UNIVERSITA' 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

Elaborato Field and Service Robotics

Professore:
Fabio Ruggiero

Candidati:
Paolo Maisto P38000191
Angelo Vittozzi P38000184

Anno Accademico 2023/2024

Contents

Introduction

In this project, the aim is to apply one of the controllers studied in the course to a car-like robot in order to create and follow a path from a starting point to an end point, with the goal of creating a simulation that closely mirrors real-world conditions. To achieve this, the simulation is conducted on maps obtained from Google Maps. These maps are converted into black-and-white images, where black represents obstacles and white represents the traversable path. These images are then further processed into binary maps suitable for use in MATLAB.

The chosen controller for this project is the **Input-Output Linearization** controller. To ensure the completeness of the project, this controller has as input a reference trajectory. This reference trajectory is generated using two different methods:

- The first method uses the **Probabilistic Roadmap** (PRM) algorithm to create the graph, followed by the **Breadth-First Search** (BFS) algorithm to calculate the optimal path.

- The second method, instead, involves using the **Reeds-Shepp curves** (RSC) algorithm to generate a graph, followed by the **Dijkstra** algorithm to find the optimal path.

Additionally, in the first case the path is smoothed to avoid sharp turns, as the car-like robot cannot turn in place unlike a unicycle model. The path generation is implemented in MATLAB, while the controller is implemented in SIMULINK.

This introduction outlines the main idea behind the project, including the application of the controller to a car-like robot, the use of real-world maps for simulation, and the generation and optimization of reference trajectories.

Chapter 1

Modeling and analysis of the car-like robot

In this chapter is explained the kinematic model taken into account for the car-like robot, used inside the controller, and then the fundamental properties of the corresponding system are analysed from a control viewpoint.

1.1 Kinematic modeling

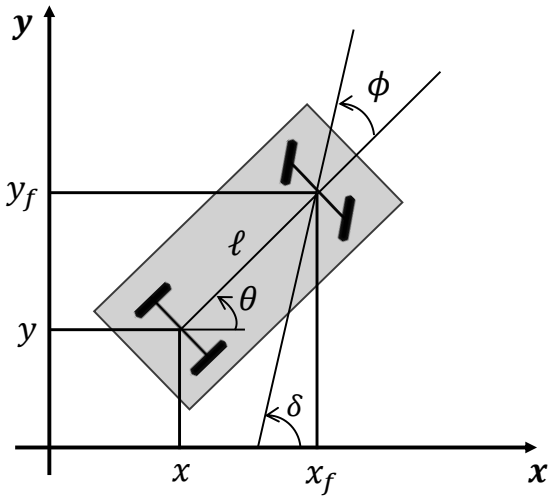


Figure 1.1: Generalized coordinates of a car-like robot

The main feature of the kinematic model of wheeled mobile robots is the presence of nonholonomic constraints due to the *rolling without slipping* condition between the wheels

and the ground. Analyzing now a robot having the same kinematics of a car, as shown in Fig.1.1.

For 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 (*car-like* model). The front wheel can be steered while the rear wheel orientation is fixed. The generalized coordinates are $q = [x \quad y \quad \theta \quad \phi]^T$, 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 nonholonomic constraints, one for each wheel:

$$\begin{aligned} \dot{x}\sin\theta - \dot{y}\cos\theta &= 0 \\ \dot{x}_f \sin(\theta+\phi) - \dot{y}_f \cos(\theta+\phi) &= 0 \end{aligned} \tag{1.1}$$

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

$$\begin{aligned} x_f &= x + \ell\cos\theta \\ y_f &= y + \ell\sin\theta, \end{aligned} \tag{1.2}$$

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

$$\dot{x}\sin(\theta+\phi) - \dot{y}\cos(\theta+\phi) - \dot{\theta}\ell\cos\phi = 0 \tag{1.3}$$

The Pfaffian constraint matrix is

$$A^T(q) = \begin{bmatrix} \sin\theta & -\cos\theta & 0 & 0 \\ \sin(\theta+\phi) & -\cos(\theta+\phi) & -\ell\cos\phi & 0 \end{bmatrix} \tag{1.4}$$

and has constant rank equal to 2.

If the car has *rear-wheel driving*, the kinematic model is derived as

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \\ \dot{\phi} \end{bmatrix} = \begin{bmatrix} \cos\theta \\ \sin\theta \\ \tan\phi/\ell \\ 0 \end{bmatrix} v + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \omega, \tag{1.5}$$

where v and ω are the *driving* and the *steering* velocity input, respectively. There is a model singularity at $\phi = \pm\pi/2$, where the first vector field has a discontinuity. This corresponds to the car becoming jammed when the front wheel is normal to the longitudinal axis of the body. However, the importance of this singularity is limited, due to the restricted range of the steering angle ϕ in most practical cases.

It has to be mentioned that a more complete kinematic model should include also the rotation angles of each wheel expressed as generalized coordinates, in order to account for the presence of actuators and sensors on the wheels axis as well as for slip phenomena occurring at the wheel-ground contact. Nevertheless, this model captures the essence of the vehicle kinematics and is well suited for control purposes.

Chapter 2

Motion planning

In this chapter the approaches that lead to the creation of the reference trajectories have been treated, which are given in input to the controller and that the car-like robot must chase.

It should be pointed out that since everything is based on finding the route on a real map, obtained as a binary image, it is a matrix composed of 0 and 1 in which the 0 express obstacles while 1 the space to be walked. Then at the end the path that is obtained is given by the nodes that are expressed as cells of this matrix whose coordinates are constituted by the row and the column. Consequently giving these coordinates in SIMULINK the path is obtained with y inverse, then to obtain the reference trajectory the y coordinate has been scaled, by applying the following formula:

$$y_{ref} = size(map, 1) - y$$

in which $size(map, 1)$ expresses the number of rows of the map, while the y the row where the node is located.

2.1 First Approach

In the first approach through the use of the algorithm PRM, given the initial and final point, a graph is obtained which creates the various connections that allow to reach the goal. Then the BFS algorithm is used to calculate the path from start point to end point.

But once the path was obtained, since the car-like robot is not able to rotate on itself, the path is made smooth in order to eliminate the cusps, which can not have been covered by the robot considered. That is implemented in the MATLAB code '***PRM_BFS_smooth***'. The methods and algorithms used are explained below.

2.1.1 Probabilistic Roadmap Method

The Probabilistic Roadmap Method (PRM) algorithm is implemented as follows:

- The navigable points are calculated first, that is, it checks which areas are not obstructed, then, with a uniform probabilistic distribution, select nodes in this free zone by checking that the minimum distance between nodes are met, imposed in order to avoid creating a graph with too close nodes between them. At the end the start and end points are also added to these points;

- The various nodes are then connected to each other so that the maximum distance constraint is respected, in order to avoid nodes too far apart; before connecting them, however, the collision check is carried out and therefore they are not connected when they touch an obstacle;

- Finally, given the connections between the nodes, the adjacency matrix, used subsequently to create the path to follow, is created, since it takes into account how the nodes are connected to each other.

2.1.2 Breadth-first search (BFS)

The algorithm is implemented as follows:

First of all a queue is initialized with the index of the starting node; a *'parent'* vector that keeps track of the path followed, a *'visited'* vector that instead keeps track of the nodes visited, to avoid visiting the node multiple times.
At this point a cycle begins that ends until the queue is empty or if the current node is the arrival node, obviously starting from the initial node:

– the nodes connected using the adjacency matrix are located;

– for each neighbor not yet visited, is set as visited and updates the vector that tracks the route and adds the neighbor to the queue to be explored later;

– the loop is repeated by extracting the first node from the queue (FIFO queue management);

At the end, the path is not returned directly but the *'parent'* vector that allows to go up from the arrival node $(269, 128)$ to the starting node $(30, 80)$ to reconstruct the path followed.

Below the implementation of the logic behind BFS algorithm is shown:

Algorithm 1 BFS logic.

```
1:  function BFS(Graph g, Node start, Node target)
2:      Queue q
3:      Set visited
4:      q.push(start)
5:      while not q.empty() do
6:          Node current = q.front()
7:          q.pop()
8:          if current = target then
9:              return True
10:         if current not in visited then
11:             visited.insert(current)
12:             for each neighbor in g.neighbors(current) do
13:                 if neighbor not in visited then
14:                     q.push(neighbor)
15:     return False
```

At the end, the path from the starting point to the end point that the robot must follow is obtained. The last remaining step is to eliminate the cusps on this path and thus make it smooth.

2.1.3 Path smoothing

The resulting path, however, consists of broken lines. The aim is to make the path smooth in order to eliminate the cusps within the path, because the car-like robot is not able to rotate on itself. To obtain a smooth path, MATLAB's `spline` function is utilized. This function allows the interpolation of the points from the original path, resulting in

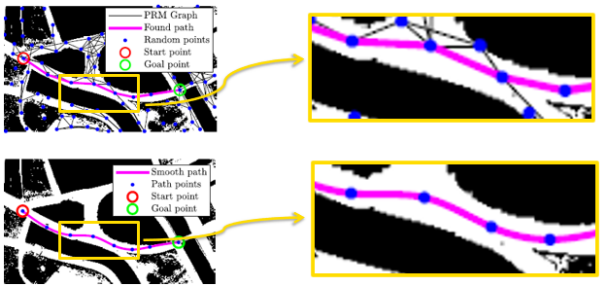


Figure 2.1: Difference between the orginal path and the smoothed path.

a continuous and differentiable trajectory. Specifically, spline interpolation is applied to the x and y points of the path, using a normalized time evolution to ensure a smooth distribution along the trajectory. The Fig. 2.1 figure illustrates the difference between the original path, composed of broken lines, and the smoothed path obtained using the spline interpolation technique. Notice that once the smooth path is obtained, a collision check is carried out with the obstacles (sometimes it can happen that the path composed of broken lines does not intersect the obstacles and the smooth path does it). If the path intersects the obstacles, another attempt is made to plan the path until a path free of obstacles is found.

2.2 Second Approach

The second approach instead, always taking into account the robot taken into consideration, the RSC method has been used, in order to avoid the process that makes the path smooth. Once obtained the graph, the path of minimum time is calculated with the Dijkstra algorithm. And finally the reference trajectory for the controller is obtained. That is implemented in the MATLAB code '***RSC_Dijkstra***'. The methods and algorithms used are explained in the next subsections.

2.2.1 Reeds-Shepp curves

This algorithm is used because it offer a robust solution for the non-holonomic robots. Their structured approach to combining elementary motion segments ensures that the generated paths are feasible especially for robots such as car-like robots that are unable

to turn on themselves, so they need to follow a path without cusps. So, in this approach, it is assumed that the robot can only follow curves named in the Fig.2.2 as: $C^+_{a,l}$ and $C^+_{a,r}$ for circular arcs, S^+_d for linear segments. As a result, the robot can move only slightly forward.

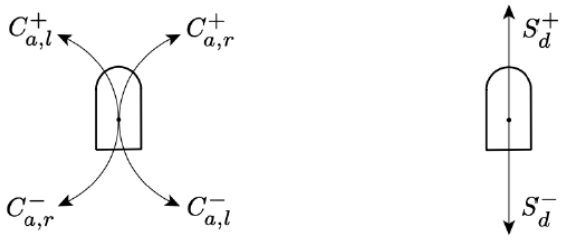


Figure 2.2: Elementary arcs.

Where:

$$v_{max}(t) = 3\,m/s, \quad \omega_{max}(t) = 0.43\,rad/s$$

So the radius of these circular arcs is determined by $\frac{v_{max}}{\omega_{max}} = 7\,m$.

And the leght of the linear segments is $7\,m$.

The algorithm has been implemented as follows:

- Starts from the starting point $(37, 80)$;
- The next 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 exceeded the node is added to a vector;
- Then the iteration repeats to the next node following the order in which they are inserted into the array until there are no more nodes to insert.

As it progresses creates the adjacency matrix that takes into account how nodes are connected to each other, useful in the next step for creating the path. Between the various nodes that are part of the graph there is also the final node $(275, 143)$.

At this point obtained the graph that connects the various nodes including the initial and final point remains only to find the optimal path that allows to go from the initial

point to the end point.

2.2.2 Dijkstra's Algorithm

In order to find the path that takes the least time possible, the Dijkstra's algorithm is implemented as follows:

- starts with the starting node defined as *'current'*;
- find the *'neighbor'* nodes using the adjacency matrix, that is the nodes that are directly connected to current;
- for each neighbor calculate how much it would cost to get there by passing through current, if it finds that getting to a neighbor by passing through current is faster than any route that has been found so far, update the distance for that neighbor, storing it;
- from these selects the node with the minimum distance currently estimated between the nodes not visited, if it is the arrival node the cycle ends; otherwise repeats the cycle starting from the node just selected that becomes the *'current'* node;

The iterations end until the queue is empty or if the current node is the arrival node.

Once the algorithm is completed, starting from the arrival node, follow the references to predecessors (prev) until you reach the starting node, thus building the optimal path (path).

At this point, the path has been obtained that allows to go from the initial point to the final point without cusps, which takes the least time possible.

2.3 Reference trajectory generation

The generation of a feasible trajectory starts with the geometric path obtained through the motion planning algorithms. To convert this geometric path into a trajectory that the robot can follow over time, the following steps are applied:

1. The initial path is represented as a geometric path $q(s)$, where s is the arc length of the path. This path consists of a sequence of Cartesian coordinates (x, y), which represent the robot's position in space without time dependence.

2. Next, a time law $s(t)$ is defined, which maps the time t to the arc length s of the geometric path. To achieve this, a 7th-degree polynomial is fitted using MATLAB's `polyfit` function and interpolated with `polyval` functions. This time law ensures a smooth temporal distribution along the path, enabling the robot to move steadily from point to point and provides a consistent evolution of the path in time.

3. The time law $s(t)$ is used to parameterize the geometric path coordinates $x(s)$ and $y(s)$ in terms of time. Using MATLAB's `interp1` function with the `spline` option, a smooth interpolation of the coordinates is obtained. The result is a continuous and differentiable trajectory $x(t)$ and $y(t)$, ensuring that the robot's movement is smooth.

4. Finally, the desired states of the trajectory $[x_d, y_d, \theta_d, \phi_d]^T$ are computed to be fed into the robot's controller. At this stage, the linear velocity $v_d(t)$, angular velocity $\omega_d(t)$, orientation angle $\theta_d(t)$, and steering angle $\phi_d(t)$ are also calculated. These values must respect velocity limits: $v_{\max} = 3\,\mathrm{m/s}$ and $\omega_{\max} = 0.43\,\mathrm{rad/s}$.

At this stage, the trajectory is ready to be used as a reference for the control system. However, before moving to the control design, it is needed to derive state reference trajectories for the car-like robot in the original kinematic description.

From the path planning algorithms described above, a feasible and smooth desired output trajectory is given in terms of the cartesian position of the car rear wheel, i.e.,

$$x_d = x_d(t), \quad y_d = y_d(t), \quad t \geq t_0 \tag{2.1}$$

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. 2.1) is feasible

when it can be obtained from the evolution of a reference car-like robot

$$\dot{x}_d = \cos\theta_d v_d \tag{2.2}$$

$$\dot{y}_d = \sin\theta_d v_d \tag{2.3}$$

$$\dot{\theta}_d = \tan\phi_d \frac{v_d}{l} \tag{2.4}$$

$$\dot{\phi}_d = \omega_d \tag{2.5}$$

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. 2.2 and 2.2 gives for the first input

$$v_d(t) = \pm\sqrt{\dot{x}_d^2(t) + \dot{y}_d^2(t)}, \tag{2.6}$$

where the sign depends on the choice of executing the trajectory with forward or backward car motion, respectively. Dividing Eq. 2.3 by 2.2, and keeping the sign of the linear velocity input under account, the desired orientation of the car is computed as

$$\theta_d(t) = \text{ATAN2}\left\{\frac{\dot{y}_d(t)}{\dot{v}_d(t)}, \frac{\dot{x}_d(t)}{\ddot{v}_d(t)}\right\}, \tag{2.7}$$

Differentiating Eq. 2.2 and 2.3 , and combining the results so as to eliminate $\dot{v}_d$, the second input is obtained

$$\dot{\theta}_d(t) = \frac{\ddot{y}_d(t)\dot{x}_d(t) - \ddot{x}_d(t)\dot{y}_d(t)}{v_d^2(t)} \tag{2.8}$$

Plugging this into Eq. 2.4 provides the desired steering angle

$$\phi_d(t) = \arctan\left(\frac{l[\ddot{y}_d(t)\dot{x}_d(t) - \ddot{x}_d(t)\dot{y}_d(t)]}{v_d^3(t)}\right) \tag{2.9}$$

which takes values in $(-\pi/2, \pi/2)$.

Finally, differentiating Eq. 2.9 and substituting the result in Eq. 2.5 yields the second input

$$\dot{\phi}_d(t) = \omega_d(t) = \ell v_d \frac{(\dddot{y}_d\dot{x}_d - \dddot{x}_d\dot{y}_d)v_d^2 - 3(\ddot{y}_d\dot{x}_d - \ddot{x}_d\dot{y}_d)(\dot{x}_d\ddot{x}_d + \dot{y}_d\ddot{y}_d)}{v_d^6 + \ell^2(\ddot{y}_d\dot{x}_d - \ddot{x}_d\dot{y}_d)^2} \tag{2.10}$$

where the time dependence is dropped for compactness in the right hand side.

Eqs. (2.6–2.10) provide the unique state and input trajectory (modulo the choice of forward or backward motion) needed to reproduce the desired output trajectory. These expressions depend only on the values of the output trajectory (Eq. 2.1) and its derivatives up to the third order.

Chapter 3

Control via exact feedback linearization

In this chapter the static feedback controller used for trajectory tracking is presented.

3.1 Input-output linearization via static feedback

For the car-like robot model, the natural output choice y' for the trajectory tracking task is

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

where y' at the first term of equation is output while y at the second term is a coordinate of middle point of rear axle of car-like robot.

The linearization algorithm begins by computing

$$\begin{bmatrix} \dot{x} \\ \dot{y} \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 \\ \sin\theta & 0 \end{bmatrix} \begin{bmatrix} v \\ \omega \end{bmatrix} = T(\theta) \begin{bmatrix} v \\ \omega \end{bmatrix} \tag{3.1}$$

At least one input appears in both components of $\dot{y}$, so that $T(\theta)$ 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.

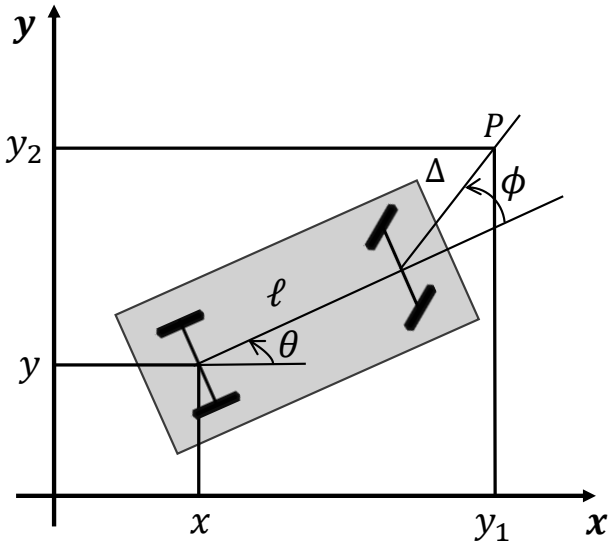


Figure 3.1: Alternative output definition for a car-like robot

A possible way to circumvent this problem is to redefine the system output as

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} x + l\cos\theta + \Delta\cos(\theta+\phi) \\ y + l\sin\theta + \Delta\sin(\theta+\phi) \end{bmatrix} \tag{3.2}$$

with $\Delta \neq 0$. This choice corresponds to selecting the representative point of the robot as P in the Fig.3.1, in place of the midpoint of the rear axle.

Differentiation of this new output gives

$$\begin{bmatrix} \dot{y_1} \\ \dot{y_2} \end{bmatrix} = \begin{bmatrix} \cos\theta - \tan\phi(\sin\theta + \Delta\sin(\theta+\phi)/l) & -\Delta\sin(\theta+\phi) \\ \sin\theta + \tan\phi(\cos\theta + \Delta\cos(\theta+\phi)/l) & \Delta\cos(\theta+\phi) \end{bmatrix} \begin{bmatrix} v \\ \omega \end{bmatrix} = T(\theta,\phi) \begin{bmatrix} v \\ \omega \end{bmatrix} \tag{3.3}$$

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

$$\begin{bmatrix} v \\ \omega \end{bmatrix} = T^{-1}(\theta,\phi) \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} \tag{3.4}$$

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

becomes

$$\begin{aligned}
\dot{y}_1 &= u_1 \\
\dot{y}_2 &= u_2 \\
\dot{\theta} &= \frac{\sin\phi\,[\cos(\theta+\phi)u_1 + \sin(\theta+\phi)u_2]}{l} \\
\dot{\phi} &= -\left[\frac{\cos(\theta+\phi)\sin\phi}{l} + \frac{\sin(\theta+\phi)}{\Delta}\right] u_1 \\
&\quad -\left[\frac{\sin(\theta+\phi)\sin\phi}{l} - \frac{\cos(\theta+\phi)}{\Delta}\right] u_2
\end{aligned} \tag{3.5}$$

which is input-output linear and decoupled (one integrator on each channel).

In order to solve the trajectory tracking problem, it was chosen the following controller:

$$u_i = \dot{y}_{id} + k_i(y_{id} - y_i), \quad k_i > 0, \quad i = 1, 2 \tag{3.6}$$

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 series of remarks is now in order.

- While the two output variables converge to their reference trajectory with arbitrary exponential rate (depending on the choice of the k_i's in eq. 3.6), the behavior of the variables θ and ϕ cannot be specified at will because it follows from the last two equations of 3.5.

- A complete analysis would require the study of the stability of the time-varying closed-loop system 3.5, with u given by eq. 3.6. In practice, one is interested in the boundedness of θ and ϕ along the nominal output trajectory. This study may not be trivial for higher-dimensional wheeled mobile robots, where the internal dynamics has dimension $n - 2$.

- Having redefined the system outputs as in eq. 3.2, one has two options for generating the reference output trajectory. The simplest choice is to directly plan a cartesian motion to be executed by the point P. On the other hand, if the planner generates a desired motion $x_d(t), y_d(t)$ for the rear axle midpoint (with associated desired velocitis $v_{id}(t)$), this must be converted into a reference motion for P by forward integration of the car-like equations, with $v = v_d(t)$. In both cases, there is no

smoothness requirement for $y_d(t)$ which may contain also discontinuities in the path tangent.

3.2 Implementation

The controller has been implemented in SIMULINK. This part is the same for both approaches, but two different files have been created: '***controller_PRM_BFS_smooth***' for the first approach and '***controller_RSC_Dijkstra***' for the second; since the simulation period changes, that is to say it is $200\,s$ for the first approach and $600\,s$ for the second. The following parameters were defined first, shown in Fig. 3.1:

$$l = 0.6\,m, \quad \Delta = 0.2\,m$$

These values together with the desired reference states obtained from MATLAB are then given as input to the block *Computation desired outputs y_{1d} and y_{2d}* which deals with returning as output the desired outputs y_{1d} and y_{2d} (Eq.3.2); which in turn together with coordinates of the point P, y_1 and y_2, are used as input to the controller block whose gains have been defined as $k_1 = k_2 = 5$ for the first approach while $k_1 = k_2 = 50$ for the second one, defined after a series of trial and error in order to check the best behaviour. Where the two virtual control inputs u are obtained as output (Eq. 3.6), and these together with θ, ϕ, l and Δ are given as input to the block *Computation velocities*; from which the velocities are obtained (Eq.3.4) and in turn they are given as input to the kinematic model of the robot under consideration. The car-like robot kinematic model is simulated through the predefined SIMULINK block *Ackermann Kinematic Model* in which the properties of the robot are defined and which gives the output states (Eq.1.5), ie x, y, θ and ϕ. Then, these values are given in feedback to obtain the outputs y_1 and y_2.

Notice that a saturation block was added on the ω variable since in some simulations it did not respect the maximum value. With this saturation block the performances were still very good.

Finally, various plots were carried out, which made it possible to understand the controller's behaviour, some of which were reported in the last chapter.

Chapter 4

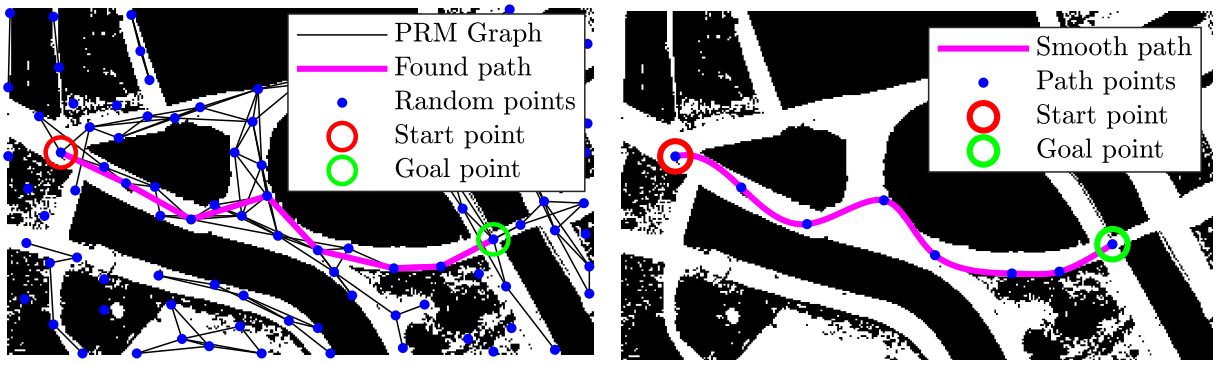
Results and simulation

In this chapter the results obtained with both approaches are reported, for space reasons only one case per approach is reported.

4.1 First Approach

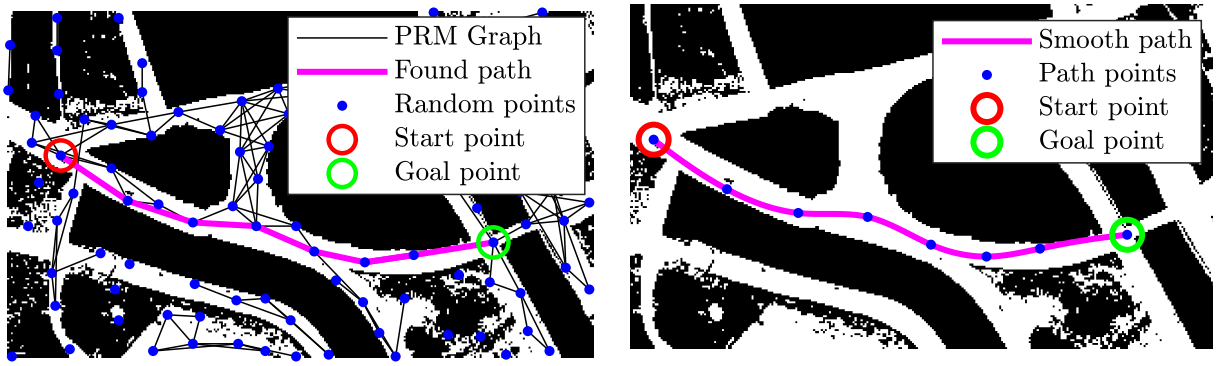
Matlab

The first approach is implemented in MATLAB code '***PRM_BFS_smooth***', from which at the end of the execution you get the path to go from the starting point to the final without obstacles. Consequently, the following are the graphs of a simulation, in which the first graphs shown (Fig.4.1) are incorrect, as the route intersects obstacles.



(a) Graph of the real map with obstacles (b) Real map with trajectory with obstacles

Figure 4.1: Trajectory with collisions.

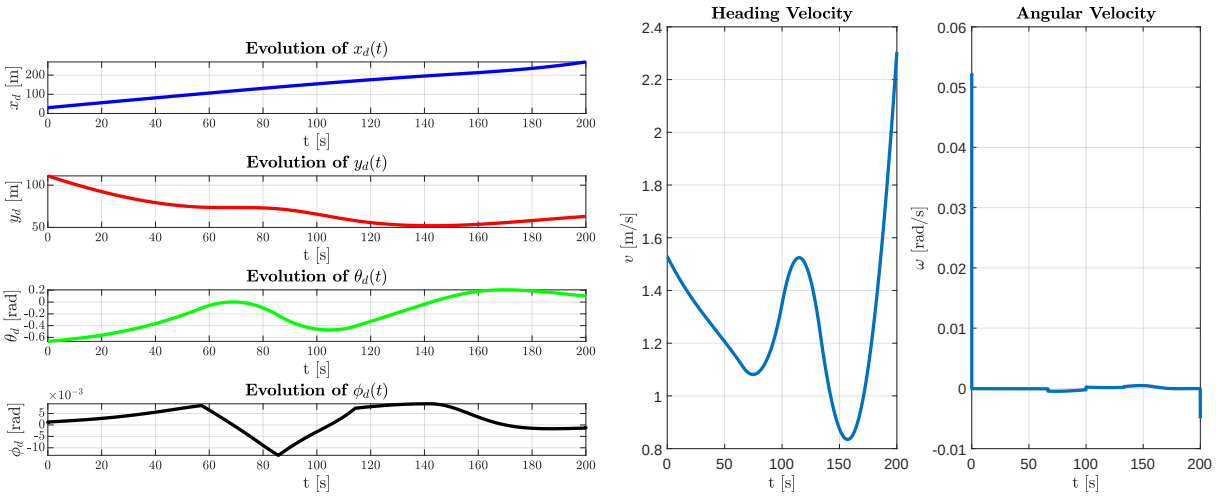


(a) Graph of the real map (b) Real map with trajectory

Figure 4.2: Trajectory without collisions.

Instead, in the Fig.4.2, the correct graphs obtained by MATLAB in the calculation of the reference trajectory are shown. As you can see on the left is the graph obtained using the PRM algorithm where the path consists of a series of broken lines connected to each other; while on the right the path obtained but smoothed is shown in magenta, while the nodes to be crossed are shown in blue and are part of the PRM graph, instead, the starting point is circled in red and the end point in green.

The last two Fig.4.3 show on the left the desired values, which are stored and together with the coordinates of the path that are provided to the controller implemented in SIMULINK. Instead, on the right the velocities obtained in MATLAB, where in all velocities plots for size issues '*Angular Velocity*' means the steering angle angular velocity.

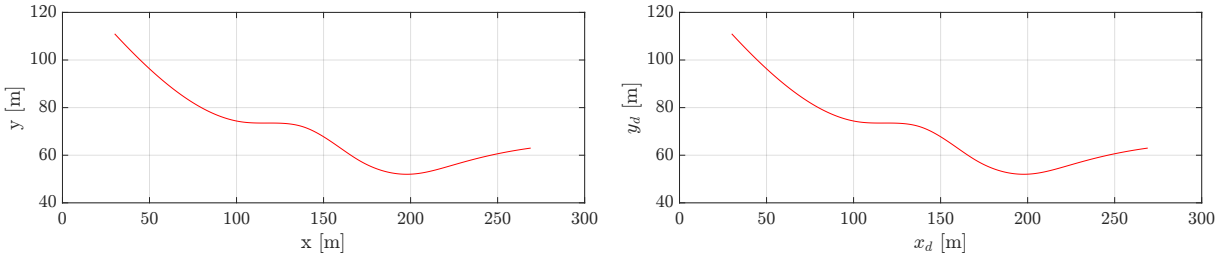


(a) Desidered values (b) Velocity

Figure 4.3: Values obtained.

Simulink

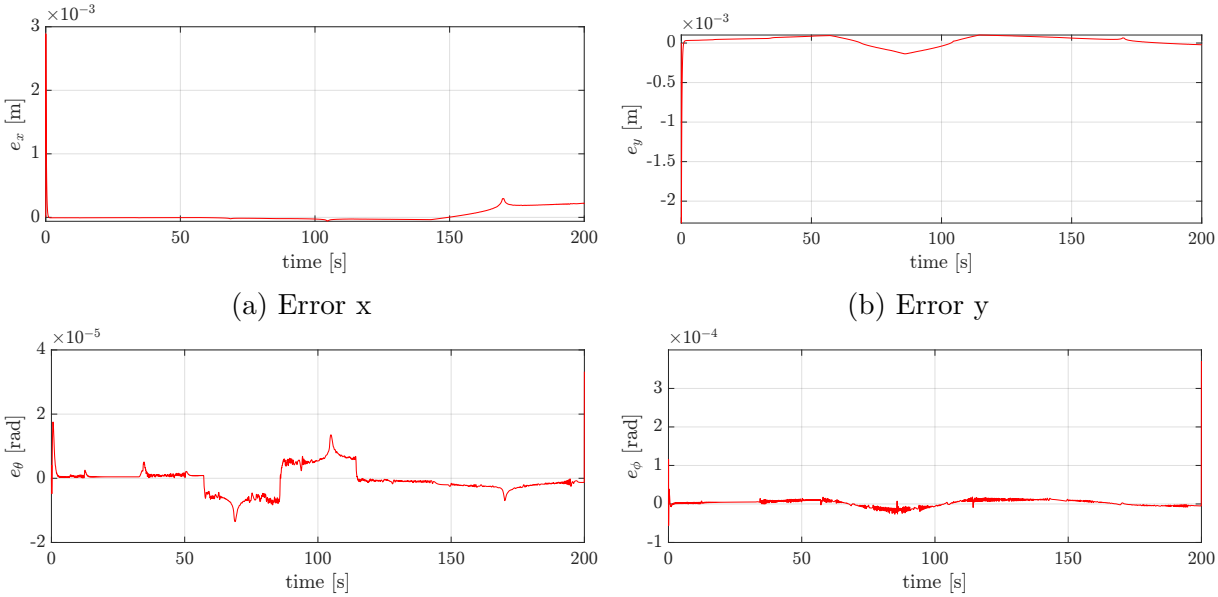
Now the results obtained for the first approach in SIMULINK are reported. The file '***controller_PRM_BFS_smooth***' contains the controller implementation. From the Fig.4.4 it is clear that the trajectory carried out is correct, because on the right is reported the desired trajectory obtained through the references of trajectory given in input to the controller, instead on the left the plot reports the trajectory that the car-like robot travels.



(a) Covered trajectory (b) Desidered trajectory

Figure 4.4: Trajectories.

Instead below in the Fig.4.5 the errors are reported. First of all the two figures above show the position errors which are both in the order of 10^{-3}. While on the bottom the orientation errors are reported which are respectively in the order of 10^{-4} and 10^{-5}. All of them are low and therefore acceptable errors which means that the path is followed



(a) Error x (b) Error y

(c) Error theta (d) Error phi

Figure 4.5: Errors obtained.

correctly.

It should be noted that the 'Input-Output Linearization' controller used is a controller which does not require a persistent trajectory, but at the expense of an uncontrolled orientation, ie θ (shown in Fig.1.1) is not controlled, but nevertheless assumes a very low error. The problem of following the full pose (position + orientation) of non-persistent trajectories is structural due to the robot's nonolonomicity.

Finally, the velocities are reported which compared to those obtained in MATLAB (reported in the Fig.4.3b) assume a similar behaviour and above all do not exceed the limits imposed:

$$v_{max}(t) = 3\,m/s, \quad \omega_{max}(t) = 0.43\,rad/s$$

Also in this case, ω means the steering angle angular velocity.

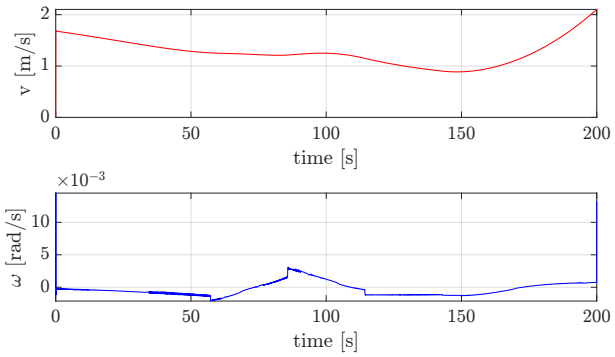


Figure 4.6: Velocities.

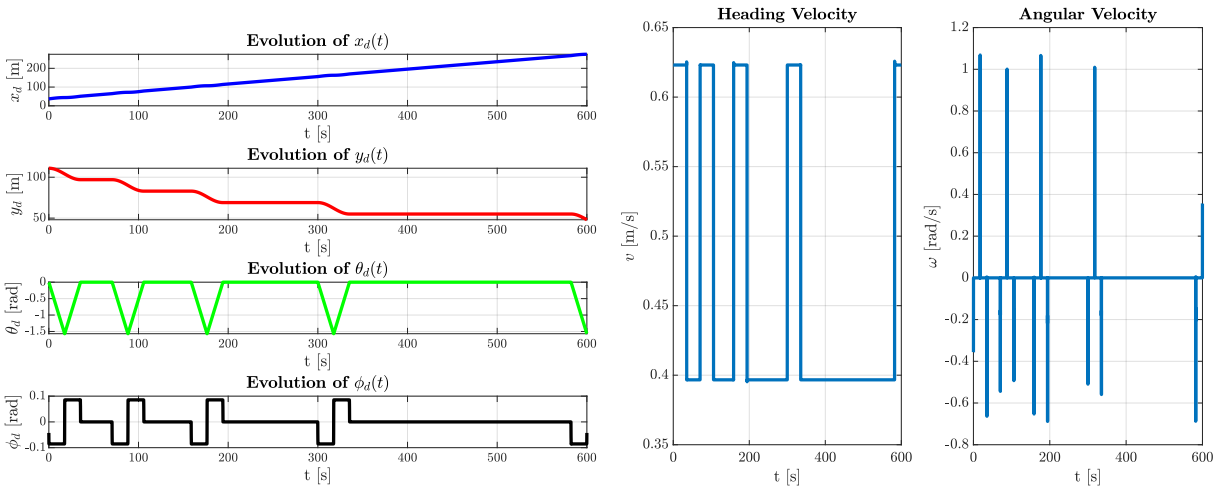
4.2 Second Approach

Matlab

The results of the second approach implemented in the MATLAB code '***RSC_Dijkstra***' are presented below. In which a single graph is obtained directly, unlike the previous approach, because using the RSC algorithm in which in the generation of curves is already checked on the collision of obstacles, any curve run will be without obstacles. In the following Fig.4.7, therefore, the nodes of the graph are represented in blue and the curves connected them in black, and in the same way as all the plots seen previously, the path is magenta, the starting point is red and the end point is green.

Figure 4.7: Graph of real map with trajectory.

Same as the previous section the last two Fig.4.8 show on the left the desired values, which are stored and together with the coordinates of the path that are provided to the controller implemented in SIMULINK. Instead, on the right the velocities obtained in MATLAB, which shows that the angular velocity of steering does not comply with the speed limits given in the next subsection, but this is not a problem thanks to the use of the saturator in the controller. It is always necessary to underline that '*Angular Velocity*' means the steering angle angular velocity.



(a) Desidered values

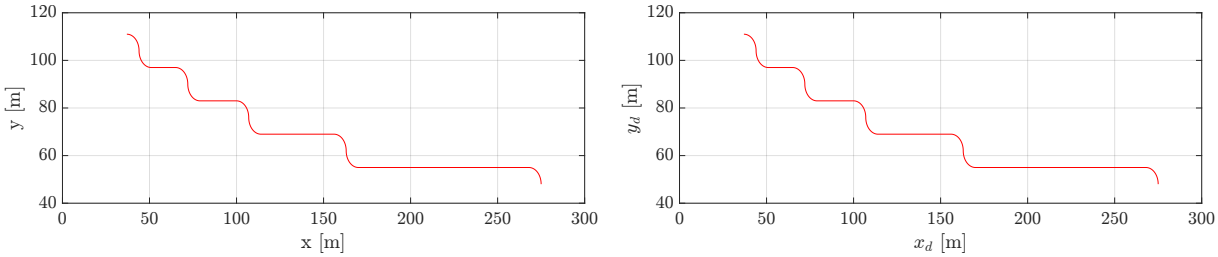
(b) Velocity

Figure 4.8: Values obtained.

Simulink

In the same way as for the first approach, the results obtained for the second approach in SIMULINK are reported. The file '***controller_RSC_Dijkstra***' contains the controller

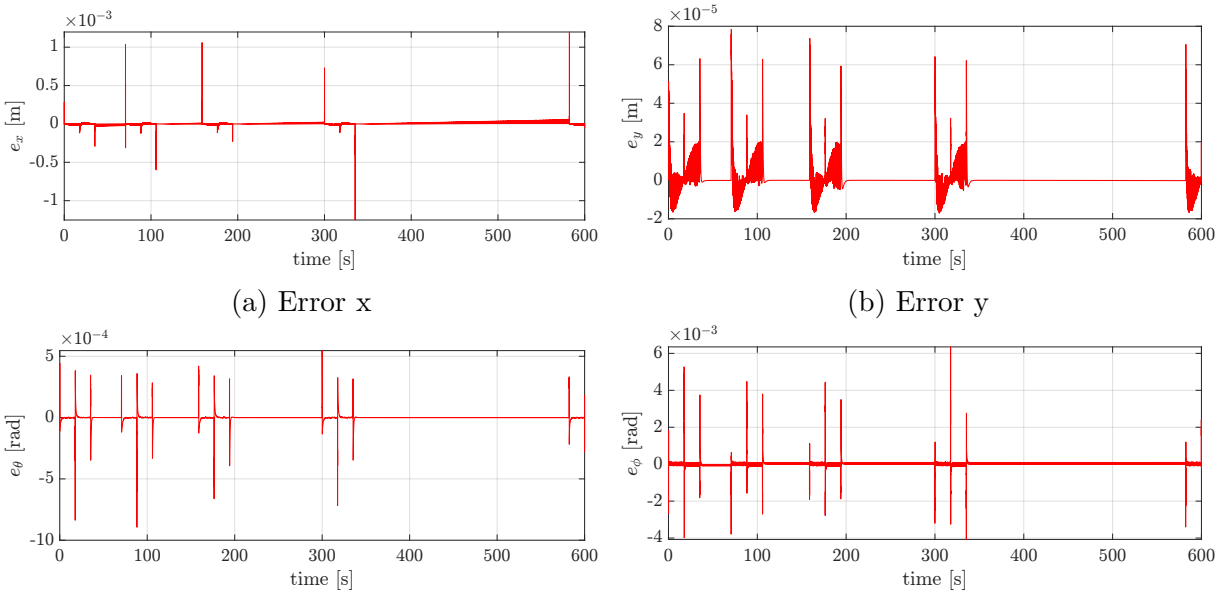
implementation. From the Fig.4.9 again on the right is reported the desired trajectory obtained through the references of trajectory given in input to the controller, instead on the left the plot reports the trajectory that the the robot under consideration runs through. It is noticed that comparing it also with the path obtained by the MATLAB plot the trajectory carried out is correct.



(a) Covered trajectory (b) Desidered trajectory

Figure 4.9: Trajectories.

In addition to the previous results shown, the subsequent plots (Fig.4.10) that report the errors obtained also behave in the same way as the first approach, obtaining almost similar results. As for the position errors, it is achieved an even lower error for the y of the order of 10^{-5} while for the x it remains in the order of 10^{-3}. However, θ and ϕ are not controlled, they have a greater error than the first approach, but it is still a small error



(a) Error x (b) Error y

(c) Error theta (d) Error phi

Figure 4.10: Errors obtained.

respectively of the order of 10^{-4} and 10^{-3}. So again the errors are considered acceptable and low, which allows to say that the trajectory is followed correctly.

At the end the velocities assume a behaviour similar to those obtained in MATLAB (shown in the Fig.4.8b), in which it is noted that the angular velocity takes equal peaks of intesity, witch the same positive or negative sign, at the time of the curves to be executed. But as already mentioned above the values are not equal because in MATLAB do not respect the speed limits while in this case it is thanks to the saturator. So both respect the limits imposed:

$$v_{max}(t) = 3\,m/s, \quad \omega_{max}(t) = 0.43\,rad/s$$

Also in this case, ω means the steering angle angular velocity.

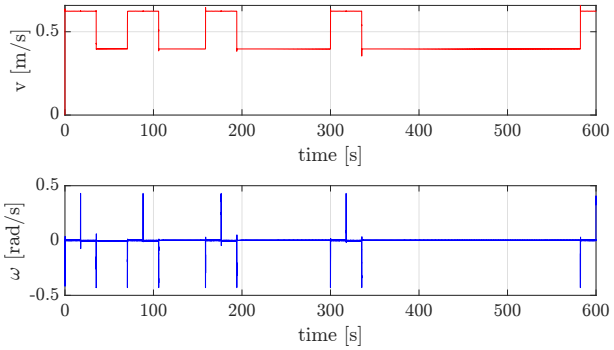


Figure 4.11: Velocities.

4.3 Improvements

First of all, it should be noted that the controller in question does not carry out any control on orientation. So the main improvement that you can make to this project is to insert a controller on orientation of the car-like robot.

Further tests that can be carried out are certainly to test an additional controller that receives the obtained reference trajectory as input, so that the behaviour of the two controllers can be compared and it can be verified which one performs better.