.e. we are including the tree generation but excluding the plotting of the results);
- **Tree dimension**: number of nodes in the tree generated by the algorithm. In case of the A* algorithm, this is the number of nodes in the graph generated during the discretization/grid generation phase;
- **Path length**: length of the path found by the algorithm. This also correspond to the path cost, given that the cost of each edge is equal to its geometrical length.

Notice that the due to the randomness nature of sampling-based algorithms, the reported results must be

interpreted as an average of 5 runs of each algorithm on each map. Moreover, the tuning parameters of the algorithms such as step size and rewiring radius have been set on all the algorithms variants to be:

Parameter	Value
Step size	1.0
Rewiring radius*	1.5

Table 1: Tuning parameters of the algorithms. The rewiring radius is only used in the RRT* algorithm.

Units are considered to be in pixels dimensions, where we recall the size of the map to be 30x30.

4.2.1 Map 1

Results obtained running the three algorithms on the first map are shown in Figures 5 and Table 2. The path found by the algorithm is shown in red, while the tree/graph is shown in black.
Notice that the fourth image (bottom right) is actually the results obtained running the A* algorithm, and it is shown here just for comparison purposes.

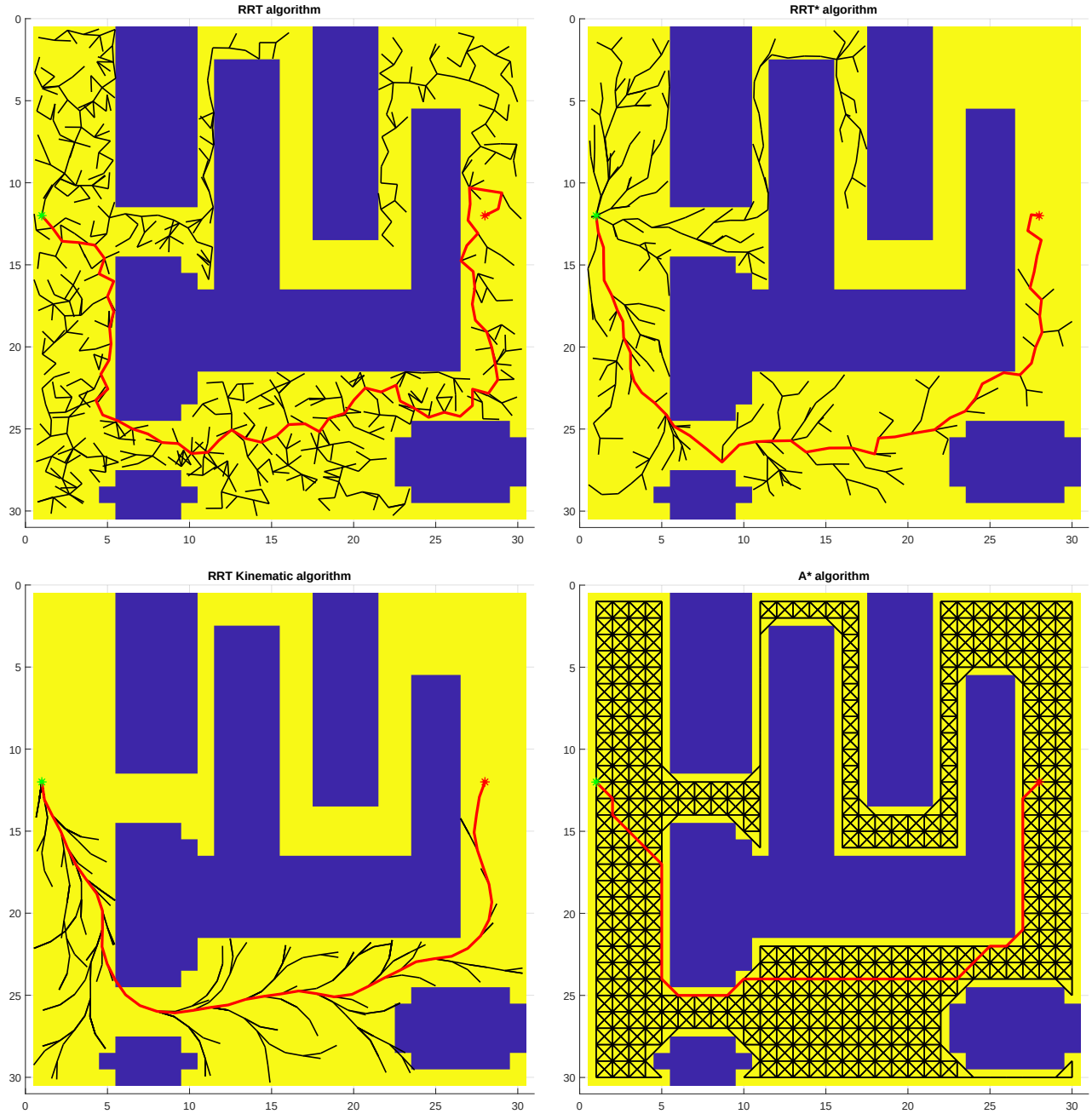

Figure 5: Tree generated and path found by the RRT, RRT* and RRT Kinematic algorithms on map 1.

Algorithm	Elapsed time (ms)	Tree dimension	Path length (m)
RRT	145	338	57
RRT*	202	271	49
RRT Kinematic	129	457	49
A*	1987	3332	47

Table 2: Results of the algorithms on map 1.

The results clearly show the core problematics of the sampling-based algorithms: the path found is not optimal. Comparing the results of the RRT and RRT* algorithms, we can see that the latter is able to find a shorter path, but it's still not able to reach the lower limit which is the one found by the A* algorithm. On the other hand, sampling-based algorithms demonstrate to be in general 10x faster than grid-based algorithms, as we can see from the elapsed time. Also, under the space complexity point of view, the search space is significantly smaller for the sampling-based algorithms. This suggests that the sampling-based algorithms are more efficient in terms of space complexity, as they do not need to explore the whole workspace to find a feasible path.

4.2.2 Map 2

Results obtained running the three algorithms on the second map are shown in Figures 6 and Table 3. The path found by the algorithm is shown in red, while the tree/graph is shown in black. Again, we have included the results of the A* algorithm for comparison purposes.

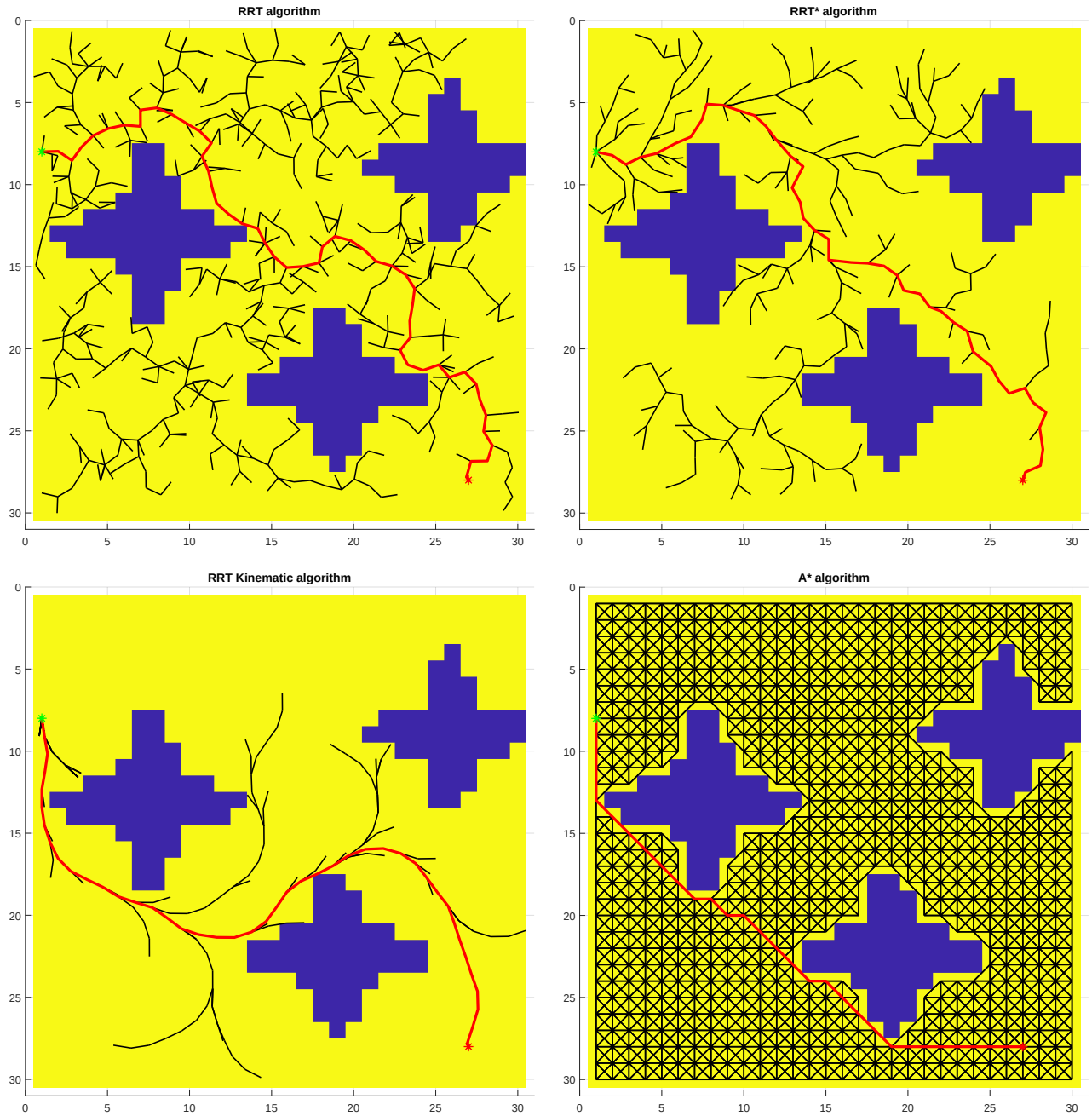

Figure 6: Tree generated and path found by the RRT, RRT* and RRT Kinematic algorithms on map 2.

Algorithm	Elapsed time (ms)	Tree dimension	Path length (m)
RRT	199	581	49
RRT*	360	383	40
RRT Kinematic	282	615	42
A*	2025	5398	37

Table 3: Results of the algorithms on map 2.

Once again, we observe that the path found by the sampling-based algorithms is suboptimal. Similar to before, the RRT* algorithm is the one able to get the closest to the optimal path found by the A* algorithm but still, it is not able to reach it. As for the elapsed time, we still recognize a factor of around 8 in speedup of the sampling-based algorithms with respect to the grid-based ones. We also recall that this particular map and start/goal configuration is the one where the A* algorithm was able to find the optimal path in the shortest time, thanks to the heuristic adopted and the lower number of possible dead-ends-street during the search. We believe that the speed-up factor might be even higher in more complex maps, where the A* algorithm would have to explore a larger number of nodes before finding the optimal path.

4.2.3 Map 3

Results obtained running the three algorithms on the third map are shown in Figures 7 and Table 4. The path found by the algorithm is shown in red, while the tree/graph is shown in black. Again, we have included the results of the A* algorithm for comparison purposes.

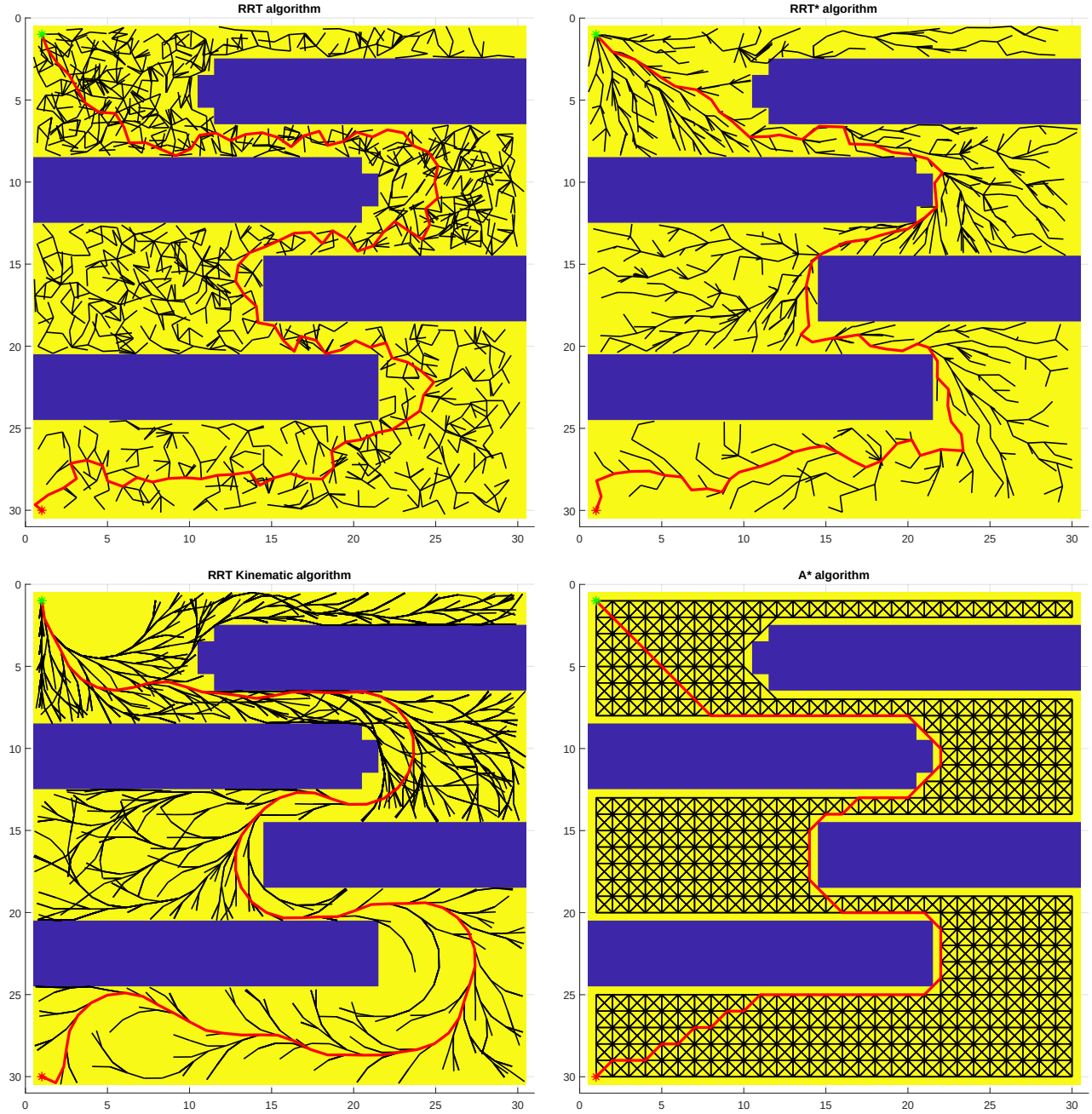

Figure 7: Tree generated and path found by the RRT, RRT* and RRT Kinematic algorithms on map 3.

Algorithm	Elapsed time (ms)	Tree dimension	Path length (m)
RRT	552	1085	97
RRT*	1208	932	74
RRT Kinematic	68601	10338	93
A*	2216	3936	74

Table 4: Results of the algorithms on map 3.

From the test performed on this map, we gain further information about the performance of the algorithms in cluttered environments. In this case, the difference in computation time between the sampling-based algorithms and the A* algorithm is not as significant as in the previous cases. The sampling approach is in fact heavily

slowed down by the four obstacles, that impose the need of numerous samples to progress in the successive sectors of the map up to the goal. This is also reflected in the distribution of the nodes in the tree, which is much denser in the upper sectors of the map (near the starting point) than in the lower ones (near the goal). RRT-kinematic clearly shows this behavior, highlighting the trapping of the algorithm in front of each obstacle. A* algorithm, on the other hand, still suffers from local dead-ends searching, but it's much quicker in finding a way to get around the obstacles.
For completeness, we also report that in the case of the RRT-kinematic algorithm, high variance in the results was observed. In particular, among the runs performed, we have one case where the algorithm was able to find a path in 8.6 seconds, while in the worse case it took 115.7 seconds.

4.2.4 Map 4

Results obtained running the three algorithms on the fourth map are shown in Figures 8 and Table 5. The path found by the algorithm is shown in red, while the tree/graph is shown in black. Again, we have included the results of the A* algorithm for comparison purposes.

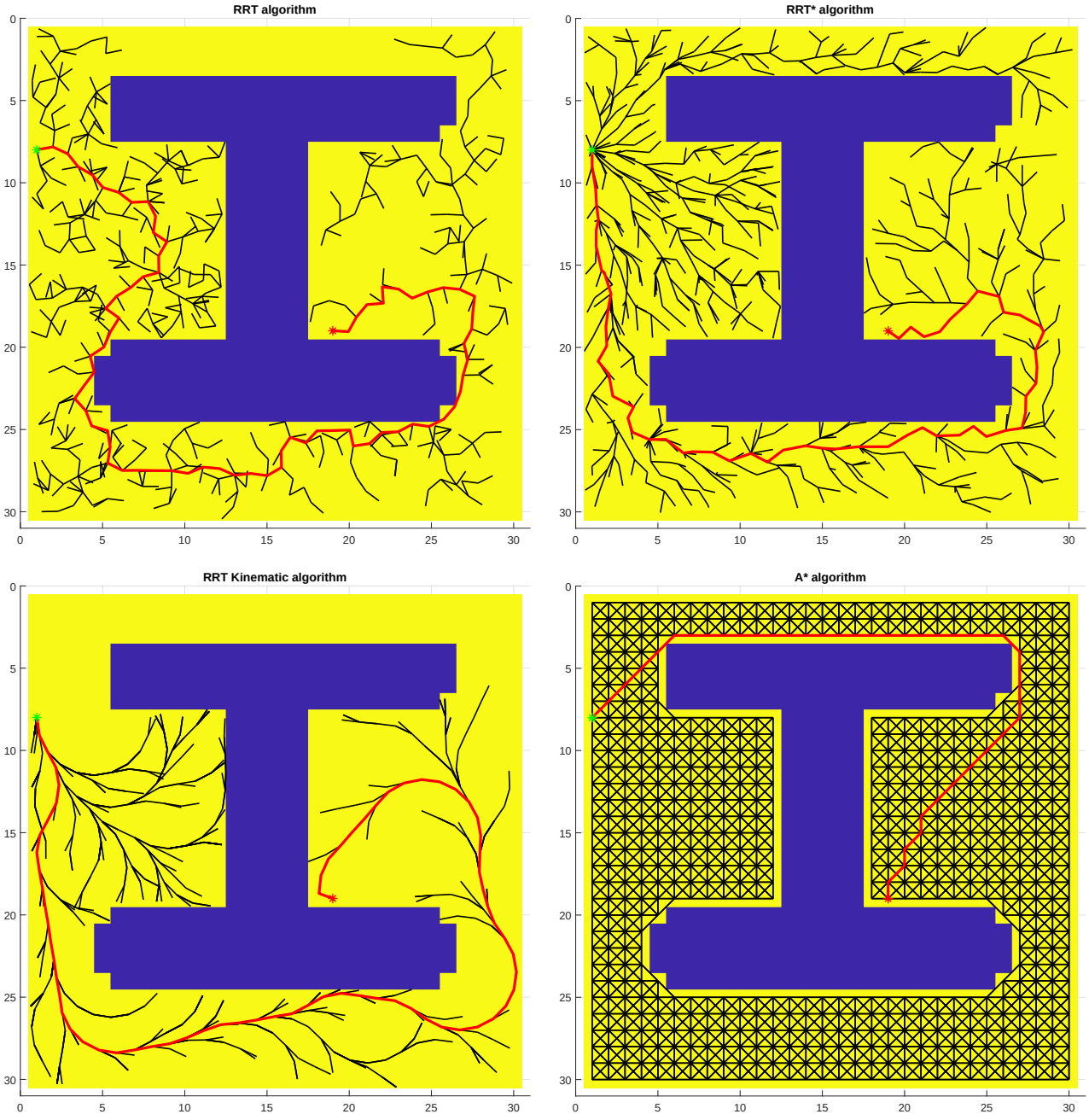

Figure 8: Tree generated and path found by the RRT, RRT* and RRT Kinematic algorithms on map 4.

Algorithm	Elapsed time (ms)	Tree dimension	Path length (m)
RRT	164	604	67
RRT*	485	618	45
RRT Kinematic	782	1649	64
A*	2607	4520	47

Table 5: Results of the algorithms on map 4.

The results obtained on this map are similar to the ones obtained on the previous ones. It's interesting to notice how the RRT was not able to explore the upper region of the map, in which instead the A* finds its optimal path. Once again, the concept of obstacles blocking the path of the sampling-based algorithms is highlighted similarly to the previous case.

5 Grid vs Sample Based Planning

So far we have limited our analysis and comparisons between the grid-based and the sample-based algorithms to a fixed set of predefined scenarios. In this section, we further investigate the performance and capabilities of the two classes of algorithms by running specific tests that are designed to highlight their strengths and weaknesses. Thanks to the flexibility provided by the rewiring mechanism of RRT*, in this section we are going to compare the performance of the A* against the RRT* algorithm.

5.1 Path Asymptotic Optimality

As already largely discussed in the previous sections, the main difference between grid-based and sample-based algorithms is that the former are guaranteed to find the optimal path, while the latter are not. Formally, this statement has also been expressed in the literature by [2], which has proven that RRT can produce arbitrarily-bad paths with non-negligible probability.
We can also observe this behavior in our implementation of the RRT* algorithm. Figure 9 shows the path generated by RRT* on the left and the path generated by A* on the right.

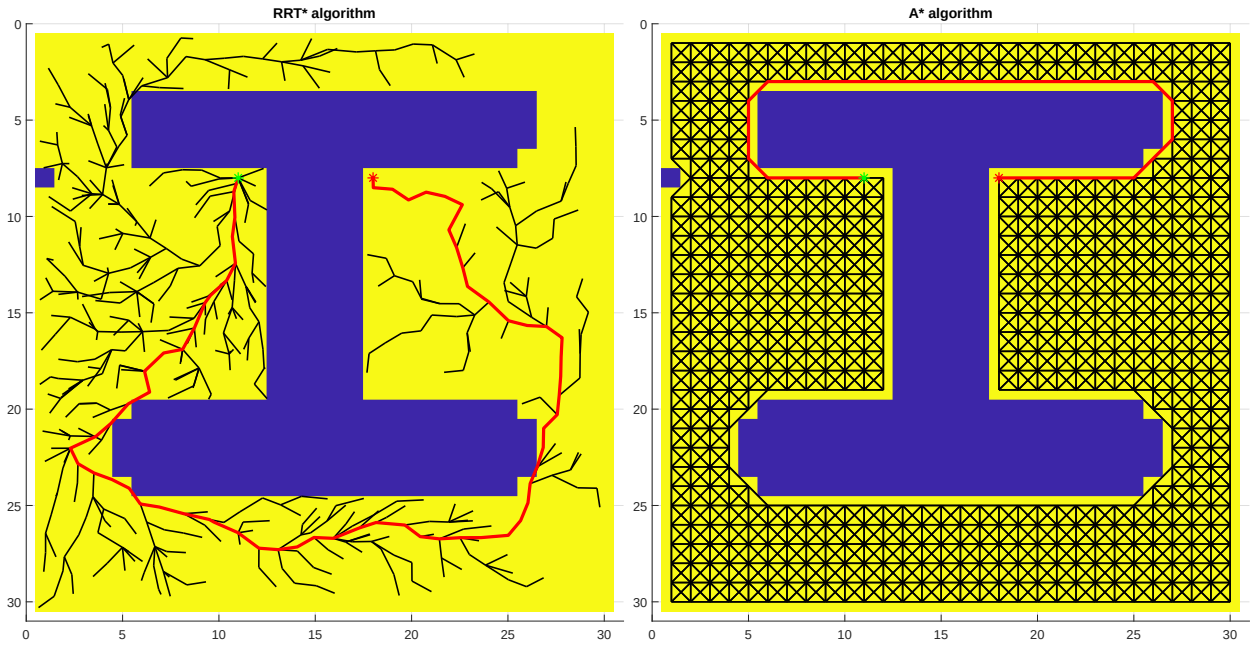

Figure 9: Path generated by RRT* (left) and A* (right) on the same scenario.

The path generated by RRT* is clearly suboptimal, choosing a much longer path only because fewer obstacles are present in that area and so higher chances of connectivity for the randomly generated nodes are present.
A similar case is shown in Figure 10.

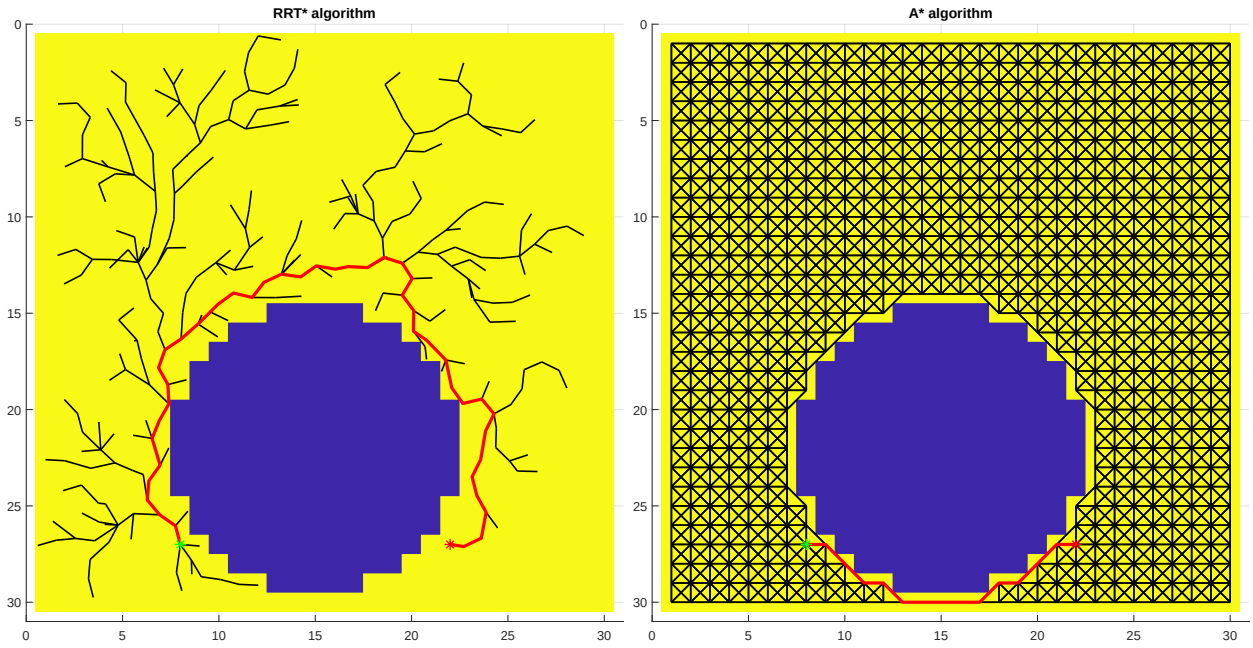

Figure 10: Path generated by RRT* (left) and A* (right) on the same scenario.

However, the RRT* algorithm is asymptotically optimal, meaning that the higher the number of nodes that the algorithm place, the more optimal the path becomes. This is a very important property, as it allows trading off between the time spent in finding a solution and the quality of the solution itself.
By removing the interruption of the algorithm after the first solution is found, we can observe how the path generated by RRT* becomes more and more optimal as the number of iterations increases. Figure 11 shows the path generated by RRT* after 1000 iterations on two different scenarios. One can appreciate how, despite the number of iterations, in the case of the circle the path is still suboptimal.

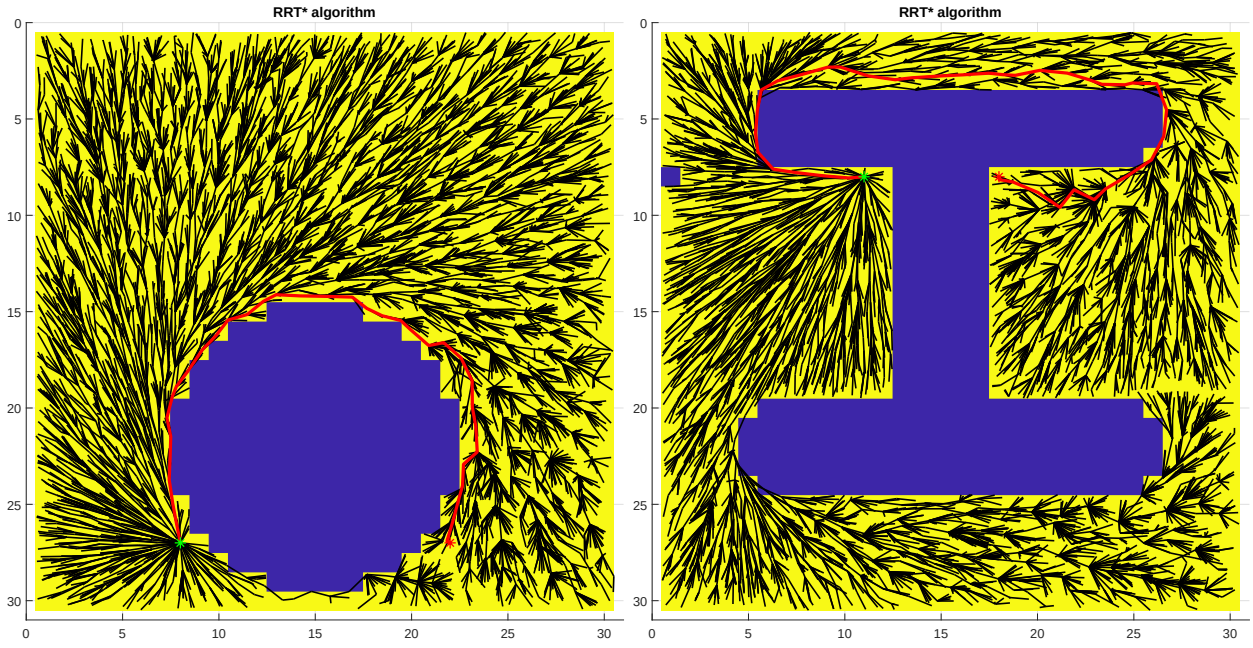

Figure 11: Asymptotic optimality of RRT* algorithm. Both the images refer to 10000 iterations of the algorithm.

In Table 6 are reported the path length and the time taken by RRT* algorithm as a function of the number of iterations performed. Results are relative to the right-hand side scenario of Figure 11.

Iterations	Path length (m)	Elapsed time (ms)
1000	68	324
5000	60	12775
10000	44	69069

Table 6: Path length and time taken by RRT* algorithm after different number of iterations.

As expected, the path length decreases as the number of iterations increases, but the time taken by the algorithm increases significantly. Thinking about deploying the algorithm on a real robot which needs the planner component to run in real-time or nearly real-time, we clearly understand that the number of iterations must be limited, and asymptotic optimality is not a property that can be exploited in practice.

5.2 Multidimensional Workspace

In this section we analyze the performance of the two algorithms on a 3D scenario, adding a third dimension to the problem. The scenario is composed of a 3D grid of obstacles, which are randomly generated in the space. Figure 12 shows a 3D view of the scenario and a stack of slicing along the Z axis.

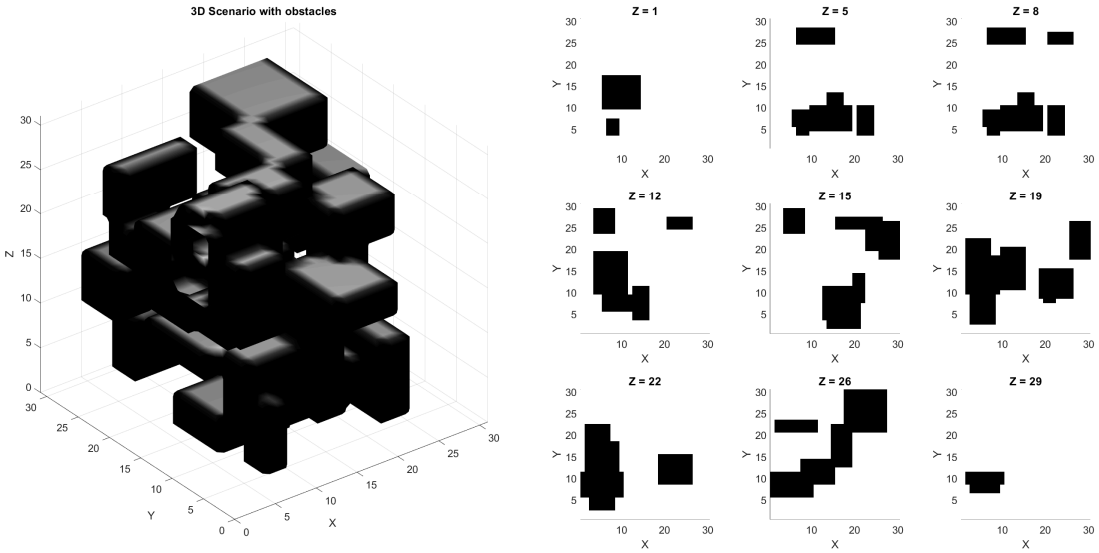

Figure 12: 3D grid of obstacles used as a scenario for the tests.

With zero-to-little modifications to the algorithms, we run both the grid-based and the sample-based algorithms on the same 3D scenario. Thanks to the increased dimensions of the problem, we clearly observe the huge difference in performance between the two class of algorithms. Table 7 summarizes the results of the tests, while Figure 13 shows the paths generated by A*, RRT and RRT* algorithms.

Algorithm	Elapsed time (ms)	Tree dimension	Path length (m)
A*	2435229 (∼40[min])	20803	87
RRT	55	191	69
RRT*	273	397	69

Table 7: Path length and time taken by each algorithm on the same 3D scenario.

One can notice that the path generated by A* is longer than the one generated by RRT and RRT*. This can be explained by the fact that the graph used to represent the workspace has been generated without allowing diagonal connections. This means that, considering a 3D unitary volume of the grid, for the A* to go from one corner to the opposite one, it needs to go through all the 3 cells edges, while instead the RRT and RRT* algorithms are able to directly connect the two corners, without going through all the intermediate cells. Thanks to this observation, a simple math shows that on average, if the diagonal connection were allowed, the path length would have been reduced by a factor of $\sqrt{3}$. This means that the path length generated by A* would have been around $\frac{87}{\sqrt{3}} \approx 50$ m, which is lower than the path length generated by RRT and RRT*, indeed proving once again the concept of optimality of the grid-based algorithms.

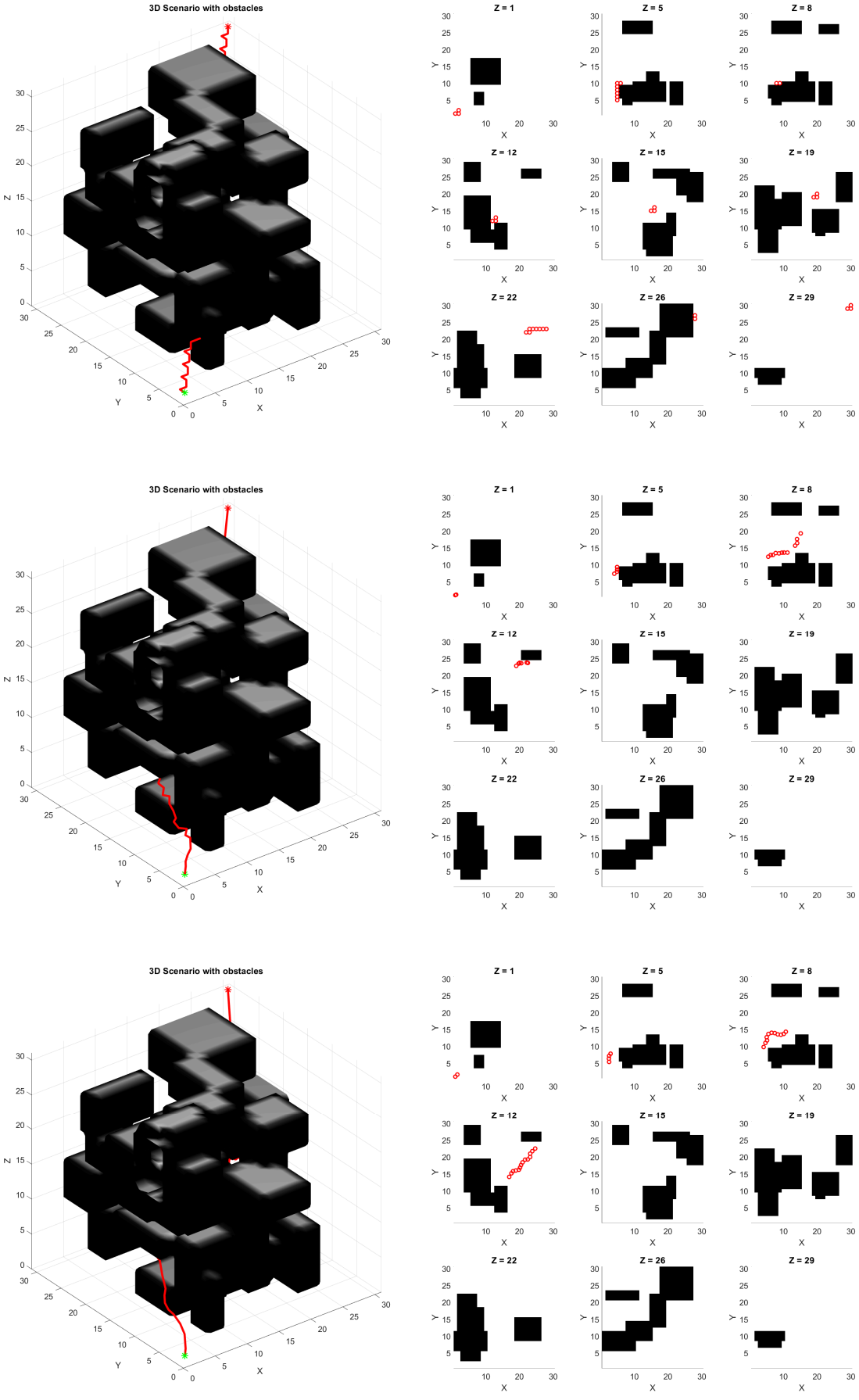

Figure 13: Path generated by A* (top), RRT (middle) and RRT* (bottom) algorithms on the same 3D scenario.

From Table 7 it's clear how the use of grid-based algorithm for single-query planning problem is unfeasible. These tests, again, highlight the strong adaptability of the sample-based algorithms to high-dimensional problems, while grid-based algorithms are not able to cope with the increased complexity of the problem.
Notice that a minor modification to both the RRT and RRT* algorithms have been applied in order to further reduce the computational time. In particular, we have found that the original algorithms excelled at fast exploration of the workspace, reaching in little amount of time every distinct volume closed by obstacles. However, once arrived at the volume of the goal configuration, they failed in creating the final connection and close the planning problem. In order to avoid useless sampling, we have added a simple logic inside the main algorithm loop that checks if the euclidean connection between the lastly added node and the goal is collision-free or not. By doing so, the time required for planning drops significantly.
Listing 5 shows the implementation for this strategy.

```
while (Node.euclideanDistance(q_new, node_final.state) > G.step_length && iter < G.
    max_iter)
    % ...
    if (G.isValidConnection(q_new, node_final.state, map))
        % We have free space along the connection with the goal
        % Stop sampleing and directly add the goal node to the tree
        break
    end
end
```

Listing 5: Implementation of the fast goal connection for sampling-based algorithm.

6 Conclusions

In this project we have implemented tree sampling-based single-query motion planning algorithms, specifically RRT, RRT*, and RRT-kinematic. We have run them on a set of predefined scenarios, and we have compared their performance in terms of path length, computation time, and number of nodes. A comparison with grid-based planning algorithms has also been performed, showing that the tree sampling-based algorithms are more efficient in terms of computation time but lacks the optimality property of grid-based ones.
We are confident that an optimized implementation of both the classes of algorithms would lead to a more efficient and effective solution for motion planning problems. However, we believe that our implementation has already been capable of providing a good overview of the performance of the algorithms and their potential applications in real-world scenarios. In particular, a massive improvement in the performance of the tree sampling-based algorithms can be achieved by using a more efficient data structure for the tree, such as a `KD-tree`. This would allow for a faster nearest neighbor search, which is a key operation in the RRT algorithm. Moreover, we are confident to say that the combination of a quick exploration of the space and a local grid-based search is the key to achieving optimality in motion planning problems. As demonstrated in the multidimensional case, the capabilities of sampling-based algorithms to quickly explore vast volumes of space in a short time are notable. However, once the first sampled is placed in the vicinity of the goal, many more samples are needed before effectively connecting the goal node. This, could be solved by using a local grid-based search, which would work as a local optimizer, reducing the effort needed to simply connect the goal node to the tree.
In conclusion, sampling-based algorithms represent a robust and scalable approach for tackling motion planning problems, especially in high-dimensional or dynamic environments, where fast replanning is often required. Due to their weakness in terms of optimality, hybrid approaches that combine the strengths of both sampling-based and grid-based methods are likely to be the most effective in practice. The integration of these two paradigms could lead to planners that are not only computationally efficient but also capable of producing high-quality solutions across a wide range of applications.
We believe that the future of motion planning lies in the development of hybrid algorithms that can leverage the strengths of both sampling-based and grid-based methods.

References

[1] Xiangkui Jiang, Zihao Wang, and Chao Dong. A path planning algorithm based on improved rrt sampling region. *Computers, Materials and Continua*, 80(3):4303–4323, 2024.

[2] Sertac Karaman and Emilio Frazzoli. Sampling-based algorithms for optimal motion planning. *International Journal of Robotic Research - IJRR*, 30:846–894, 06 2011.