UNIVERSITÀ DEGLI STUDI DI NAPOLI

FEDERICO II

FACOLTA' DI INGEGNERIA

DIPARTIMENTO DI INGEGNERIA ELETTRICA E DELLE
TECNOLOGIE DELL'INFORMAZIONE

CORSO DI LAUREA MAGISTRALE IN
INGEGNERIA DELL'AUTOMAZIONE E
ROBOTICA

Final technical report Field and Service Robotics

Navigation planning and control for a car-like robot

Professor:
Fabio Ruggiero

Candidate:
Antonio Manzoni P38000234

Academic Year 2023/2024

Contents

Chapter 1

Introduction

1.1 Navigation planning problem: a general view

Before speaking about the aim of the project and how it has been implemented it might be useful to understand in the first place the problem we are going to deal with. Navigation planning is widely found in different applications, such as:

- Autonomous vehicles navigating through traffic
- Robot arms in manufacturing picking and placing objects
- Unmanned aerial vehicles (UAVs) navigating around obstacles
- Surgical robots planning precise movements inside the human body
- Warehouse robots efficiently moving inventory
- Space exploration rovers navigating unknown terrains

The navigation planning problem is to compute a continuous path that connects a start configuration 'S' and a goal configuration 'G', while avoiding collision with obstacles. Navigation planning can be seen as the integration of path planning and trajectory planning into a complete system for robot navigation. The first finds a geometrically valid path in space that avoids obstacles, determining "where" the robot should go, while the second adds the temporal law: it takes the path and defines "when" and "how" the robot should travel it, considering velocities, accelerations, and dynamic constraints. Path planning problems come in various forms depending on the context:

- **Sampling-Based Planning**: rather than trying to represent the entire configuration space explicitly, sampling-based methods like Rapidly-Exploring Random Trees (RRT) randomly sample points in the space and try to connect them to build a graph representing possible paths.
- **Grid-Based Planning**: the environment is discretized into a grid, and algorithms like A*, Dijkstra's algorithm, search for paths through this grid. These methods are particularly useful in well-mapped environments.

To be even more precise, navigation planning also includes:

- Real-time localization
- Dynamic obstacle detection
- Replanning capability during execution

but for the sake of simplicity these topics will not be covered in the project.

1.2 Aim of the project

The aim of the project is to control a car-like robot in such a way as to perform a navigation planning task with three different strategies of path generation:

- A Sampling-Based approach using **Rapidly-Exploring Random Tree*** (RRT*) algorithm
- Two Grid-Based approaches using **A*** and **Dijkstra** algorithms

An offline planning has been considered, so the environment is known and the obstacles are static. Many robots have movement restrictions that make planning more challenging. The chosen robot for this project is not an unicycle model, which can turn in place and follow discontinuous paths, so the path planning step has been made smooth in such a way as to respect the constraints of our model. In particular, for the sampling-based approach the path has been made smooth using **Pseudo-Clothoid curves** while for both the grid-based approaches the **Reeds-Shepp curves** (RSC) have been used. From the path it is now possible to add the time law to retrieve the trajectory that the robot must follow. An **Input-Output Linearization** controller has been employed to fulfill the control part of the project. The path planning and the trajectory planning steps have been implemented in MATLAB while the controller has been implemented in SIMULINK. To be as close as possible to the reality it has been chosen as real model the **YUHESEN FR-09 PRO**.

Figure 1.1: Real robot model

Chapter 2

Modeling and analysis of car-like robot

2.1 Kinematic model of car-like robot

In this chapter the focus is pointed on the study of the kinematic model of a car-like robot. The configuration of this robot is represented by the position and orientation of its main body in the plane, and by the angle of the steering wheels. Two velocity inputs u_1 and u_2 are available for motion control. The main feature of the kinematic model of wheeled mobile robots is the presence of non-holonomic constraints due to the rolling without slipping condition between the wheels and the ground.

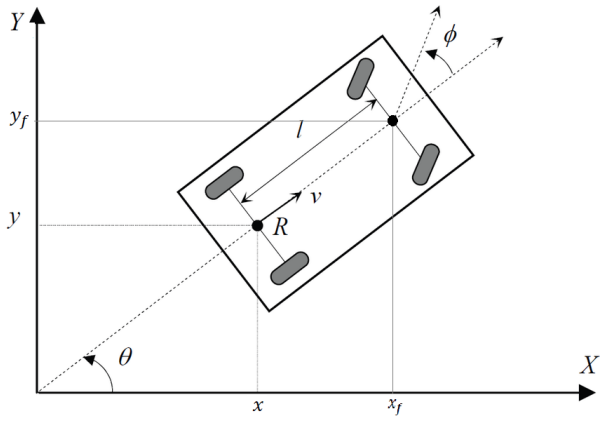


Figure 2.1: Configuration of a car-like robot

For the sake of simplicity, it is assumed that the two wheels on each axle (front and rear) collapse into a single wheel located at the midpoint of the axle, thus creating a bicycle model. For our analysis we will consider the front wheel as steerable, whereas the rear wheel's orientation is fixed. The generalized coordinates are: $q = \begin{bmatrix} x & y & \theta & \phi \end{bmatrix}^T$, where x, y are the cartesian coordinates of the rear wheel, θ measures the orientation of the car body with respect to the x axis, and ϕ is the steering angle. The system is subject to two non-holonomic constraints, one for each wheel:

$$\begin{aligned} \dot{x}\sin\theta - \dot{y}\cos\theta &= 0 \\ \dot{x}_f \sin(\theta+\phi) - \dot{y}_f \cos(\theta+\phi) &= 0 \end{aligned} \tag{2.1}$$

with x_f, y_f denoting the cartesian coordinates of the front wheel. By using the rigid-body constraint:

$$\begin{aligned} x_f &= x + \ell\cos\theta \\ y_f &= y + \ell\sin\theta, \end{aligned} \tag{2.2}$$

where ℓ is the distance between the wheels, the second kinematic constraint becomes:

$$\dot{x}\sin(\theta+\phi) - \dot{y}\cos(\theta+\phi) - \dot{\theta}\ell\cos\phi = 0 \tag{2.3}$$

The Pfaffian constraint matrix is:

$$A^T(q) = \begin{bmatrix} \sin\theta & -\cos\theta & 0 & 0 \\ \sin(\theta+\phi) & -\cos(\theta+\phi) & -\ell\cos\phi & 0 \end{bmatrix} \tag{2.4}$$

and has constant rank equal to 2. If the car has rear-wheel drive, the kinematic model is derived as:

$$\dot{q} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \\ \dot{\phi} \end{bmatrix} = \begin{bmatrix} \cos\theta \\ \sin\theta \\ \frac{\tan\phi}{\ell} \\ 0 \end{bmatrix} u_1 + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} u_2 \tag{2.5}$$

In this context, $u_1 = v/\cos\phi$ and $u_2 = \omega$ represent the driving and steering velocity input, respectively. A model singularity is observed for the steering angle $\phi = \pm\pi/2$, where the first vector field exhibits a discontinuity. This phenomenon is analogous to the vehicle becoming immobilized when the front wheel is aligned with the longitudinal axis of the body. Nevertheless, the significance of this singularity is constrained by the limited range of the steering angle ϕ in the majority of practical scenarios.

2.2 Controllability analysis

Equation 2.5 may be rewritten as:

$$\dot{q} = g_1(q)v + g_2(q)\omega \quad \text{with} \quad g_1 = \begin{bmatrix} \cos\theta \\ \sin\theta \\ \frac{\tan\phi}{\ell} \\ 0 \end{bmatrix}, g_2 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \tag{2.6}$$

The above system is non-linear, driftless, and there are less control inputs than generalized coordinates. Although every driver's experience suggests that a car-like robot should be fully controllable, it is not trivial to establish such a property on a mathematical basis. A useful tool for testing the controllability of driftless nonlinear systems is the *Lie Algebra rank condition*. To verify this, it is necessary to establish whether the dimension of the accessibility distribution, denoted by $\Delta_a(q)$, is equal to the number of degrees of freedom, denoted by n, of the system under consideration. In this case, n is equal to 4. Since:

$$\Delta_a = \text{span}\{g_1, g_2, g_3, g_4\} \tag{2.7}$$

where:

$$g_3 = [g_1, g_2] = \left[0,\, 0,\, -\frac{1}{\ell\cos^2\phi},\, 0\right]^T \tag{2.8}$$

$$g_4 = [g_1, g_3] = \left[-\frac{\sin\theta}{\ell\cos^2\phi},\, \frac{\cos\theta}{\ell\cos^2\phi},\, 0,\, 0\right]^T \tag{2.9}$$

In accordance with prior statements, the system will be completely non-holonomic if the resulting matrix

$$F = \begin{bmatrix} \cos\theta & 0 & 0 & -\frac{\sin\theta}{\ell\cos^2\phi} \\ \sin\theta & 0 & 0 & \frac{\cos\theta}{\ell\cos^2\phi} \\ \frac{\tan\phi}{\ell} & 0 & -\frac{1}{\ell\cos^2\phi} & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \tag{2.10}$$

is full rank. Since $rank(F) = 4 \Rightarrow dim(\Delta_a) = 4 \Rightarrow$ the system is completely non-holonomic.

Chapter 3

Navigation planning

3.1 Sampling-based approaches

Sampling-based planners are a class of motion planning algorithms that generate a roadmap of the robot's configuration space by sampling random points and connecting them with feasible paths. These methods are particularly suited for high-dimensional spaces as they do not require an elaborate modeling of the environment. They are probabilistically complete, meaning that they can find a solution with probability approaching one as the number of samples increases. Moreover, variants like RRT* can provide asymptotically optimal paths.

3.1.1 RRT and RRT*

The RRT algorithm is quite straightforward. A random point is generated, the nearest node in the tree is found, and the tree is extended from the nearest node toward each random point. Each time a vertex is created, a check must be made that the vertex lies outside of an obstacle. Furthermore, chaining the vertex to its closest neighbor must also avoid obstacles. The algorithm ends when a node is generated within the goal region, or a limit is hit. To generate the optimal path, the RRT* algorithm is proposed. First, RRT* records the distance each vertex has traveled relative to its parent vertex. This is referred to as the `cost()` of the vertex. After the closest node is found in the graph, a neighborhood of vertices in a fixed radius from the new node is examined. If a node with a cheaper cost than the proximal node is found, the cheaper node replaces the proximal node. The second difference RRT* adds is the **rewiring** of the tree (Fig 3.1). After a vertex has been connected to the cheapest neighbor, the neighbors are again examined. Neighbors are checked if being rewired to the newly added vertex will make their cost decrease. If the cost does indeed decrease, the neighbor is rewired to the newly added vertex. This feature makes the path more smooth and we can appreciate it with increasing the number of iterations (Fig 3.2). Although obtaining the optimal path is essential in some applications, the RRT* algorithm usually consumes much time and huge memory usage to converge to the optimal path. The majority of computing effort comes from obstacle avoidance.

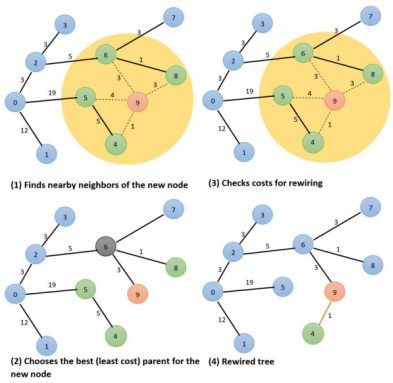


Figure 3.1: Rewiring

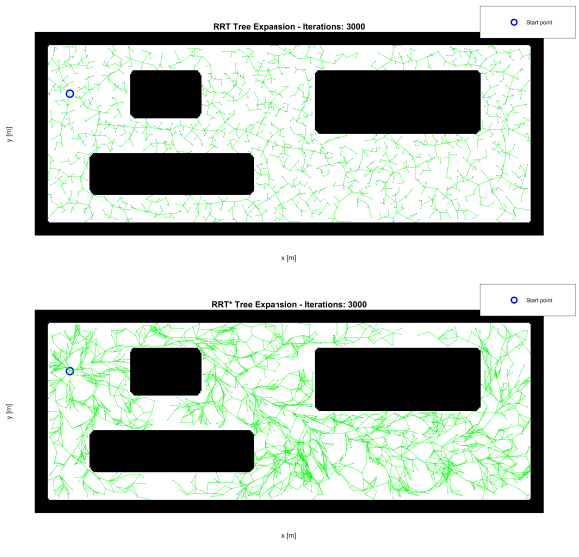


Figure 3.2: Comparison between RRT and RRT*

3.1.2 RRT* implementation with Pseudo-Clothoid curves

The RRT* approach shown in the bottom of the figure 3.2 and implemented in the code ***'rrt_star_no_path.m'*** inside the folder ***'RRT_RRTstar_Comparisons'*** has been modified in such a way as to retrieve a path from a start node to a goal node which is feasible for our robot. To achieve this, the file ***'rrtStar_smooth.m'*** has been created. The code implements the RRT* algorithm as follows:

- **Initialization**: the expansion of the tree originates from the start point with an initial cost equal to 0. Also the variables that keep track of the best path are initialized.
- **Main loop**: it executes a maximum number of iterations and implements the following steps:
  1. Random sampling towards the goal.
  2. Find the nearest point $q_{nearest}$ to the random point through the computation of the euclidean distance.
  3. Compute the new point q_{new} steering from the nearest point to the random point with the set maximum distance. Check if the new point is outside the map or in collision with an obstacle, if it occurs then skip to the next iteration. This step includes the generation of the smooth curve.
  4. Find the nodes from the new point that are within a certain radius (**neighbors**) and from them extract the node q_{min} which minimizes the cost:
  $$cost(q_{min}) = cost(q_{nearest}) + distance(q_{min}, q_{nearest}) \tag{3.1}$$
  5. Add q_{new} to the tree with q_{min} as **parent** and memorize the total cost.
  6. Rewiring phase: all the neighbors are examined excluding the parent (this node has been already connected in the best way possible to the new node) and for each of them it is verified whether the total cost of reaching from the root through the new node is lower than the cost of the current path. In other words, a check is performed to determine if a more efficient alternative path exists that uses the new node as an intermediate point to reach the existing neighboring node. To evaluate this, a connection between the new node and the neighboring node is made. The algorithm then calculates the potential cost of this new connection by adding the cost of the new node (the cost to reach the new node from the root) and the length of the newly generated connection. If this sum is less than the current cost of the neighboring node, and if the path does not collide with obstacles in the map, the algorithm proceeds by cutting the existing connection of the neighboring node with its current parent and reconnects it to the new node instead. When a better path is found, this improvement is propagated to all the tree descendants. All this process can be written in a more compact way in pseudocode:
  $$\forall\, q_{\text{near}} \in Q_{\text{near}} :$$
  $$\text{If } \text{cost}(q_{\text{new}}) + \text{distance}(q_{\text{new}}, q_{\text{near}}) < \text{cost}(q_{\text{near}}) :$$
  $$\text{parent}[q_{\text{near}}] \leftarrow q_{\text{new}}$$
  $$\text{cost}[q_{\text{near}}] \leftarrow \text{cost}(q_{\text{new}}) + \text{distance}(q_{\text{new}}, q_{\text{near}})$$
  7. When a node is sufficiently close to the goal the complete path from the goal to the start is reconstructed following the parents and an update of the best path is made if this has a better cost than the previous ones.
- **Helper functions**: a series of functions, which are recalled by the main loop, have been implemented to build the code in a modular and more readable way.

Recent studies have proposed integrating Clothoid curves into RRT-based path-planning algorithms to ensure smoothness and prevent jerky movements, which is crucial for robots that need to comply with physical motion limitations. Clothoid curves are a family of curves which have the advantage that the curvature varies linearly with the length of the curve:

$$c(s) = ks + c_i \tag{3.2}$$

Where k is the rate of change of curvature (sharpness) of the curve and c_i is the initial curvature and s is the curve length. In the Clothoid curve, we define the curvature as positive if the rotational change of the tangent vector is clockwise. This feature enables the avoidance of obstacles and compliance with the kinematic constraints of the robot's motion. Additionally, the centrifugal acceleration is continuous, preventing sudden changes that could otherwise impair the robot's stability. We define $x(s)$ and $y(s)$ to denote the x-axis coordinates and y-axis coordinates at the arc length s:

$$x(s) = \sqrt{\frac{2}{k}} \cdot \mathrm{C}\left(\sqrt{\frac{k}{2}} \cdot s\right) \tag{3.3}$$

$$y(s) = \sqrt{\frac{2}{k}} \cdot \mathrm{S}\left(\sqrt{\frac{k}{2}} \cdot s\right) \tag{3.4}$$

Where $C(z)$ and $S(z)$ are the Fresnel cosine integral and sine integral:

$$\mathrm{C}(z) = \int_0^z \cos\left(\frac{\pi}{2}u^2\right) du \tag{3.5}$$

$$\mathrm{S}(z) = \int_0^z \sin\left(\frac{\pi}{2}u^2\right) du \tag{3.6}$$

Given an initial posture, sharpness, and the distance along the curve, the clothoid can be described by the following equations:

$$x = x_0 + \cos(\theta_0) \cdot x(s) - \sin(\theta_0) \cdot y(s) \tag{3.7}$$

$$y = y_0 + \sin(\theta_0) \cdot x(s) + \cos(\theta_0) \cdot y(s) \tag{3.8}$$

$$\theta = \theta_0 + k \cdot s \tag{3.9}$$

Where x_0, y_0 are the two-dimensional coordinates of the given pose and θ_0 is the angle between the x-axis and the pose direction.

Figure 3.3 shows an example of the clothoid curve.

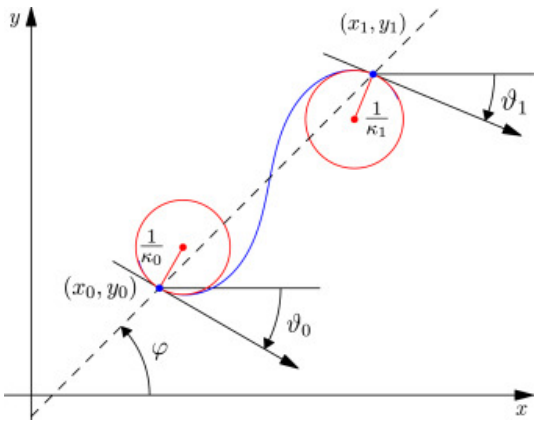


Figure 3.3: Clothoid curve

Since those curves rely on the Fresnel integrals they are computationally demanding and could be affected by approximation errors. So, a simplified version has been implemented. The concept is to generate a curve with a

variable turning radius that decreases as the angle increases, which means that the radius gets smaller as you progress around the curve, in this way:

$$r(\theta) = \frac{r_0}{1 + \alpha|\theta - \theta_0|} \tag{3.10}$$

Where r_0 is the initial radius of the curve, α is the rate of change of the angle and $|\theta - \theta_0|$ is the covered angular distance. This creates a spiral-like path where the curvature increases gradually, like a clothoid. This approach needs a fixed center of the arc which is calculated by using the current position and heading direction d_n, which is equal to -1 for the left curves and +1 for the right curves, in this way:

$$\begin{bmatrix} x_c \\ y_c \end{bmatrix} = \begin{bmatrix} x_n \\ y_n \end{bmatrix} + r(\theta) \cdot \begin{bmatrix} \cos\left(\theta_n + d_n \frac{\pi}{2}\right) \\ \sin\left(\theta_n + d_n \frac{\pi}{2}\right) \end{bmatrix} \tag{3.11}$$

This center is perpendicular to the current heading direction d_n. For each point of the curve the angle θ is calculated as the current angle plus how far we've turned:

$$\theta = \theta_n + d_n \cdot s \tag{3.12}$$

Note that d_n doesn't have the same role of k, which has been replaced by α. Finally, we can compute the x and y coordinates directly without using a numerical integration:

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x_c \\ y_c \end{bmatrix} + r(\theta) \cdot \begin{bmatrix} \cos(\theta - d_n \frac{\pi}{2}) \\ \sin(\theta - d_n \frac{\pi}{2}) \end{bmatrix} \tag{3.13}$$

The problem that arises is that the radius could become too small at the end of the curve and so the curvature might be unfeasible to follow. To overcome this, the pseudo-clothoid has been combined with a cubic Bézier curve, which has been implemented as follows:

$$\mathbf{B}(t) = (1-t)^3\mathbf{P}_0 + 3(1-t)^2 t\mathbf{P}_1 + 3(1-t)t^2\mathbf{P}_2 + t^3\mathbf{P}_3, \quad 0 \le t \le 1. \tag{3.14}$$

Where:

- Po is the start point of the curve and coincides with the last point of the pseudo-clothoid.
- P1 and P2 are two intermediate points which affect the form of the curve.
- P3 is the target point, which is the sampled random point.

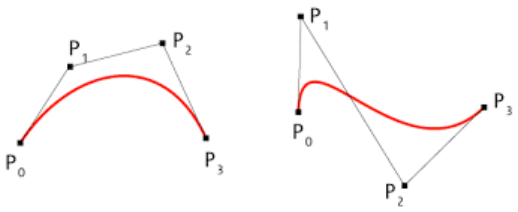


Figure 3.4: Cubic Bézier curves

3.2 Search-based approaches

Search-based planners are a class of motion planning algorithms that rely on searching through a discretized representation of the configuration space to find a feasible path. These planners typically employ graph-based search algorithms, such as A* or Dijkstra's algorithm, to find the shortest or the optimal path from the robot's initial position to the goal. Search-based planners include:

- **Grid-based Planners**: These planners discretize the environment into a grid and use search algorithms to navigate through the cells while avoiding obstacles.
- **Visibility Graph Planners**: These planners construct a graph by connecting the robot's initial position, goal, and obstacle vertices, and use search algorithms to find the shortest path along the visibility graph.

These methods are complete and optimal in the discretized space but suffer from the "curse of dimensionality" as the size of the discretized space grows exponentially with the number of dimensions.

3.2.1 Dijkstra and A* algorithms

Dijkstra's algorithm creates a set of "visited" and "unvisited" nodes. The algorithm assigns a distance of zero to the starting node and all other nodes' distance values are set to infinity. Then, each neighbor is visited and its distance from the current node is determined, if the distance is less than the previously defined distance value, then the value is updated. Once all neighboring values are updated, the algorithm adds the current node to the 'visited' set and repeats the process for the next neighboring node with the shortest distance value. The algorithm continues until all nodes have been moved from "unvisited" to "visited". Dijkstra's Algorithm is guaranteed to find a shortest path from the starting point to the goal, as long as none of the edges has a negative cost. This produces an optimal solution, but the computation is expensive, so A* has been introduced.

A* is another path-finding algorithm that extends Dijkstra's algorithm by adding **heuristics** to stop certain unnecessary nodes from being searched. The A-star algorithm uses a global cost function:

$$f(n) = g(n) + h(n) \tag{3.15}$$

where $g(n)$ is the cost function of the path from the initial state to the node n and $h(n)$ is the heuristic function, which is used to rank the nodes and decide which node to expand next. This method replaces the minimum distance criterion used in Dijkstra, and so, every time the algorithm finds a best path, instead of updating the minimum distance we update the $g(n)$ and $h(n)$ functions. Heuristics should never overestimate the real cost to be admissible. At one extreme, if $h(n) = 0$, then only $g(n)$ plays a role, and A* turns into Dijkstra's Algorithm, which is guaranteed to find a shortest path. There are several ways to define the heuristic function, common choices are the **Euclidean distance** and the **Manhattan distance** (*"Taxicab geometry"*). The first measures the straight-line distance between two points (Eq.3.16), while the second measures the distance between two points by adding the absolute differences of their coordinates (Eq.3.17).

$$h_{euclidean}(n) = \sqrt{(x_n - x_{goal})^2 + (y_n - y_{goal})^2} \tag{3.16}$$

$$h_{manhattan}(n) = |x_n - x_{goal}| + |y_n - y_{goal}| \tag{3.17}$$

Where x_n and y_n are the coordinates of the current node, while x_{goal} and y_{goal} are the coordinates of the goal.

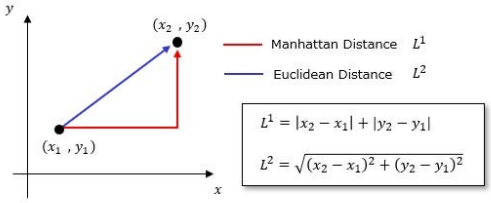


Figure 3.5: Heuristics

3.2.2 Implementations with Reeds-Shepp curves exploration

The two implementations combine the classical approaches of Dijkstra and A* algorithms to find the shortest and the best path respectively with Reeds-Shepp curves to generate smooth paths for our non-holonomic robot. Since the two algorithms share similar operational principles, their implementations naturally exhibit parallel structures. The codes have been produced inside the two files: ***'dijkstra_planning.m'*** and ***'a_star_planning.m'*** and are structured as follows:

- **Initialization**: a struct containing all the data used to perform the exploration has been initialized. Then, the *occupancy matrix*, which discretizes the map with a grid, has been defined. This matrix maps every cell of the grid with a physical position while keeping tracks of the obstacles inside the environment and also memorizes the cells that have been already visited during the exploration (1 stays for obstacles and visited nodes while 0 represents the free space and the unvisited nodes). We need to clarify that we are still referring to the spatial domain in a discretized version and not to the graphs' domain used during the path finding phase. The *queue* initially contains only the start point with a 0 angle. In the A* algorithm the queue also contains the two costs $g(n)$ from the start point and the estimated cost $h(n)$ to the objective.

- **Exploration loop**: at each iteration the first point of the queue is taken and removed from the queue, then the new potential points are computed with the RSC method. For sake of simplicity, we assume that the robot can perform only forward movements, so, referring to the figure 3.6, only $C^+_{a,l}$, $C^+_{a,r}$ and S^+_d are allowed.

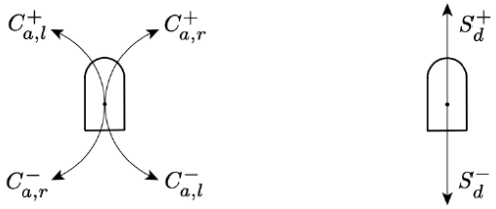


Figure 3.6: Reeds-Sheep curves

The new points are calculated in the following order: following the straight line, then following the curve to the right and finally the curve to the left; at the generation of each subsequent point, it verifies that this has not already been visited, it checks the collision with obstacles along the curve and if both are satisfied the node is added to the queue and the occupancy matrix is updated. The loop continues until there are no more points inside the queue. A* introduces an intelligent prioritization of the nodes; in fact, while Dijkstra selects the first node from the queue, A* prioritizes the one which has the minimum total cost (Eq.3.15), which is then removed by the queue. Moreover, A* verifies if the current node is close enough to the objective, if this is true then the exploration stops.

- **Adjacency matrix**: The exploration result is now converted into an adjacency matrix that takes into account how nodes are connected to each other and the relative distances, so now we are into the graphs' domain. This matrix is useful in the next step to find the path.

- **Shortest path/best path finding**: Now the two algorithms find the shortest/best path inside the network of the created connections, as has already been explained in the previous section. Two things must be highlighted:

  1. The heuristic used by the A* algorithm in this phase works in the graphs' domain operating on the adjacency matrix, which represents connections which have already been validated, so doesn't worry anymore onto the collisions and boundaries but takes care only to the path cost.

  2. Since the algorithms operate in a discretized domain and the parameters for exploration are fixed, it could happen that it is impossible to reach exactly the goal coordinates with the normal exploration. To overcome this problem, a smooth forced connection, with a spline curve, has been implemented in the last part of the path between the goal and a close point to it. To be more robust, a check for obstacles has been made.

- **Helper functions**: Like for RRT* implementation, there have been also implemented functions to make the code more modular and readable.

3.3 Trajectory planning

The goal of trajectory planning is to generate the reference inputs to the motion control system which ensures that the robot executes the planned trajectories. Planning consists of generating a time sequence of the values attained by an interpolating function (typically a polynomial) of the desired trajectory. The steps necessary to obtain a feasible trajectory have been implemented inside the files ***'des_trajectory.m'*** and are the following:

- **Representation of the geometric path:** from the geometric path $q(s)$, where s is the normalized arc length of the path, is possible to retrieve the Cartesian coordinates (x, y) of the points, that represent the robot's position in the 2D space without specifying a timing law.

- **Timing law generation:** the timing law $s(t)$ is defined in such way to map the time t to the arc length s of the geometric path. In other words, the timing law is doing a parametrization of the geometric path coordinates $x(s)$ and $y(s)$ in terms of time, and so $x(t)$ and $y(t)$. A 5th-order polynomial for RRT* and a piecewise cubic Hermite polynomial for A* and Dijkstra have been chosen to fulfill this task. These time laws ensure a smooth temporal distribution along the path, enabling the robot to move steadily from point to point while providing a consistent evolution of the path in time. The trajectory has been discretized using a more refined temporal grid for the interpolation. In MATLAB, for RRT*, this step was achieved by using the functions `polyfit` to generate the 5th order polynomial approximation and `polyval` that allows the polynomial model to be evaluated at any intermediate point. For the other two approaches a simple piecewise cubic interpolation with `pchip` was implemented.

- **Interpolation:** the coordinates mapped in terms of time now had to be interpolated. For each instant in the new grid, the corresponding value of s(t) is calculated and the coordinates (x(t), y(t)) are determined by cubic spline interpolation, subsequently applying a smoothing function to ensure continuity. In MATLAB

this step was achieved by using the function `interp1` with the `spline` option for RRT*. For the other two approaches it has been used the function `ppval`.

- **Desired states and kinematics computation:** in this last stage the desired states of the trajectory: $\begin{bmatrix} x_d & y_d & \theta_d & \phi_d \end{bmatrix}^T$ have been computed to be fed into the robot's controller, and also the linear velocity $v_d(t)$, angular velocity $\omega_d(t)$, orientation angle $\theta_d(t)$, and steering angle $\phi_d(t)$ have been calculated. The velocity inputs must remain confined inside their limits: $v_{\max} = 5\,\mathrm{m/s}$ and $\omega_{\max} = 0.56\,\mathrm{rad/s}$. To ensure this safeguards on the velocities have been implemented inside the code.

A feasible and smooth desired output trajectory is given in terms of the cartesian position of the car rear wheel:

$$x_d = x_d(t), \quad y_d = y_d(t), \quad t \geq t_0 \tag{3.18}$$

This natural way of specifying the motion of a car-like robot has an appealing property. In fact, from this information it is able to derive the corresponding time evolution of the remaining coordinates (state trajectory) as well as of the associated input commands (input trajectory) as shown hereafter. The desired output trajectory (Eq. 3.18) is feasible when it can be obtained from the evolution of a reference car-like robot:

$$\dot{x}_d = \cos\theta_d v_d \tag{3.19}$$

$$\dot{y}_d = \sin\theta_d v_d \tag{3.20}$$

$$\dot{\theta}_d = \tan\phi_d \frac{v_d}{l} \tag{3.21}$$

$$\dot{\phi}_d = \omega_d \tag{3.22}$$

for suitable initial conditions $(x_d(t_0), y_d(t_0), \theta_d(t_0), \phi_d(t_0))$ and piecewise-continuous inputs $v_d(t)$, for $t \geq t_0$. Solving for v_d from Eq. 3.19 and 3.20 gives for the first input:

$$v_d(t) = \pm\sqrt{\dot{x}_d^2(t) + \dot{y}_d^2(t)} \tag{3.23}$$

where the sign depends on the choice of executing the trajectory with forward or backward car motion, respectively. Dividing Eq. 3.20 by 3.19, and keeping the sign of the linear velocity input under account, the desired orientation of the car is computed as:

$$\theta_d(t) = \mathrm{ATAN2}\left\{\frac{\dot{y}_d(t)}{v_d(t)}, \frac{\dot{x}_d(t)}{v_d(t)}\right\} \tag{3.24}$$

Differentiating Eq. 3.19 and 3.20, and combining the results so as to eliminate $\dot{v}_d$, the second input is obtained:

$$\dot{\theta}_d(t) = \frac{\ddot{y}_d(t)\dot{x}_d(t) - \ddot{x}_d(t)\dot{y}_d(t)}{v_d^2(t)} \tag{3.25}$$

Plugging this into Eq. 3.21 provides the desired steering angle:

$$\phi_d(t) = \arctan\left(\frac{l[\ddot{y}_d(t)\dot{x}_d(t) - \ddot{x}_d(t)\dot{y}_d(t)]}{v_d^3(t)}\right) \tag{3.26}$$

which takes values in $(-\pi/2, \pi/2)$. Finally, differentiating Eq. 3.26 and substituting the result in Eq. 3.22 yields the second input:

$$\dot{\phi}_d(t) = \omega_d(t) = l v_d \frac{(\dddot{y}_d\dot{x}_d - \dddot{x}_d\dot{y}_d)v_d^2 - 3(\ddot{y}_d\dot{x}_d - \ddot{x}_d\dot{y}_d)(\dot{x}_d\ddot{x}_d + \dot{y}_d\ddot{y}_d)}{v_d^6 + l^2(\ddot{y}_d\dot{x}_d - \ddot{x}_d\dot{y}_d)^2} \tag{3.27}$$

where the time dependence is dropped for compactness in the right hand side. Eqs. 3.23-3.27 provide the unique state and input trajectory needed to reproduce the desired output trajectory. These expressions depend only on the values of the output trajectory (Eq. 3.18) and its derivatives up to the third order.

Chapter 4

IO feedback linearization control

The input-output linearization control is an analytical design and model-based approach that aims to reduce the original nonlinear problem to a simpler linear control problem. This type of control does not require a persistent trajectory, but the orientation is left uncontrolled.

4.1 Input-output linearization via static feedback

Naturally, for a car-like model, the output choice y' for a trajectory tracking task is:

$$y' = \begin{bmatrix} x \\ y \end{bmatrix} \tag{4.1}$$

Please note that the term y' on the left hand side of the equation is the output, while the term y on the right hand side is a coordinate of the middle point of the rear axle.
The linearization algorithm begins by computing the time derivatives of these outputs:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 \\ \sin\theta & 0 \end{bmatrix} \begin{bmatrix} v \\ \omega \end{bmatrix} = T(\theta) \begin{bmatrix} v \\ \omega \end{bmatrix} \tag{4.2}$$

Where $T(\theta)$ is the actual decoupling matrix of the system. Since this matrix is singular, static feedback fails to solve the input-output linearization and decoupling problem.

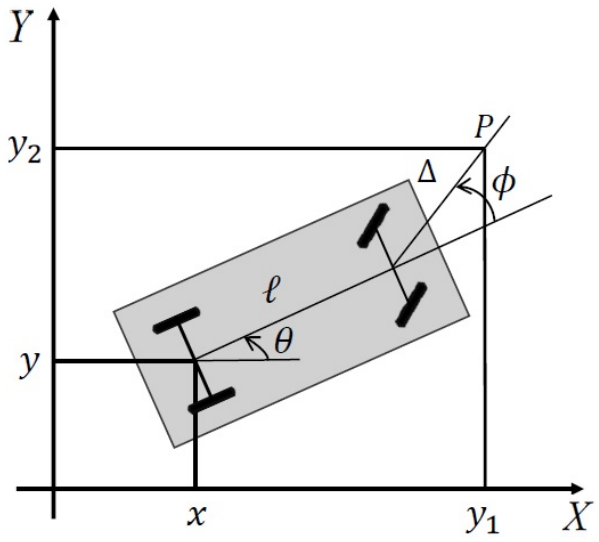


Figure 4.1: Alternative output definition for a car-like robot

It is possible to override the problem by changing the system output considering a point P located at a certain distance $\Delta \neq 0$ from the midpoint of the rear axle like in figure 4.1:

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} x + l\cos\theta + \Delta\cos(\theta + \phi) \\ y + l\sin\theta + \Delta\sin(\theta + \phi) \end{bmatrix} \tag{4.3}$$

Differentiation of this new output gives:

$$\begin{bmatrix}\dot{y}_1\\ \dot{y}_2\end{bmatrix} = \begin{bmatrix}\cos\theta - \tan\phi(\sin\theta + \Delta\sin(\theta+\phi)/l) & -\Delta\sin(\theta+\phi)\\ \sin\theta + \tan\phi(\cos\theta + \Delta\cos(\theta+\phi)/l) & \Delta\cos(\theta+\phi)\end{bmatrix}\begin{bmatrix}v\\ \omega\end{bmatrix} = T(\theta,\phi)\begin{bmatrix}v\\ \omega\end{bmatrix} \tag{4.4}$$

Since $\det T(\theta,\phi) = \Delta/\cos\phi \neq 0$, we can set $\dot{y} = u$ (an auxiliary input value) and solve for the inputs v and ω as:

$$\begin{bmatrix}v\\ \omega\end{bmatrix} = T^{-1}(\theta,\phi)\begin{bmatrix}u_1\\ u_2\end{bmatrix} \tag{4.5}$$

In the globally defined transformed coordinates (y_1, y_2, θ, ϕ), the closed-loop system becomes:

$$\begin{aligned}\dot{y}_1 &= u_1\\ \dot{y}_2 &= u_2\\ \dot{\theta} &= \frac{\sin\phi[\cos(\theta+\phi)u_1 + \sin(\theta+\phi)u_2]}{l}\\ \dot{\phi} &= -\left[\frac{\cos(\theta+\phi)\sin\phi}{l} + \frac{\sin(\theta+\phi)}{\Delta}\right]u_1 - \left[\frac{\sin(\theta+\phi)\sin\phi}{l} - \frac{\cos(\theta+\phi)}{\Delta}\right]u_2\end{aligned} \tag{4.6}$$

which is input-output linear and decoupled (one integrator on each channel). In order to solve the trajectory tracking problem, the following simple controller can be designed:

$$u_i = \dot{y}_{id} + k_i(y_{id} - y_i), \quad k_i > 0, \quad i = 1,2 \tag{4.7}$$

obtaining exponential convergence of the output tracking error to zero, for any initial condition $(y_1(t_0), y_2(t_0), \theta(t_0), \phi(t_0))$. A complete analysis would require the study of the stability of the time-varying closed-loop system (Eqs. 4.6), with u given by Eq. 4.7. In practice, we are interested in the boundedness of θ and ϕ along the nominal output trajectory.

4.2 Controller implementation

The controller has been implemented in the SIMULINK file ***'IO_fl_controller'***. To switch between the three different approaches it is necessary to uncomment the lines related to the desired one inside the section: "Model Properties" > "Callbacks" > *InitFcn*. A simulation period of $600s$ is suitable for our purpose. As first thing the wheelbase length l and the Δ distance have been defined. The first one is present inside the datasheet and it is equal to $0.85m$ while the second is arbitrary and it has been set equal to $0.28m$ (total length of the robot - $l/2$). Those two parameters together with the desired reference states and the current states are the inputs of the two blocks *Computation desired outputs* and *Computation outputs*; the first one gives, as the names suggest, the two desired outputs y_{1d} and y_{2d} while the second returns the coordinates of the point P y_1 and y_2. Then, those outputs enter inside the block of the IO controller whose gains have been defined, after a trial and error approach, as $k_1 = k_2 = 2$ for RRT* and $k_1 = k_2 = 40$ for both A* and Dijkstra. The controller provides the virtual control inputs u_1 and u_2 which, along with θ, ϕ, l and Δ are used by the block *Computation velocities* to compute the linear and angular velocities. To add an higher level of safety, in addition to the safeguards implemented inside the MATLAB code, two saturation blocks for the velocities have been added. The premade block *Ackermann Kinematic Model* has been emploied to simulate the kinematic model of our robot. The block accepts vehicle speed and steering angle angular velocity, which are both scalar values. The block outputs a four-element state vector of xy-positions, heading, and steering angle, $\begin{bmatrix}x & y & \theta & \phi\end{bmatrix}$. The vehicle position and heading angle are defined at the center of the rear axle. Those outputs, which are the current states, are fedback to the *Computation outputs* block to obtain y_1 and y_2. To see the results a series of plots have been carried out.

Chapter 5

Simulations and results

In this chapter the results obtained by the simulations of the three approaches are discussed. Before starting, it is useful to know that for each approach a ***'main.m'*** script has been created. These files:

- load the map;
- define the start [30 120] and goal point [400 135];
- set the parameters for the algorithms;
- recall their specific planning script and the trajectory creation.

5.1 RRT*

Matlab

The following figure 5.1 represents the results of the simulation obtained by using the RRT* path planning algorithm with pseudo-clothoid curves.

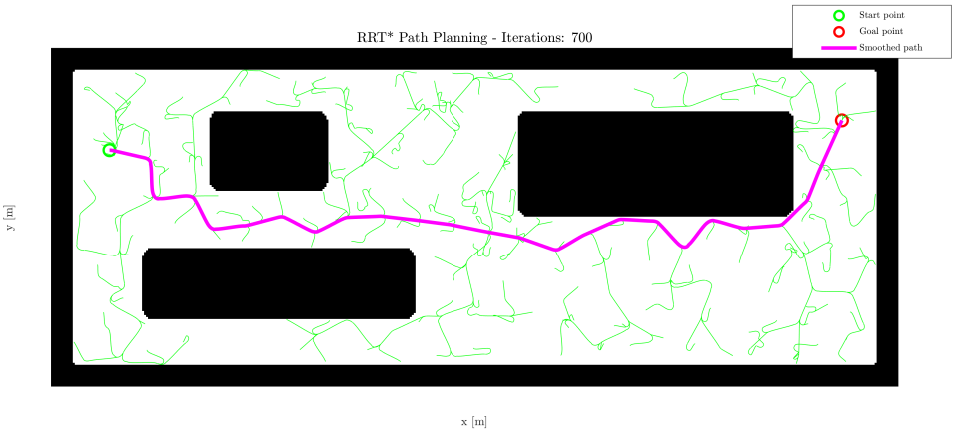


Figure 5.1: RRT* with pseudo-clothoid curves

We have a start point marked with the green circle and a goal point marked with the red circle. The algorithm, after the max number of iterations (700 in this case), creates the best path that has been marked with the magenta line. In addition, the results of the trajectory planning are shown:

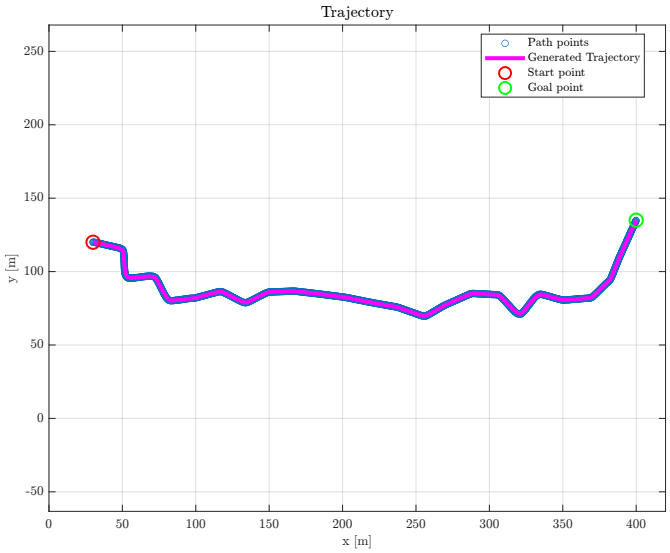
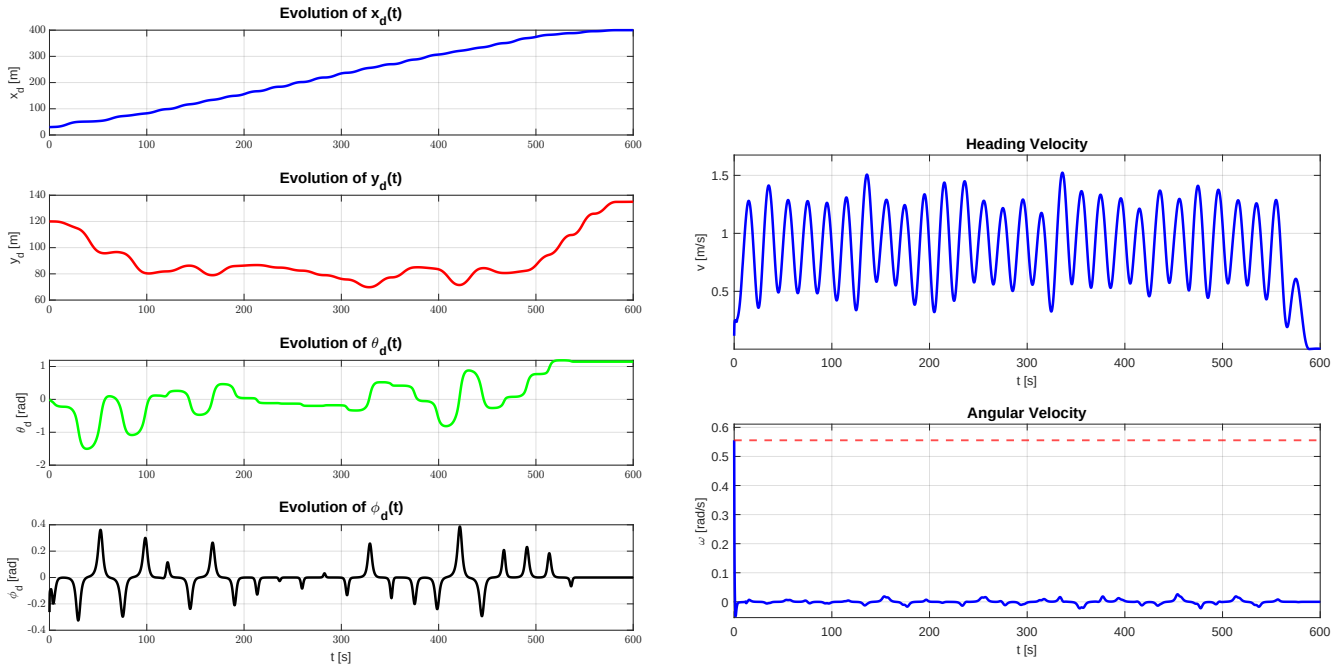


Figure 5.2: reference trajectory

As it is possible to see, the trajectory follows in a good way the points of the path generated by the previous step. The evolution of the desired states and the two velocities v (heading) and ω (angular) are reported:



(a) Evolution of states

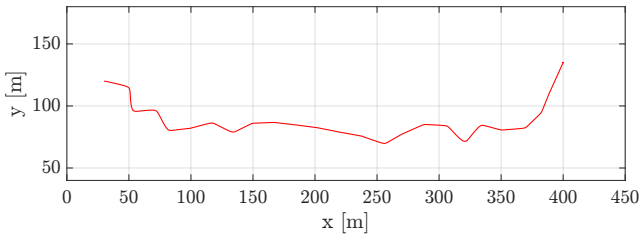
(b) Velocities

Figure 5.3: Obtained values

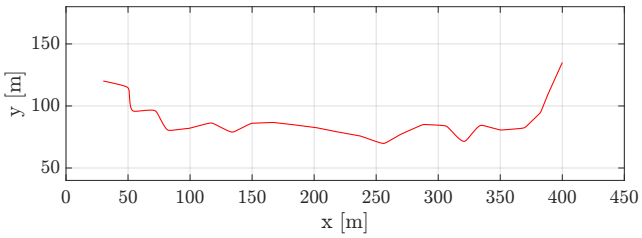
From the plot of the angular velocity it is noted that the safeguards have been implemented correctly and the angular velocity stays confined in its bound ($0.56 rad/s$).

Simulink

Now the results obtained in SIMULINK for this approach are shown. From the figure 5.4 it is clear that the trajectory carried out is correct, because on the right is reported the desired trajectory obtained through the references of trajectory given in input to the controller, instead, on the left the plot reports the trajectory that the car-like robot performs.



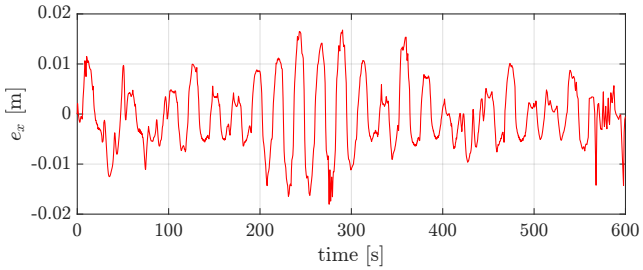
(a) Performed trajectory



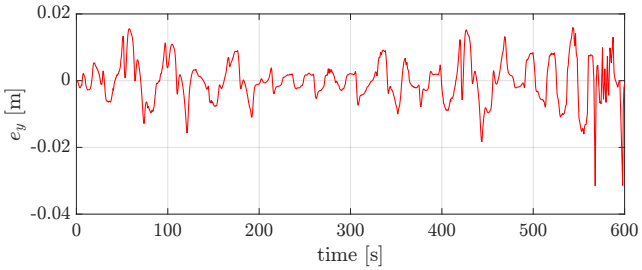
(b) Desired trajectory

Figure 5.4: Trajectories

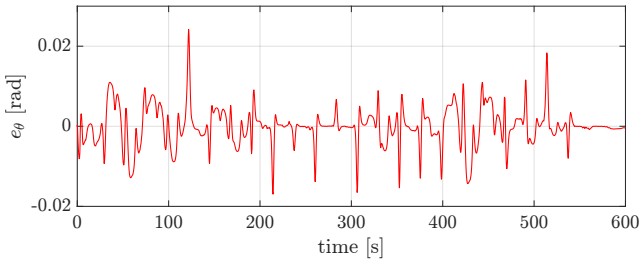
Instead below in the figure 5.5 the errors are reported. The first two plot show the position errors, which are both in the order of 10^{-2}, while on the bottom the orientation errors are reported. Since the 'Input-Output Linearization' controller left the orientation uncontrolled, as it has already explained in the chapter 4, an error of the magnitude of 10^{-1} on ϕ occurs, but for our purposes it should not alarm.



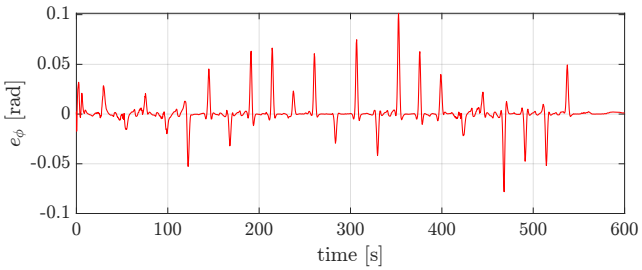
(a) Error x



(b) Error y



(c) Error theta



(d) Error phi

Figure 5.5: Errors

At last, the velocities are reported in figure 5.6, which compared to those obtained in MATLAB (reported in the figure 5.3b) assume a similar behaviour and above all do not exceed the imposed limits.

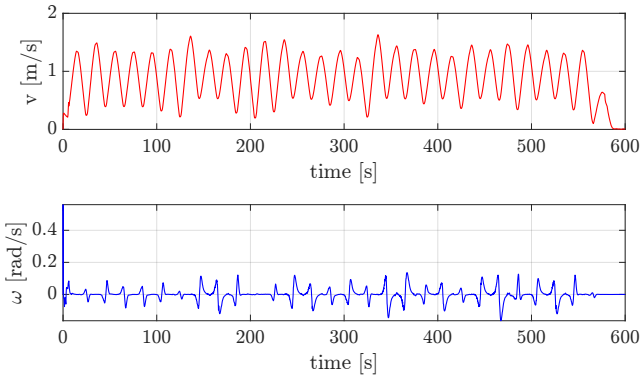


Figure 5.6: Velocities

5.2 Dijkstra

Matlab

In the following figure 5.7, the nodes of the graph are represented in blue and the curves connected them in black, and in the same way as RRT*, the path is the magenta line, the starting point is the green circle and the end point is the red circle. In addition, the smooth forced connection in the last part of the path is shown.

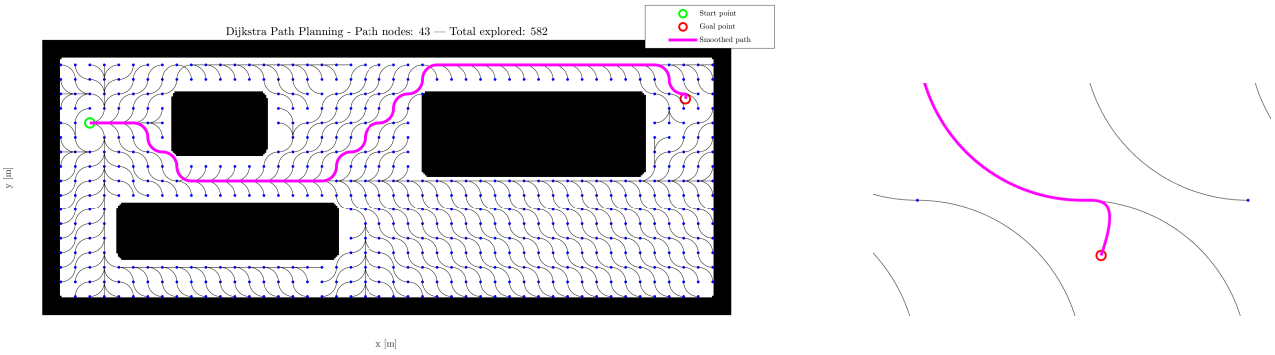


Figure 5.7: Dijkstra algorithm with RSC and smooth forced connection

The results of the trajectory planning are shown in figure 5.8. Also in this case the trajectory follows almost perfectly the points of the path.

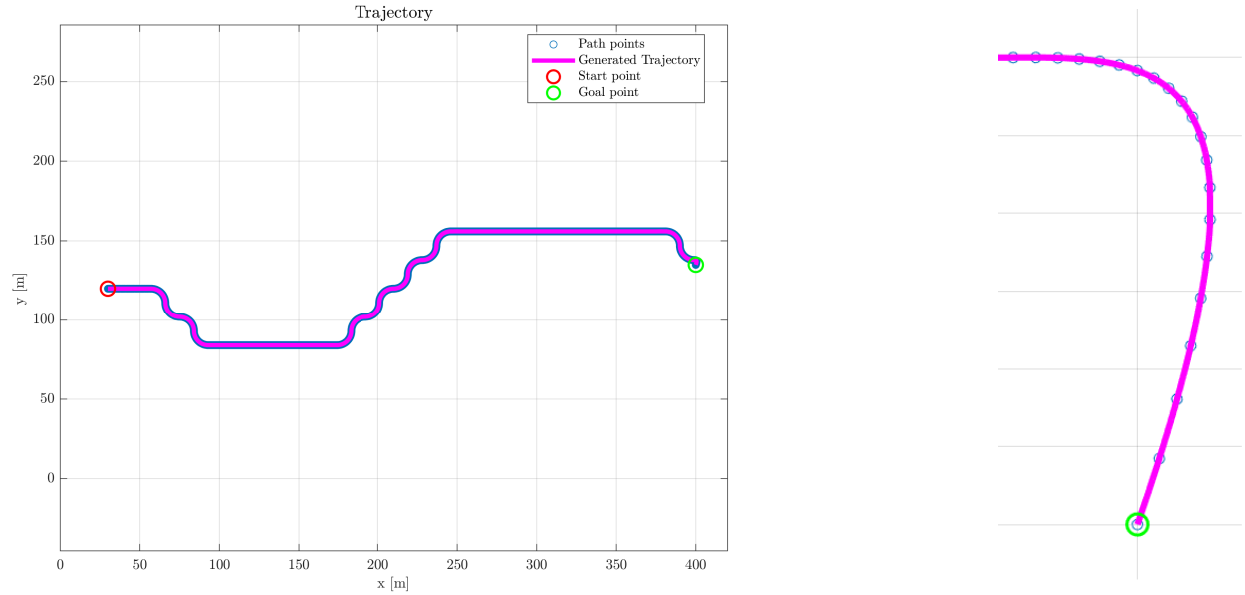
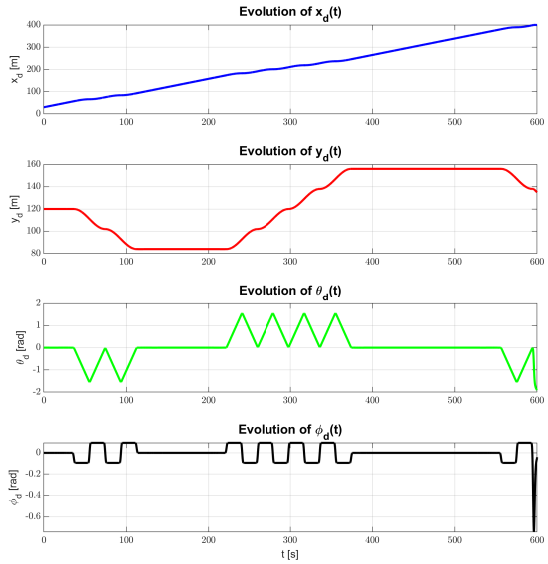
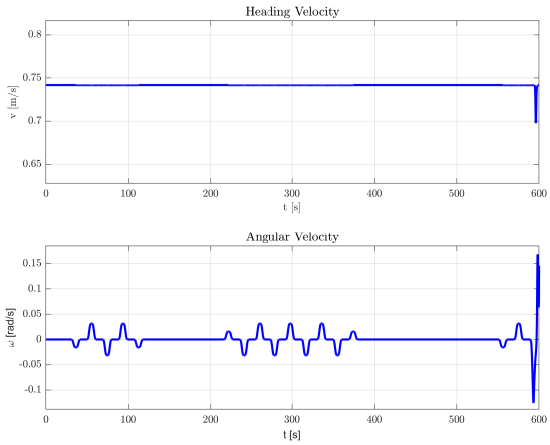


Figure 5.8: Reference trajectory

Same as the previous section in the figure 5.9 the evolution of the desired states is reported on the left, while on the right there are the velocities. As it is possible to see in the last part of the simulation there is a huge variation of ϕ_d, which almost reaches its maximum (set to $\pi/4$ inside the file *'des_trajectory.m'*). This is due to the fact that the smooth forced connection in this case has a sharp turn. As consequence the heading velocity naturally tends to decrease, while the angular velocity follows exactly the steering angle variation resulting in spikes which are still fully contained inside the boundaries.



(a) Evolution of states

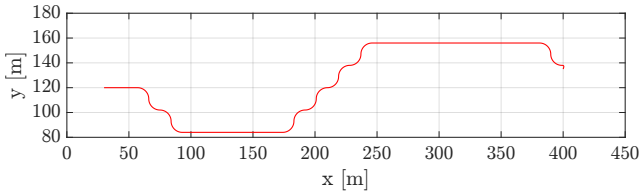


(b) Velocities

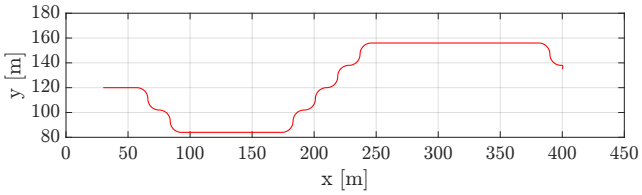
Figure 5.9: Obtained values

Simulink

Now the results obtained in SIMULINK for this approach are shown. From the figure 5.10, it is possible to see that also in this case the trajectory carried out faithfully follows the desired one.



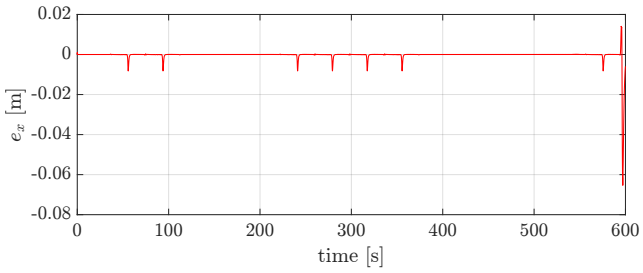
(a) Performed trajectory



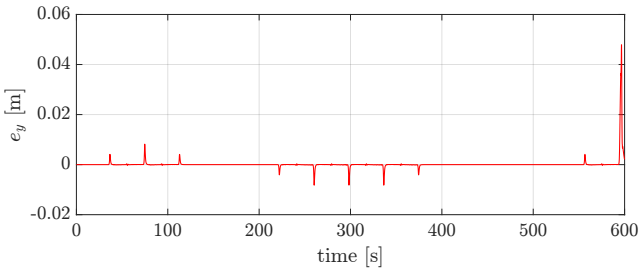
(b) Desired trajectory

Figure 5.10: Trajectories

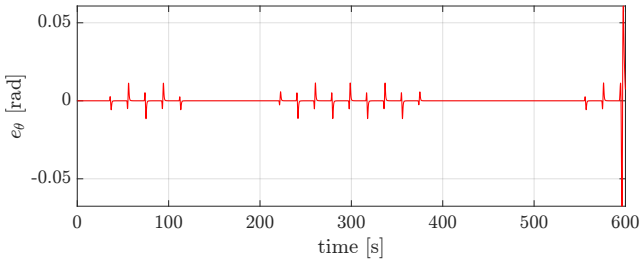
In the figure 5.11 the errors are reported. The first two plot show the position errors which for the whole path are on the magnitude of 10^{-3} while for the final turn reach a spike that brings them to the magnitude of 10^{-2}. Nevertheless it is noted that those spikes are fastly recovered when the goal is reached. The same scenario is present also for the orientation errors with a clarification: in this case the error on ϕ from a magnitude of 10^{-2} passes to a magnitude of 10^{-1}, but it is still recovered when the goal is reached.



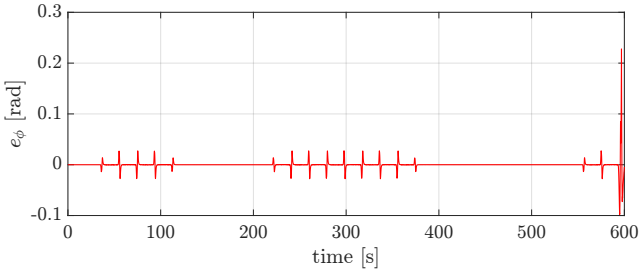
(a) Error x



(b) Error y



(c) Error theta



(d) Error phi

Figure 5.11: Errors

At last, the velocities are reported in figure 5.12, which compared to those obtained in MATLAB (reported in the figure 5.9b) assume a similar behaviour and above all do not exceed the imposed limits.

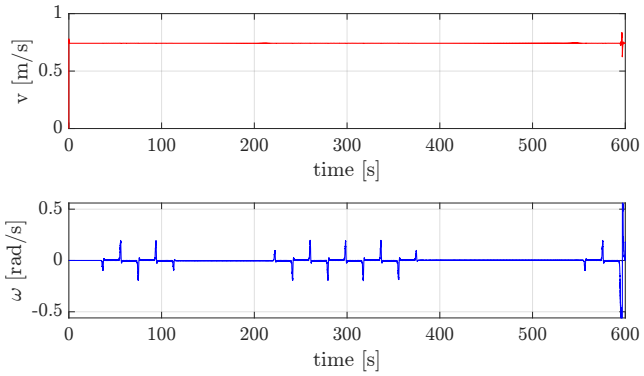


Figure 5.12: Velocities

5.3 A*

Matlab

For A* two figures are provided, the figure 5.13 shows the path obtained by using the euclidean heuristic while the figure 5.14 instead shows the results with the manhattan heuristic. Still, in both the figures, the nodes of the graph are represented in blue and the curves connected them in black, and in the same way as the first two approaches, the path is the magenta line, the starting point is the green circle and the end point is the red circle. In addition, the smooth forced connection in the last part of the path is shown.

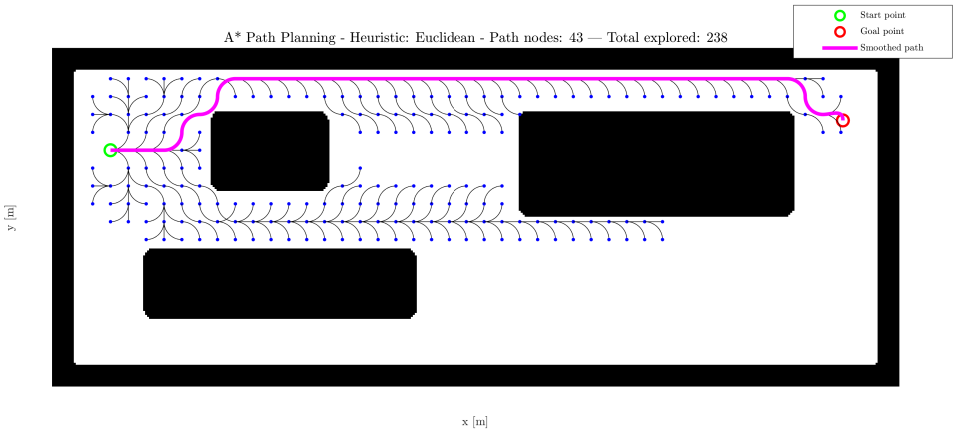


Figure 5.13: A* with RSC and euclidean heuristic

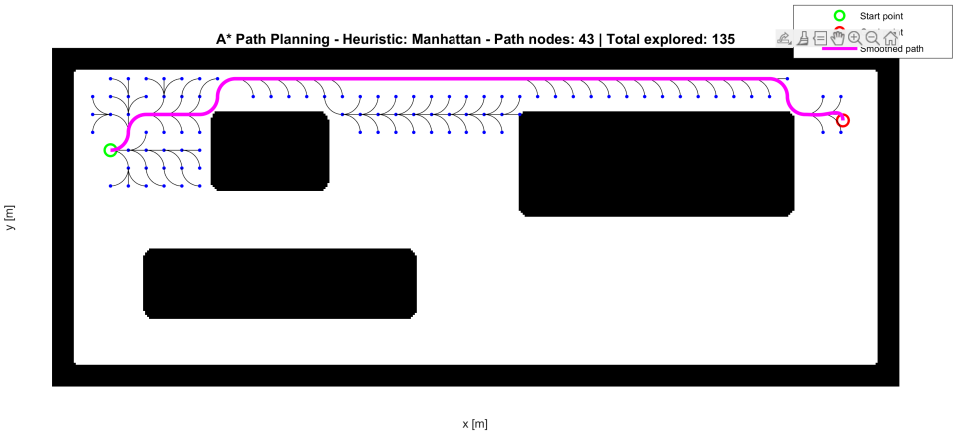


Figure 5.14: A* with RSC and manhattan heuristic

For both the results the forced smooth connection is the same and it is reported in the following figure.

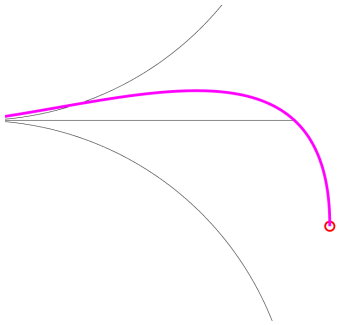


Figure 5.15: A* forced connection

In this case the magenta path deviates from the black reference path before the end because the algorithm avoids creating a cusp that would occur if it directly connected the last point of the path to the node closest to the goal. The results of the trajectory planning are shown in figure 5.8. Also in this case the trajectory follows almost perfectly the points of the path. Due to the space limitations only the results with the manhattan heuristic will be shown. In the figure 5.16 the desired trajectory is reported, while in the figure 5.17 are illustrated the evolution of the states and the velocities.

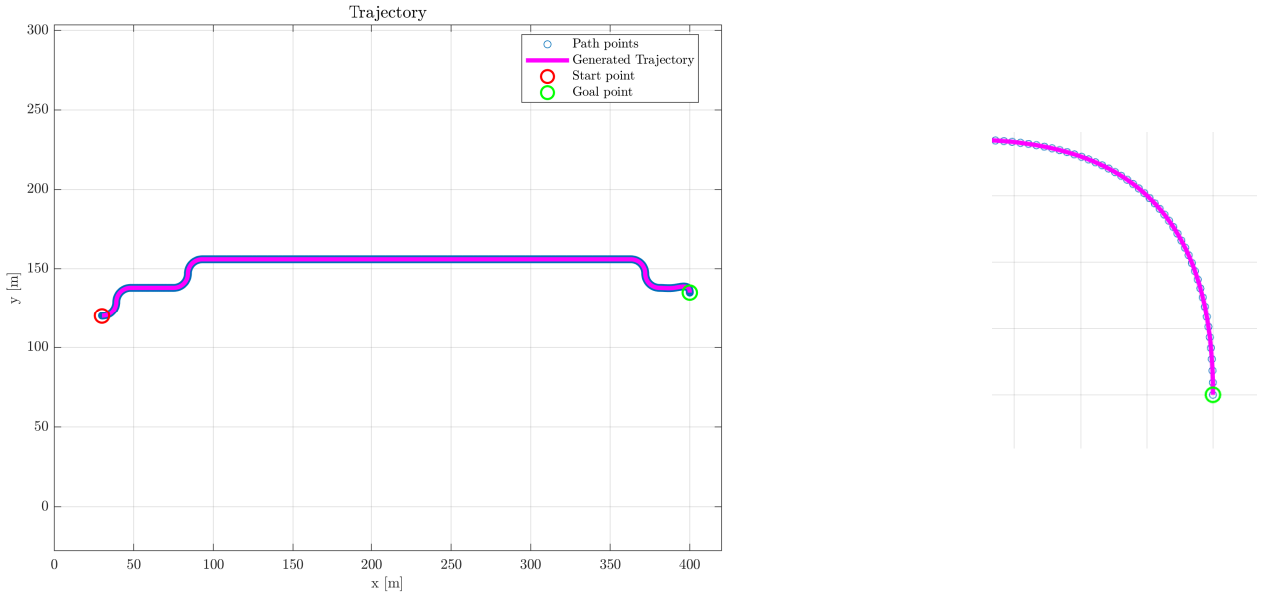
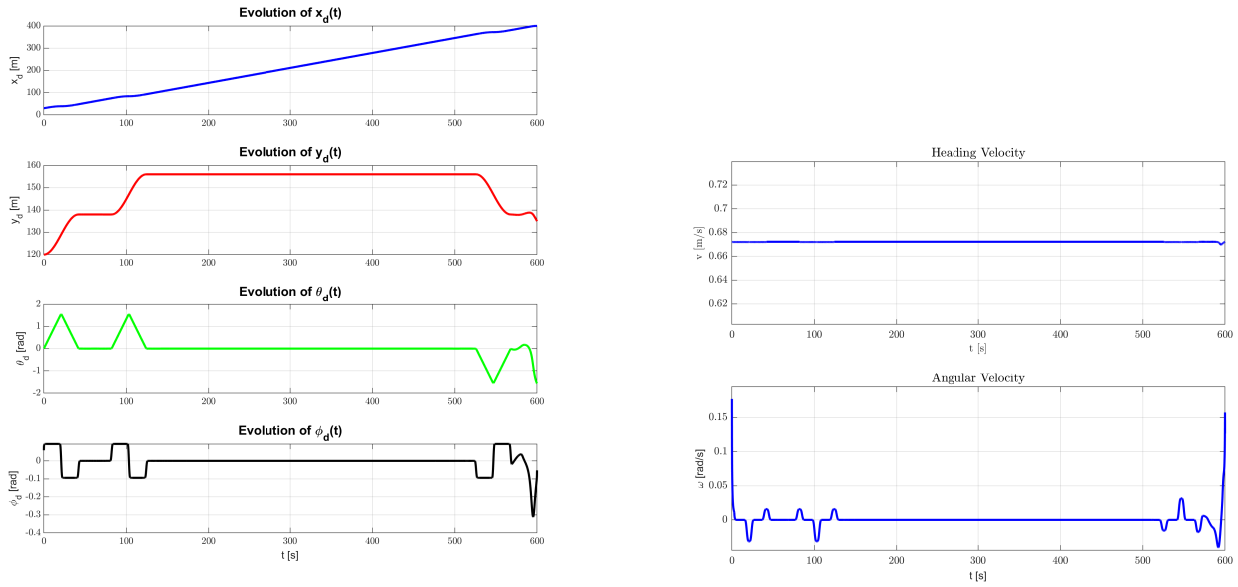


Figure 5.16: Reference trajectory

Like for Dijkstra, also here there is a peak on the angular velocity caused by the rapid variation of ϕ but now the final turn is less sharp than the previous case.



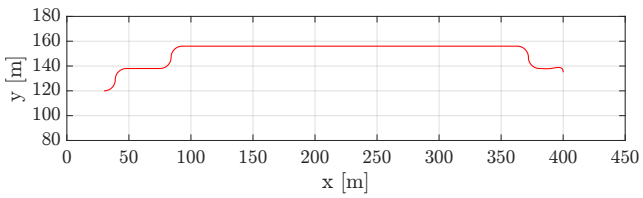
(a) Evolution of states

(b) Velocities

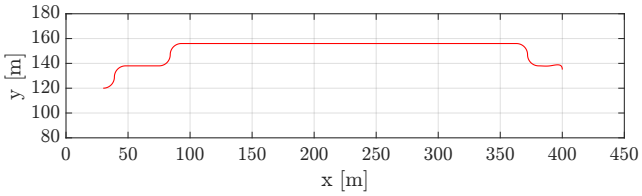
Figure 5.17: Obtained values

Simulink

The performed and the desired trajectories are illustrated in the figure 5.18.



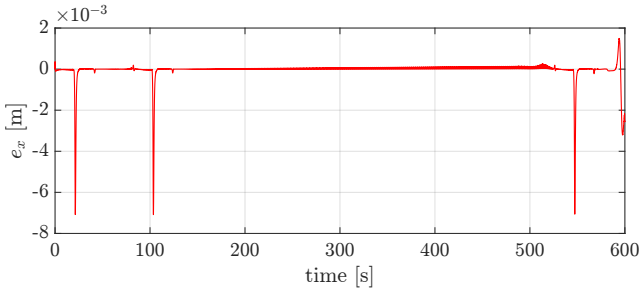
(a) Performed trajectory



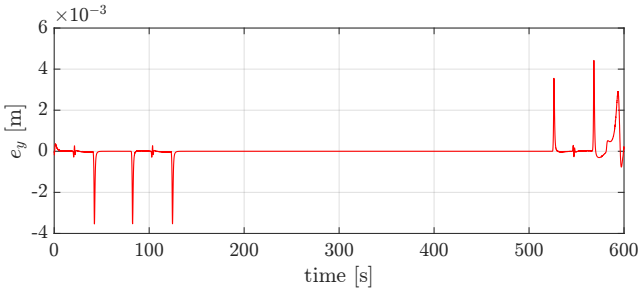
(b) Desired trajectory

Figure 5.18: Trajectories

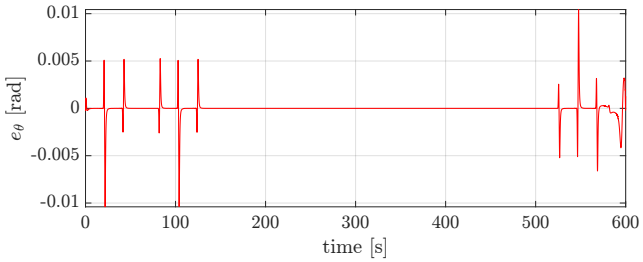
In the figure 5.19 the errors are reported. In this case it is noted that there are no increments of the magnitude in the last part of the trajectory.



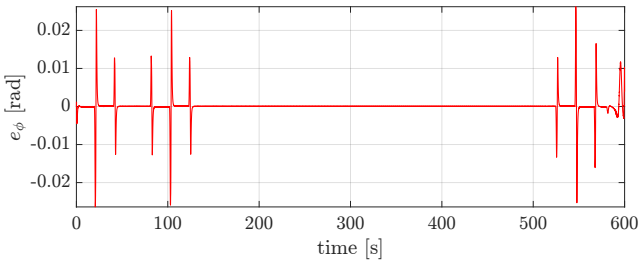
(a) Error x



(b) Error y



(c) Error theta



(d) Error phi

Figure 5.19: Errors

Lastly, the plots of velocities are reported in the following figure.

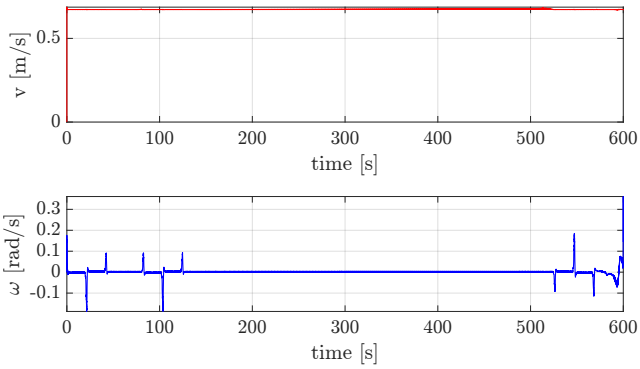


Figure 5.20: Velocities

5.4 Exploration comparison between A* and Dijkstra

In this section, we compare the exploration efficiency between A* (with both Euclidean and Manhattan heuristics) and Dijkstra's algorithm. The analysis focuses on the number of nodes explored and the resulting path quality. The code has been produced inside the file ***'comparison_Astar_Djikstra.m'***.

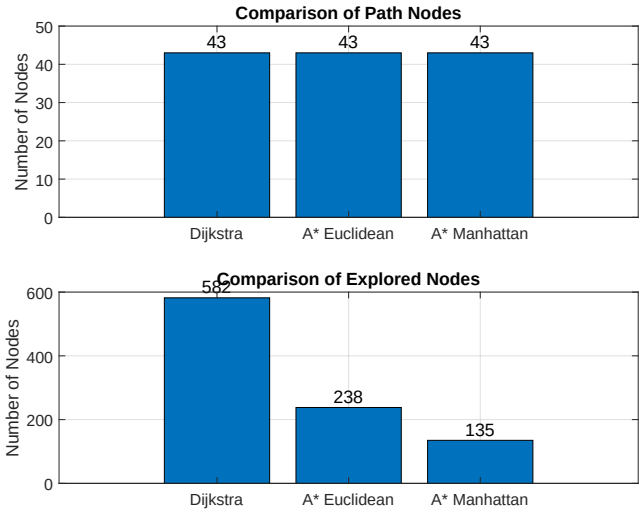


Figure 5.21: Comparisons of the path nodes

As shown in Figure 5.21, while all three approaches generate paths with a similar number of nodes, they differ significantly in their exploration efficiency. In particular, A* with the manhattan heuristic outclasses the other two strategies. Also the ratio of the exploration is provided in the figure 5.22.

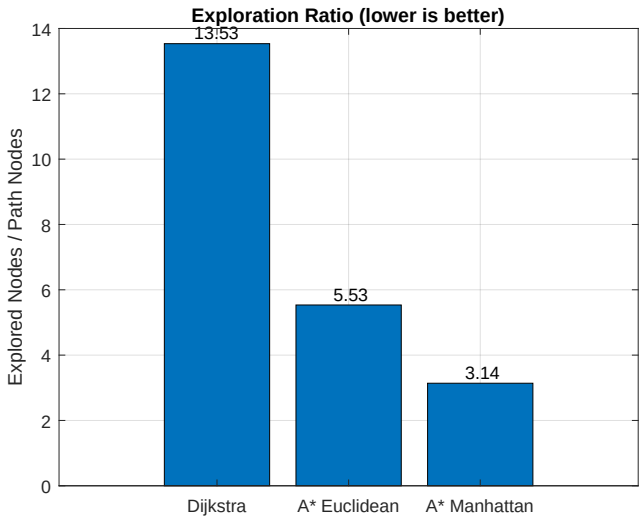


Figure 5.22: Exploration ratio

5.5 Final considerations

Sampling-based and search-based planners each have their strengths and weaknesses, which make them suitable for different types of motion planning tasks.

- **Complexity**: Sampling-based planners are well-suited for high-dimensional spaces and complex environments, as they can efficiently explore the space without requiring an explicit representation of the environment. In contrast, search-based planners may struggle in high-dimensional spaces due to the "curse of dimensionality" and increased computational complexity.
- **Optimality**: Search-based planners generally provide optimal paths within their discretized representation of the space, while RRT* is asymptotically optimal, meaning it converges to the optimal solution given sufficient iterations.
- **Convergence**: Basic sampling-based planners, like RRT, can quickly generate approximate solutions, while search-based planners may require more time to find an optimal solution, especially in high-dimensional spaces or complex environments. RRT* takes the exploratory efficiency typical of sampling-based methods but integrates optimization mechanisms that allow to converge toward optimal solutions. The convergence time of RRT* is nevertheless longer than basic RRT, while still generally remaining more efficient than complete search-based methods in complex environments. In conclusion, both sampling-based and search-based planners offer unique approaches to robot motion planning, with each being well-suited for different types of tasks and environments. Sampling-based planners excel in high-dimensional spaces and complex environments, efficiently exploring the configuration space without requiring an explicit representation of the environment. On the other hand, search-based planners focus on finding optimal paths by leveraging graph-based search algorithms in discretized representations of the configuration space. This distinction, however, is no longer as sharp due to the development of hybrid algorithms that incorporate the strengths of both approaches, offering a balance between rapid space exploration and optimality guarantees.

The best choice to take advantage of both approaches would be one that uses a sampling-based planner for the first part of the exploration, since we need a fast exploration, and in proximity of the goal, switches to a search-based approach to find the best path possible.

Bibliography

[1] A. De Luca, G. Oriolo, and C. Samson, "Feedback Control of a non-holonomic Car-like Robot," in *Robot Motion Planning and Control*, J.-P. Laumond, Ed. Berlin, Heidelberg: Springer, 1998, ch. 4, pp. 171–253.

[2] H. Kwon, D. Cha, J. Seong, J. Lee, and W. Chung, "Trajectory Planner CDT-RRT* for Car-Like Mobile Robots toward Narrow and Cluttered Environments," *Sensors*, vol. 20, no. 5, pp. 1489, 2020.

[3] N. Ma, J. Wang, and M. Q.-H. Meng, "Conditional Generative Adversarial Networks for Optimal Path Planning," *arXiv:2012.03166*, 2020.

[4] M. Vendittelli, J.-P. Laumond, and C. Nissoux, "Obstacle Distance for Car-Like Robots," *IEEE Transactions on Robotics and Automation*, vol. 15, no. 4, pp. 678-691, 1999.

[5] K. Ran, Y. Wang, C. Fang, Q. Chai, X. Dong, and G. Liu, "Improved RRT* Path-Planning Algorithm Based on the Clothoid Curve for a Mobile Robot Under Kinematic Constraints," *Sensors*, vol. 24, no. 23, pp. 7812, 2024.

[6] D.H. Shin, S. Singh, and W. Whittaker, "Path Generation for a Robot Vehicle Using Composite Clothoid Segments," *Intelligent Components and Instruments for Control Applications*, pp. 443-448, 1992.

[7] S. G. Manyam, D. Casbeer, A. L. Von Moll, and Z. Fuchs, "Shortest Dubins Path to a Circle," *AIAA Guidance, Navigation, and Control Conference*, pp. 1–14, 2019.

[8] L. E. Dubins, "On Curves of Minimal Length with a Constraint on Average Curvature, and with Prescribed Initial and Terminal Positions and Tangents," *American Journal of Mathematics*, vol. 79, no. 3, pp. 497–516, 1957.

[9] A. Bessas, A. Benalia, and F. Boudjema, "Robust Output Trajectory Tracking of Car-Like Robot Mobile," *J. Electrical Systems*, vol. 12, no. 3, pp. 541–555, 2016.

[10] T. Klasson, "Robotic Path Planning: RRT and RRT*," *Medium*, 2018. [Online]. Available: https://theclassytim.medium.com/robotic-path-planning-rrt-and-rrt-212319121378

[11] "Path Planning and Navigation," *MIT*, 2021. [Online]. Available: https://fab.cba.mit.edu/classes/865.21/topics/path-planning/robotic.html

[12] A. Patel, "A* Algorithm Implementations," *Stanford Theory*, 2021. [Online]. Available: https://theory.stanford.edu/ amitp/GameProgramming/AStarComparison.html

[13] "Manhattan Distance," *ScienceDirect*, 2024. [Online]. Available: https://www.sciencedirect.com/topics/computer-science/manhattan-distance

[14] MATLAB, "Path Planning with A* and RRT — Autonomous Navigation, Part 4," YouTube, Feb. 2022. [Online]. Disponibile: `https://www.youtube.com/watch?v=QR3U1dgc5RE`.

[15] A. Becker, "RRT, RRT* & Random Trees," YouTube, Feb. 2022. [Online]. Disponibile: `https://www.youtube.com/watch?v=Ob3BlkQJEw`.

[16] T. Zhao, “RRT* Algorithm Explained,” YouTube, Feb. 2022. [Online]. Disponibile: https://www.youtube.com/watch?v=_aqwlBx2NFk.