UNIVERSITA' DEGLI STUDI DI NAPOLI FEDERICO II

Dipartimento di Ingegneria Elettrica e delle Tecnologie dell'Informazione
Corso di Laurea Magistrale in Ingegneria Robotica e dell'Automazione

Elaborato finale del corso **Field and Service Robotics**

Control of a car-like warehouse robot via different types of path planning and estimated feedback

Anno Accademico 2024/2025

Candidato
Arsen Hudyma
matr. P38000269

Indice

Introduction

The scenario to implement within this projects is related in a warehouse and the main protagonist is a wheeled robot that knows the environment and has to go from the warehouse's main room to a destination room. Since the environment may be complex, a path planning technique has to be used, but one by which we can be sure to avoid obstacles. Regarding this point I've explored and implemented two pathways that can be used. I'll show you that the resulting paths are quite good but however, since the vehicle's state will be estimated through numerical integration, , we need robustness, and so the original map must be changed somehow. Of course another relevant issue is the control, in this case the adopted strategy regards a model-based framework, aiming to stabilize a virtual reference point rigidly linked to the mechanism of the robot, so that it follows a predefined trajectory. This is achieved through an input-output linearization technique, and of course we have to choose a valid kinematic model for the robot:

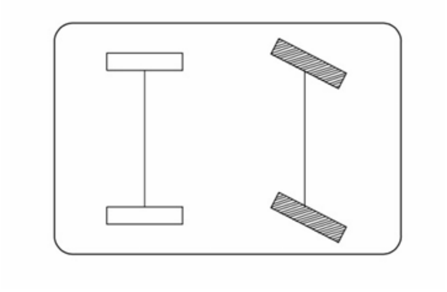
Chapter 1

Car Model

In this first chapter, we will see the adopted kinematic model for the representation of the warehouse robot, then we'll see the controllability analysis. Then the last topic will be presentation of the real world robot that has been considered in this scenario and it's kinematic parameters that have been used in the simulation as constraints for the motion.

1.1 Car-Like Kinematic Model

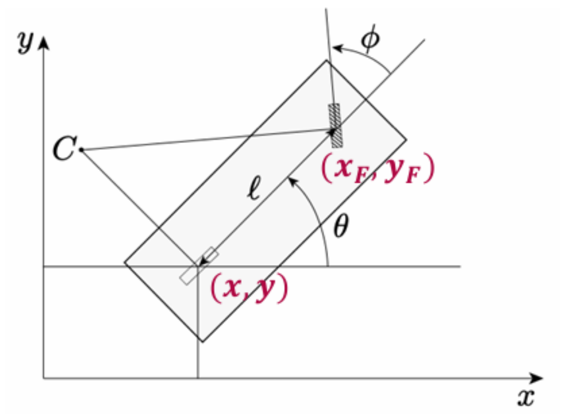
The car-like model features are two fixed (rear) wheels and two steerable wheels (front), but as we know we might have other types of configurations. The system is powered by two motors: one dedicated to traction, acting on the rear fixed wheels, and the other controlling the steering angle of the front wheels.



The bicycle model is kinematically equivalent to the car-like robot, and it simplifies the vehicle by representing each axle, both front and rear with a single virtual wheel positioned at the center of the axle. The generalized coordinates are the following.

$$q = [x, y, \theta, \phi]^T \tag{1.1}$$

And their meanings are these one instead:



Where x and y are the coordinates related to the rear wheel, x_f and y_f instead are related to the front wheel, θ represents the orientation of the vehicle with respect to the x axis, Φ is the steering angle, l the distance between the front and rear axis and for last C is the istantaneous centre of rotation.

$$x_f = x + l\cos\theta \tag{1.2}$$

$$y_f = y + l\sin\theta \tag{1.3}$$

The system is subject to two non-holonomic constraints (pure rolling constraint condition), one for each wheel:

$$\dot{x_f} sin(\theta + \phi) - \dot{y_f} sin(\theta + \phi) = 0 \tag{1.4}$$

$$\dot{x} sin\theta - \dot{y} sin\theta = 0 \tag{1.5}$$

By substituting the 1.2 and 1.3 in the first constraint we have:

$$\dot{x} sin(\theta + \phi) - \dot{y} sin(\theta + \phi) - \dot{\theta} l cos\phi = 0 \tag{1.6}$$

And the Pfaffian constraint matrix has constant rank equal to 2 and it is:

$$C = \begin{bmatrix} \sin(\theta + \phi) & -\cos(\theta + \phi) & -lcos\phi & 0 \\ \sin(\theta) & -\cos(\theta) & 0 & 0 \end{bmatrix} \tag{1.7}$$

If the robot is rear driven then it's kinematic model is:

$$\dot{q} = \begin{bmatrix} \cos\theta \\ \sin\theta \\ \tan\phi/l \\ 0 \end{bmatrix} v + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \omega \tag{1.8}$$

$$u_1 = v/cos\phi \tag{1.9}$$

$$u_2 = \omega \tag{1.10}$$

So v and ω represent respectively the linear velocity of the vehicle and the steering angular velocity, we can notice that a singularity occurs when the steering angle ϕ has values of 90° or 270°. However this conditions can't occur in real world scenarios because the steering angle of the wheels is always limited. A solution could also consist in the use the front wheels as the ones subject to the rotation and also traction.

1.2 Controllability analysis of the bicycle

The equation 1.8 can be rewritten as:

$$\dot{q} = g_1(q)v + g_2(q)\omega \tag{1.11}$$

The system under consideration exhibits the following properties: It is **nonlinear**, it is **drift-less**, meaning there is no autonomous term in the system dynamics, it has **fewer control inputs** (v_1, v_2) than the number of generalized coordinates (4 in total).

Although it may seem intuitive, based on everyday driving experience that a car-like vehicle should be able to reach any arbitrary configuration, establishing this capability on a rigorous mathematical basis requires verifying the **accessibility rank condition**, a fundamental tool for analyzing the controllability of drift-less nonlinear systems.

This condition states that the dimension of the *accessibility distribution* $\Delta_a(q)$ must equal the number n of the system's degrees of freedom. In our case, $n = 4$. The accessibility distribution is constructed by taking the span of the control vector fields g_1 and g_2, and extending it recursively with their successive *Lie brackets*, until the span potentially reaches the full tangent space.

If this span covers the entire state space at each configuration q, the system is said to be **locally accessible** and hence **controllable**.

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

Where:

$$g_3 = [g_1, g_2] = \begin{bmatrix} 0 \\ 0 \\ -\frac{1}{l\cos^2\phi} \\ 0 \end{bmatrix}, \quad g_4 = [g_1, g_3] = \begin{bmatrix} -\frac{\sin\theta}{l\cos^2\phi} \\ -\frac{\cos\theta}{l\cos^2\phi} \\ 0 \\ 0 \end{bmatrix}$$

Then we construct the matrix F with this 4 column vectors:

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

Since the matrix F has full rank, that is, $\text{rank}(F) = 4$, then $\dim(\Delta_a) = 4$. In this case, the system is fully **nonholonomic**, and it can be concluded that the system is accessible, meaning it can reach any configuration in the state space through a suitable choice of inputs.

1.3 Robot's kinematic parameters: MiR250 and OTTO 100

A requirement for this project was so do a state-of-the-art research based on the warehouse robots that are being currently used in the industry. Because the first goal is to have some knowledge about the adopted robots (like for example if the more used ones are the differential drive or the car like robots), but also to obtain some parameters of the robot to use in the simulation of the scenario, such as for example the maximum linear velocity or the distance between the wheels. The results of this research is that the most diffused robots are the differential ones due to their simplicity and the fact that in most cases this kind of robots are of a small size (even with small sizes they can have an average maximum load of 200 kg). But since another requirement of the project was to use a car-like robot I needed to do a deeper reasearch and it brought me to the: MiR250 (left) and to the $OTTO$ 100 (right):

Those are two robots that can be represented via the car-like model, I used both because kinematically and dimensionally they're very similar and each one has some parameters in the datasheet that aren't specified in the other. The considered parameters are: maximum heading velocity $v_{\max} = 2\ m/s$, maximum steering velocity $\omega_{\max} = 1.25\ rad/s$, maximum steering angle $\phi_{\max} = 0.69\ rad$, wheelbase length $l = 0.175\ m$ or $l = 0.475\ m$ and radius of the wheels $wheelR = 0.1\ m$. In the code you may notice that there are two different robot's parameters files, one is the short version and the other is the long one. This choice was made due to the fact that by an addictional analisys, I've discovered that the traction wheels of the MiR250 are at it's center, the others are just used for stability, so I wanted to consider also this difference in the simulation.

Chapter 2

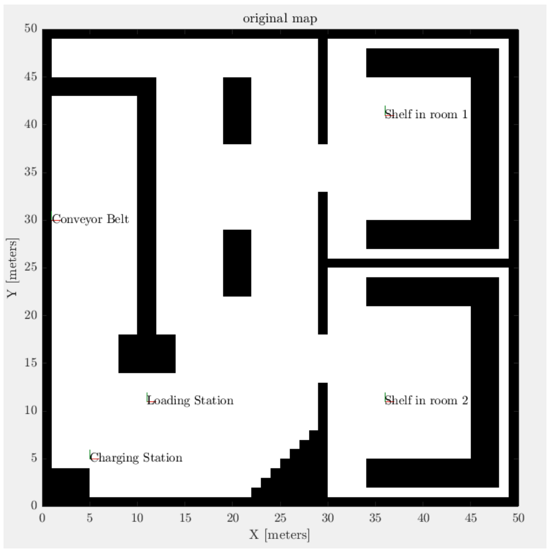
Warehouse's Map and Planning Among It

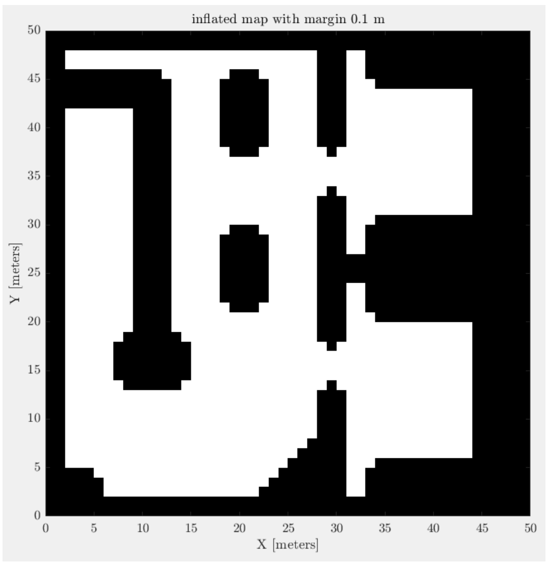
In this chapter I'll explain how the map of the warehouse is implemented considering also the obstacles being inside of it, then since we need some robustness to avoid collisions with the obstacles, we'll see some operations done on the original map to obtain a modified map that is inflated with respect to the obstacles. Then we'll be arguing about the path planning and trajectory generation. Regarding the first point I'll show you the adopted techniques to obtain a path that respects the car-like robot constraints and also avoids the obstacles. Then we'll see also the generation of a suitable time law by which we can ensure that the velocity limits of the robot aren't exceeded.

2.1 Map Definition

Starting from the project's constraints that we have about the environment, that consist in having an environment formed by three rooms, one being the warehouse and the latter being some destination rooms, and considering that the map is static we already have an intuition that the map needs to be modelled as a *grid*. This means that we're discretizing the environment in cells, that in this case are each of 1 m^2 and the overall dimension of the map is $50x50$ m. Each cell can be classified as: *occupied*, and so its value is 1; $free$, with a corresponding value of 0; *not yet discovered*, which means that the associated value is 0.5 (this isn't our case).

With inspiration of some warehouse designs I've realized a 2D one, which has also different types of obstacles to add some variety in it. A problem arises immediately by considering that in real world scenarios (for example because we're using *odometry*) we're not sure if the robot still can collide with an obstacle or not because of some effects that we're not considering or simply because of the accuracy, fortunately the solution arise as fast as the problem: $inflate$ the obstacles, in a way that the planner considers a modified map by which we can have a security margin on the motion of the robot. In particular it's a morphological dilatation of the input binary matrix (the grid) that consist in overlapping a disk of a certain radius r on every occupied cell and then marking as occupied all the cells inside the area of the disk, using a disk instead of another entity (for example a square) makes the dilatation isotropic. However here are shown the two maps, at the left the original one and at the right the inflated one:





2.2 Path Planning

Once we have defined the map our goal is to make our robot move from a starting position that is inside the warehouse to a destination position inside one of the two storage rooms, while of course avoiding the obstacles. In our help may come different approaches and algorithms fortunately and for this project I've implemented two types of them: the first approach is based on Rapidly-Exploring Random Trees RRT (in this implementation it's combined with RRG)+A* and the other one is based on Reed-Shepps curves. The goals are almost the same but the procedures really different and in the next section we'll see them in detail.

2.2.1 RRT/RRG+A*

The first approach consist in using a cascade of algorithms, firstly the RRT/RRG and then their output is used by A*. What RRT does basically consist in exploring the environment to end up to the destination position starting from an initial one, at each iteration it takes a step ahead that can be in a random direction or oriented to the destination (goal bias, practically we're pulling the algorithm where we need to). Once the next step is decided it also considers if the point is colliding with some obstacle or not, so it can decide if discard that step or keep it, until it reaches the goal. This way of proceed implies that the result of the algorithm is a tree structure, because it expands in the environment but there's only one path described by nodes in which we have to go through a sequence of arcs. This implies that the founded path isn't the optimal one and in general has a zig-zag shape. Here's the general algorithm instead:

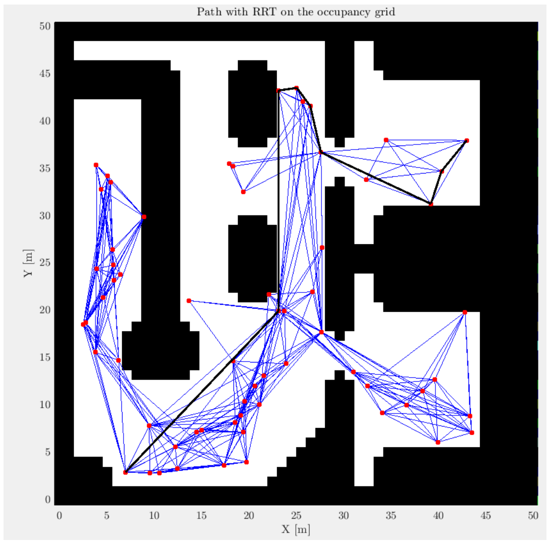
```
Algorithm 3: RRT
1 $V \leftarrow \{x_{\text{init}}\}$; $E \leftarrow \emptyset$;
2 for $i = 1, \ldots, n$ do
3     $x_{\text{rand}} \leftarrow$ SampleFree$_i$;
4     $x_{\text{nearest}} \leftarrow$ Nearest$(G = (V, E), x_{\text{rand}})$;
5     $x_{\text{new}} \leftarrow$ Steer$(x_{\text{nearest}}, x_{\text{rand}})$ ;
6     if ObtacleFree$(x_{\text{nearest}}, x_{\text{new}})$ then
7         $V \leftarrow V \cup \{x_{\text{new}}\}$; $E \leftarrow E \cup \{(x_{\text{nearest}}, x_{\text{new}})\}$ ;
8 return $G = (V, E)$;
```

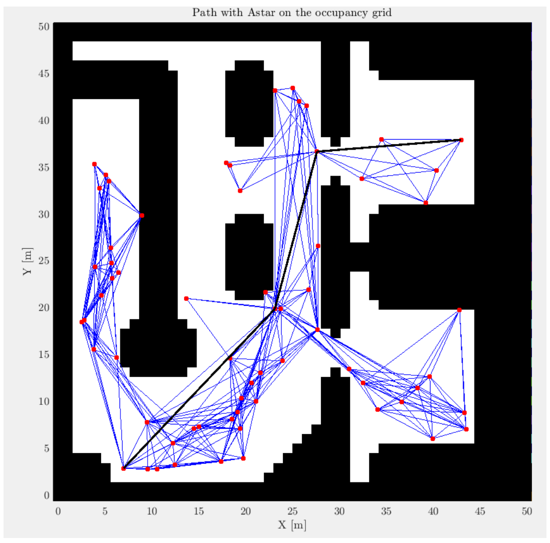
However we're not worried about the path, rather instead we want it to be not even near to an optimal path, because we'll find a better one with the A* algorithm. Unfortunately this latter algorithm works not on a tree structure but on a graph instead, specifically the issue is that we need to have more possible paths between the starting and the destination position to be able to find the shortest possible path. In fact if we pass the list of all the nodes and arcs obtained from RRT the path given by A* will be the same, because when RRT connects the start and the end position the algorithm end, giving us only *one* possible path (a different thing happens with RRT*). Based on these facts I've modified RRT to obtain as an output the path of RRT but rather than a tree structure a graph one, via RRG:

Algorithm 5: RRG

```
1  $V \leftarrow \{x_{\text{init}}\}$; $E \leftarrow \emptyset$;
2  for $i = 1, \ldots, n$ do
3      $x_{\text{rand}} \leftarrow$ SampleFree$_i$;
4      $x_{\text{nearest}} \leftarrow$ Nearest$(G = (V, E), x_{\text{rand}})$;
5      $x_{\text{new}} \leftarrow$ Steer$(x_{\text{nearest}}, x_{\text{rand}})$ ;
6      if ObtacleFree$(x_{\text{nearest}}, x_{\text{new}})$ then
7          $X_{\text{near}} \leftarrow$ Near$(G = (V, E), x_{\text{new}}, \min\{\gamma_{\text{RRG}}(\log(\text{card}\,(V))/\,\text{card}\,(V))^{1/d}, \eta\})$ ;
8          $V \leftarrow V \cup \{x_{\text{new}}\}$; $E \leftarrow E \cup \{(x_{\text{nearest}}, x_{\text{new}}), (x_{\text{new}}, x_{\text{nearest}})\}$ ;
9          foreach $x_{\text{near}} \in X_{\text{near}}$ do
10             if CollisionFree$(x_{\text{near}}, x_{\text{new}})$ then $E \leftarrow E \cup \{(x_{\text{near}}, x_{\text{new}}), (x_{\text{new}}, x_{\text{near}})\}$
11 return $G = (V, E)$;
```

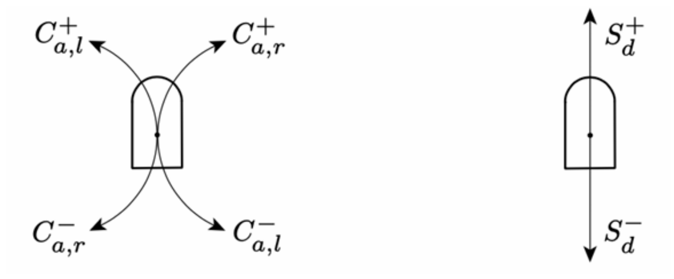
Which basically makes more connections between every node, then another issue arises: for A* we need to have a well connected graph, but it can explode and have too many edges. The solution that I've found is to use the K-NN approach, which consist by connecting the current node to at maximum the K nearest ones based on the minimum distance between them (obviously the ones avoiding obstacles too). Once that we've a graph we can use A*, this other algorithm minimizes the cost function $f(n) = g(n) + h(n)$ to reach the destination node based on $g(n)$ that is the cost to reach the current node starting from the initial one and $h(n)$ which is a heuristic cost based on the estimation of the remaining distance to the goal. The output is the minimal path constituted by the sequence of nodes that we need to go trough. Since this algorithm is longer to introduce and the fact that this report is limited I'll directly show you a comparison, on the left there's the path obtained with RRT/RRG and on the right the one via A*:



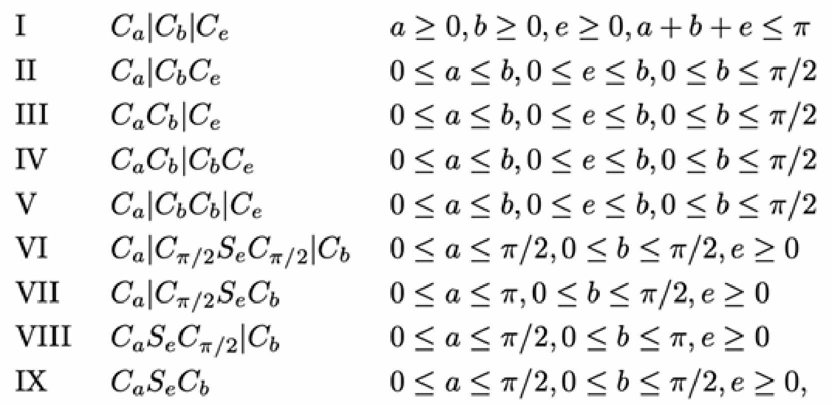


2.2.2 Reed Shepps Curves

This other path planning technique tries to obtain an optimal path in terms of the distance from a starting point to the finish one and its particularity is that the path is constructed via the concatenation of elementary arcs, that can be of two types mainly: *straight lines* of a certain length that can be traveled forward or backward by the robot, and arcs of circle of variable length that can be clockwise or counter-clockwise and as before forward or backward to add more flexibility (it's basically an upgrade of Dubins Curves):



Where Ca denotes an arc of distance a, Sd a line segment of distance d, $+/-$ forward/backward motion and r/l a clockwise/counter-clockwise motion. It's proven that the optimal path is constituted maximum of 5 elementary arcs, this means that the algorithm must evaluate every possible combination of paths, which are divided in *nine* groups leading to 48 possible combinations, of which we take simply the shorter. Here are shown the 9 groups:



I	$C_a \| C_b \| C_e$	$a \geq 0, b \geq 0, e \geq 0, a+b+e \leq \pi$
II	$C_a \| C_b C_e$	$0 \leq a \leq b, 0 \leq e \leq b, 0 \leq b \leq \pi/2$
III	$C_a C_b \| C_e$	$0 \leq a \leq b, 0 \leq e \leq b, 0 \leq b \leq \pi/2$
IV	$C_a C_b \| C_b C_e$	$0 \leq a \leq b, 0 \leq e \leq b, 0 \leq b \leq \pi/2$
V	$C_a \| C_b C_b \| C_e$	$0 \leq a \leq b, 0 \leq e \leq b, 0 \leq b \leq \pi/2$
VI	$C_a \| C_{\pi/2} S_e C_{\pi/2} \| C_b$	$0 \leq a \leq \pi/2, 0 \leq b \leq \pi/2, e \geq 0$
VII	$C_a \| C_{\pi/2} S_e C_b$	$0 \leq a \leq \pi, 0 \leq b \leq \pi/2, e \geq 0$
VIII	$C_a S_e C_{\pi/2} \| C_b$	$0 \leq a \leq \pi/2, 0 \leq b \leq \pi, e \geq 0$
IX	$C_a S_e C_b$	$0 \leq a \leq \pi/2, 0 \leq b \leq \pi/2, e \geq 0,$

But when the environment has some obstacles it may happen that the optimal path crosses on of them, because the algorithm does not take in account about them, so a method that can be utilized is to segment the principal path in more smaller ones, in a way that we can ensure that during all the minor paths and so the overall one the path doesn't cross an obstacle.

2.3 Trajectory Generation

A nice thing that simplifies the trajectory definition is that, like for the unicycle the generalized coordinates of the bicycle are the flat outputs of it. So supposed that we have a desired trajectory that is feasible, expressed in terms of:

$$x_d = x_d(t), \qquad y_d = y_d(t), \qquad t \geq t_0$$

Then we can obtain the corresponding time evolution of the other coordinates of the system, and so it's state evolution, and the associated command inputs that are needed to perform the motion that we want. Considering:

$$\dot{x}_d = v_{1d} \cos\theta_d \qquad \dot{y}_d = v_{1d} \sin\theta_d \tag{2.1}$$

$$\dot{\theta}_d = \frac{v_{1d}}{l} \tan\phi_d \qquad \dot{\phi}_d = v_{2d} \tag{2.2}$$

From the first 2 equations we can retrieve the desired velocity (positive if the motion is forward, negative otherwise):

$$v_{1d}(t) = \pm\sqrt{\dot{x}_d^2(t) + \dot{y}_d^2(t)} \tag{2.3}$$

The orientation is defined as (taking into account the sign of v_{1d}):

$$\theta_d(t) = \text{atan2}\left(\frac{\dot{y}_d(t)}{v_{1d}(t)}, \frac{\dot{x}_d(t)}{v_{1d}(t)}\right) \tag{2.4}$$

Then we can also obtain:

$$\dot{\theta}_d(t) = \frac{\ddot{y}_d(t)\dot{x}_d(t) - \ddot{x}_d(t)\dot{y}_d(t)}{v_{1d}^2(t)} \tag{2.5}$$

Substituting this one in the first of (2.2) we obtain the desired steering angle:

$$\phi_d(t) = \arctan\left(\frac{l\left(\ddot{y}_d(t)\dot{x}_d(t) - \ddot{x}_d(t)\dot{y}_d(t)\right)}{v_{1d}^3(t)}\right) \tag{2.6}$$

And at the end, derivating this we obtain the steering velocity:

$$v_{2d}(t) = \frac{l v_{1d}(\dddot{y}_d\dot{x}_d - \dddot{x}_d\dot{y}_d)v_{1d}^2 - 3(\ddot{y}_d\dot{x}_d - \ddot{x}_d\dot{y}_d)(\ddot{x}_d\dot{x}_d - \ddot{y}_d\dot{y}_d)}{v_{1d}^6(t) + l^2(\ddot{y}_d\dot{x}_d - \ddot{x}_d\dot{y}_d)^2} \tag{2.7}$$

So basically we can obtain the state and the input trajectories based on the desired motion of the robot, but we can go in trouble when $v_{1d}(t) = 0$ for some $t \geq t_0$ because the derived expression aren't going to be valid. So, a small but still important detail is that when using RS curves and in particular having a sequence of more simpler paths, it's better to avoid having the last point of the path i being also the first of the path $i + 1$.

Chapter 3

Sensing and Control

3.1 I/O Linearization Control

The adopted control technique aims to stabilize the position of a point $P = (y_1, y_2)$ in the cartesian space, so that it can be near to a desired trajectory. This point is rigidly connected to the robot at a certain distance b from the front wheel (if the b value is negative then we're refering the point to the rear part of the robot or above). The position of the point with respect to the robot's configuration is:

$$\begin{cases} y_1 = x_f + b\cos(\theta + \phi) \\ y_2 = y_f + b\sin(\theta + \phi) \end{cases} \tag{3.1}$$

Deriving it with respect to t and by using the kinematic model:

$$\begin{bmatrix} \dot{y}_1 \\ \dot{y}_2 \end{bmatrix} = T(\theta,\phi) \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} \tag{3.2}$$

where $T(\theta,\phi)$ is known as the input matrix of the reduced system:

$$T(\theta,\phi) = \begin{bmatrix} \cos\theta - \tan\phi\sin\theta - \dfrac{b\tan\phi}{l}\sin(\theta+\phi) & -b\sin(\theta+\phi) \\ \sin\theta + \tan\phi\cos\theta + \dfrac{b\tan\phi}{l}\cos(\theta+\phi) & b\cos(\theta+\phi) \end{bmatrix} \tag{3.3}$$

We can invert T if $b \neq 0$, and in such case the system is fully actuated. So we can:

$$\begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = T^{-1}(\theta,\phi) \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} \tag{3.4}$$

with u_1 and u_2 being some auxiliary control inputs. If we put (3.4) in (3.2) we obtain:

$$\dot{y}_1 = u_1 \qquad \dot{y}_2 = u_2 \tag{3.5}$$

And we obtain that:

$$\begin{cases} u_1 = \dot{y}_{1d} + k_1(y_{1d} - y_1) \\ u_2 = \dot{y}_{2d} + k_2(y_{2d} - y_2) \end{cases} \tag{3.6}$$

with $k_1, k_2 > 0$. With this choices we have the certainty that the position of the point P will converge exponentially to the desired trajectory (y_{1d}, y_{2d}). This technique doesn't control directly the orientation of the robot. However, as the point P follows the desired trajectory, the robot naturally aligns its orientation to keep P on the path. As a result, the orientation error tends to zero over time, even if it's not explicitly controlled.

3.2 Feedback Strategies

As we well know, we need the closure of the control loop with the feedback to have better performances, this implies that in this case we need to know at each t the robot's state. But when we work in the field of robotics it's not so simple like for robot manipulators instead.

3.2.1 Full-State Feedback

So as a starting point we assume to have some reliable proprioceptive and exteroceptive sensors by which we know at each t the state of the system.

3.2.2 Odometry

Then we proceed with the odometry, by which we can have an estimation of the MiR's state via some data obtained by encoders, IMU or even the commands sent to the robot, but it can have some drift phenomenon of the error with the increasing of the simulation time. However we could use also GPS data in this case if it's signal can reach up to inside the warehouse. In this project specifically I'm using a numerical method for the estimation of the robot's state: the Runge Kutta integration of 2nd and 4th order:

2nd Order Runge-Kutta

This method is more precise than the Euler's approximation and the computational load is lighter that the 4th order R-K:

$$\begin{cases} x_{k+1} = x_k + v_{1k} T_s \cos\left(\theta_k + \frac{1}{2} T_s \frac{v_{1k}\tan\phi_k}{l}\right) \\ y_{k+1} = y_k + v_{1k} T_s \sin\left(\theta_k + \frac{1}{2} T_s \frac{v_{1k}\tan\phi_k}{l}\right) \\ \theta_{k+1} = \theta_k + T_s \frac{v_{1k}\tan\phi_k}{l} \\ \phi_{k+1} = \phi_k + T_s v_{2k} \end{cases} \tag{3.7}$$

Ts is the sampling time and v_{1k}, v_{2k} are the command velocities

4th Order Runge-Kutta

It's a more precise integration but of course computationally heavier:

$$\begin{cases} x_{k+1} = x_k + \frac{T_s v_{1k}}{6}\left[\cos\theta_k + 5\cos\left(\theta_k + \frac{1}{2} T_s \frac{v_{1k}\tan\phi_k}{l}\right)\right] \\ y_{k+1} = y_k + \frac{T_s v_{1k}}{6}\left[\sin\theta_k + 5\sin\left(\theta_k + \frac{1}{2} T_s \frac{v_{1k}\tan\phi_k}{l}\right)\right] \\ \theta_{k+1} = \theta_k + T_s \frac{v_{1k}\tan\phi_k}{l} \\ \phi_{k+1} = \phi_k + T_s v_{2k} \end{cases} \tag{3.8}$$

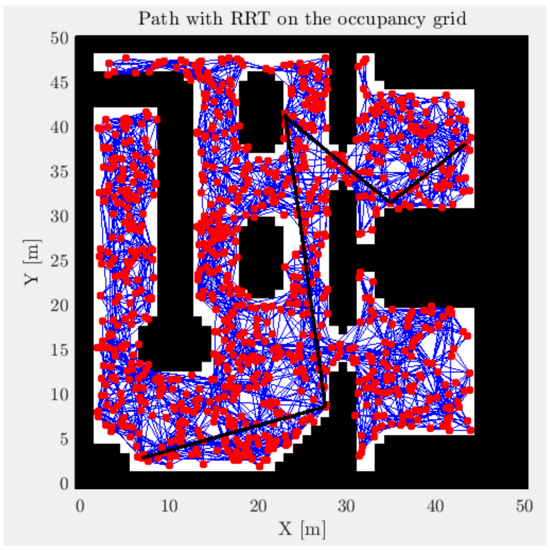
Chapter 4

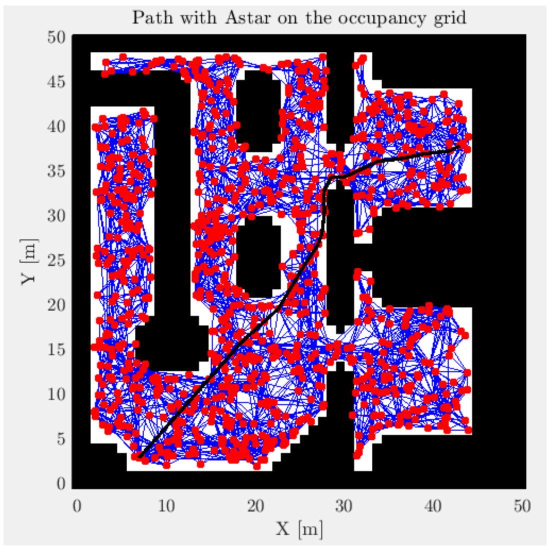
Results and Discussion

In this final chapter I'll show you the result and the performances of the whole system that has been implemented, starting from the results of the path planning for the robot inside an environment that has some constraints and so how the various implemented techniques can avoid effectively the obstacles, with considerations on the pros and lacks of each.Then we'll see how the inflation of the obstacles can help us to avoid collision and so to have a more robust control, considering also that in this scenario the odometry is being used, and this means that for longer simulations a drift phenomenon of the error may occur, in terms of the effective position with respect to the evaluated approximation of it. In particular I'll show you an example of the use of RRT/RRG+A* and then the RS curves, considering as feedback respectively the estimation of the pose though the 2^{nd} and 4^{th} Runge-Kutta.

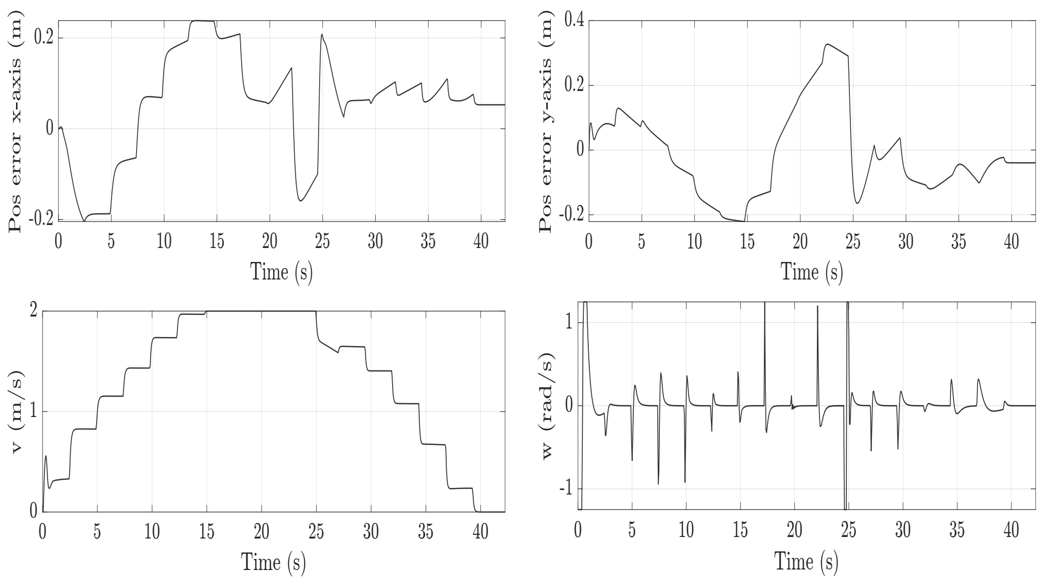
4.1 Test with RRT/RRG+A*

In this first scenario the destination room is the upper one in the map, and the results of path planning are the sequent ones:





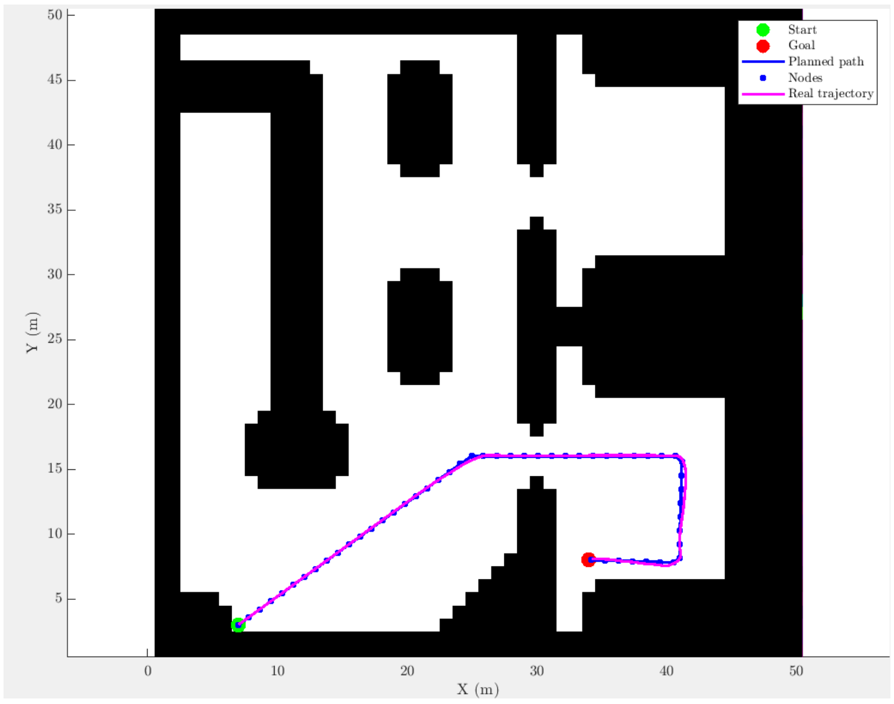
From these initial results we can notice that the use of A* is very relevant because otherwise we would have a segmented trajectory which would bring the robot for sure not even close to the shortest path. Instead with A* even if we don't have *the best* path it's sure that it is a good one. Problems may arise because A* doesn't consider the obstacles and so in order to have a path that somehow minimizes the distance, and the result is that the desired trajectory is near to obstacles, but with their inflation we're sure that we have a safety margin. Then let's see the results in terms the controlled system's behavior:



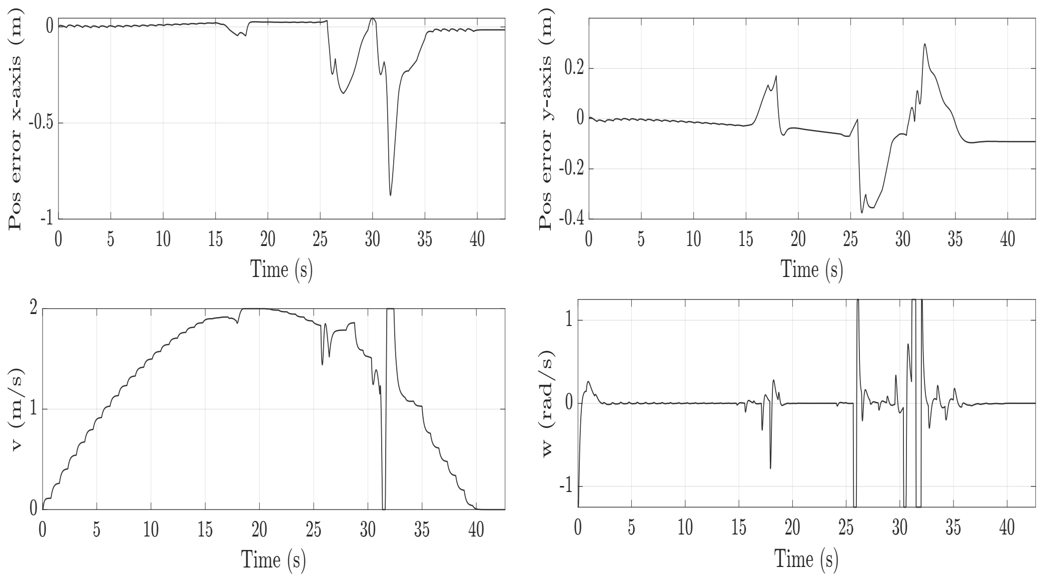
Something that we can notice is that the error doesn't converge to zero, but considering that the map area is 2500 m^2 it's quite good (however the errors regarding $y1$ and $y2$ are converging to zero). Then if we remember what was said in section 2.3 we can see that the robot's kinematic parameters are being respected and also that we have a trajectory time evolution scaling.

4.2 Test with Reed-Shepps Curves

This time the destination room is the lower one, since we're using RS and they don't implement some sort of obstacle consideration, the input must be a sequence of points through which we can be sure that the path will be collision free. So now I'll show you the difference between the planned path (in blue) and the real one that the robot is doing trough the control (in purple):



We can notice that the real path converges to the planned one but also the tracking capabilities are really good, a conclusion may be that tracking capabilities depend also on the number of points/nodes present on the actual path. To conclude, like before I'll show you the performance of the controlled system:


Pos error x-axis (m)
Pos error y-axis (m)
v (m/s)
w (rad/s)
Time (s)