Field and Service Robotics
Technical Report

Motion Planning and Motion Control of a differential-drive robot

Instructor:
Fabio Ruggiero

Students:
Marco Caianiello M58/242

Contents

List of Figures

Introduction

This report has been written for the exam of Field and Service Robotics with the aim to explore the several possible solution for solving the motion planning problem of a differential-drive robot. In the first chapter of this work will be analyzed the kinematic model of the differential drive robot and its similarity with the unicycle kinematic model. In addition, it will be presented the real robot model used for the simulation purpose with its characteristics. In the second chapter is possible to find the solution at the motion planning problem. Hence, the mathematical formulation and the choices that have been done to complete the assignment. The tools used are the RRT algorithm combined with the $A^\star$ algorithm. In the same chapter can be found a detailed description of the simulation environment. The third chapter contain the mathematical description about the controller implemented for the motion control of the robot. They have been developed the Input/Output Feedback Linearization controller and the Posture controller in order to resolver the trajectory tracking and posture regulation problem. In the last chapter are shown the software that have been used to implement and simulate the project and how they communicate. In addition, in the same chapter are presented the results of the simulations in two different environment for testing the quality of the solution developed.

Chapter 1

Robot Description

1.1 Differential-drive robot

A differential-drive robot is made up of a rigid body with three wheels that provide the motion. Two of them are free to rotate around a common axis of rotation, the other one, typically a caster wheel, has the function to keep the robot statically balanced and it is passive during the motion. The two fixed wheels are separately controlled by two different drivers, so different values of angular velocity may be arbitrarily imposed. If we consider ω_l and ω_r as their rotational velocities, we can easily classified four type of possible motion:

- $\omega_l = \omega_r$: moving forward or backward
- $\omega_l > \omega_r$: turning right
- $\omega_l < \omega_r$: turning left
- $\omega_l = -\omega_r$: turning on the spot

To study a differential-drive robot we need to know the Cartesian coordinates (x,y) of the midpoint of the segment joining the two wheel centres and the angle θ that represent the orientation of the rigid body. These parameters are the same that are used to describe the configuration of a unicycle and then the two vehicles can be considered kinematically equivalent.

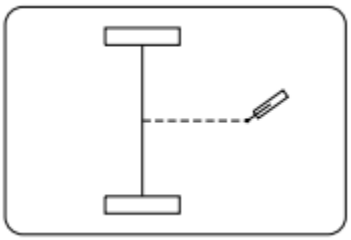

Figure 1.1: A differential-drive mobile robot

1.1.1 Kinematic Model

The unicycle configuration is completely described by $q = [\,x \quad y \quad \theta\,]^T$, where the Cartesian coordinates (x,y) indicates the contact point on the ground and θ the orientation respect the x axis.

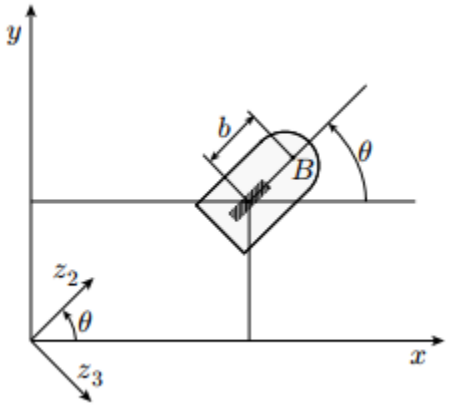

Figure 1.2: Generalized coordinates for a unicycle

The vehicle can not have a velocity component orthogonal to the sagittal plane, so only a pure rolling movement is allowed. This constraint can be express in the Pfaffian form as

$$\dot{x}\sin\theta - \dot{y}\cos\theta = [\,\sin\theta \quad \cos\theta \quad 0\,]\dot{q} = 0 \tag{1.1}$$

The (1.1) is a nonholonomic constraint because whole the configuration space C remain accessible, while not all the instant motion are allowed. The matrix $G(q)$ can be built with the columns as basis of the null space associated to the Pfaffian constraint

$$G(q) = [\, g_1(q) \quad g_2(q) \,] = \begin{bmatrix} \cos\theta & 0 \\ \sin\theta & 0 \\ 0 & 1 \end{bmatrix} \tag{1.2}$$

All the admissible generalized velocities at q are obtained as linear combination of $g_1(q)$ and $g_2(q)$, then the kinematic model of the unicycle can be expressed as

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \cos\theta \\ \sin\theta \\ 0 \end{bmatrix} v + \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \omega \tag{1.3}$$

The two inputs v and ω represent respectively the driving velocity and the sterring velocity, i.e. the wheel angular speed around the vertical axis. The (1.3) can be applied, as said above, to the differential-drive robot and the inputs v and ω can be expressed as function of the angular speeds of the left and right wheel, ω_l and ω_r:

$$v = \frac{r(\omega_r + \omega_l)}{2} \qquad w = \frac{r(\omega_r - \omega_l)}{d}$$

where r is the wheel radius and d the distance between their centres.

1.2 Turtlebot3 - Waffle

TurtleBot3 is a small, affordable, programmable, ROS-based mobile robot for use in education, research, hobby, and product prototyping. The TurtleBot3 can be customized into various ways depending on the project where it is used. For this project has been chosen the model named "Waffle", a differential drive robot that has the characteristics reported in the table 1.1. This robot has been chosen because its core technology is SLAM and Navigation, then perfectly suits with the aim of this project. In addition, the official site provide a lot of documentation and ROS-code like the file that has been used in this work for the simulations in the physics engine.

In the fig. 1.3 is possible to read the robot's dimensions. For the aim of this project the main information is the external radius equals to 0.22m. This parameter will be useful for building the obstacles space required for the Motion Planning.

Maximum translational velocity	0.26 m/s
Maximum rotational velocity	1.82 rad/s (104.27 deg/s)
Maximum payload	30kg
Size (L x W x H)	281mm x 306mm x 141mm
Threshold of climbing	10 mm or lower
Expected operating time	2h
SBC (Single Board Computers)	Intel® Joule™ 570x
Actuator	XM430-W210
IMU sensor1	Gyroscope 3 Axis
IMU sensor2	Accelerometer 3 Axis
IMU sensor3	Magnetometer 3 Axis
Power adapter (Input)	100-240V, AC 50/60Hz, 1.5A
Power adapter (Output)	12V DC, 5A

Table 1.1: Turtlebot3 - Waffle specifications

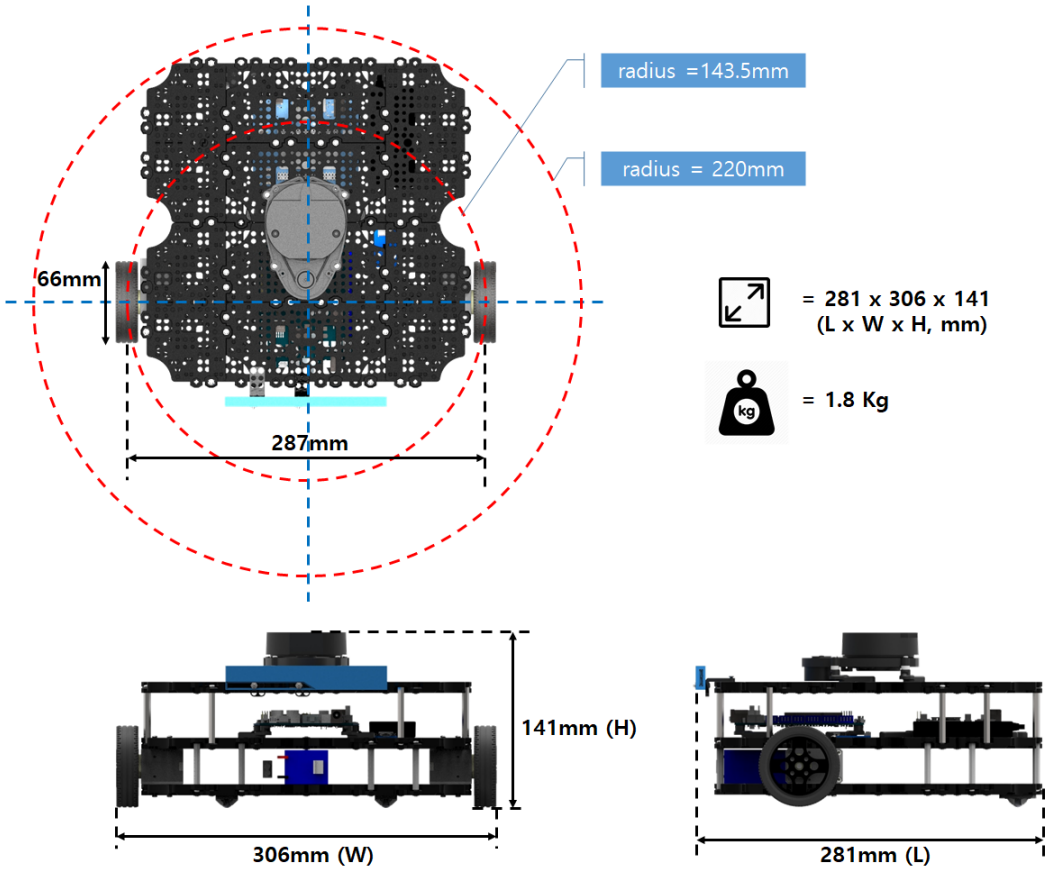

Figure 1.3: Turtlebot3 - Waffle dimensions

Chapter 2

Motion Planning

2.1 Configuration Space and Obstacles Space

In this chapter will be expose the solution adopted to solve the Motion Planning problem. Consider $W = \mathbb{R}^N$ the robot workspace and O the set of the obstacles in W, the motion planning problem consist to find a path, if it exist, for the robot from a start configuration to a final configuration, defined in W, avoiding collisions between the robot and the obstacles in O. The robot configuration can be expressed thanks its generalized coordinates, as seen above for the unicycle, and the set of all the configurations that the robot can assume is called Configuration Space C. The configuration space of a robot is obtained as a Cartisian product of its workspace and the special orthonormal group $SO(m)$ of real $(m \times m)$. In the specific case of the unicycle the workspace has dimension equal to 2, $W = \mathbb{R}^2$, and m equal to 2, $SO(2)$, hence the C is $\mathbb{R}^2 \times \mathrm{SO}(2)$, whose dimension is $n = 3$.
As said previously, in the workspace can be placed some obstacles that the robot have to avoid while it is trying to reach the goal configuration. An obstacle can be modeled as the subset of configurations that cause a collision, include simple contact between the robot (B) and the the effective obstacle (O_i):

$$CO_i = \{q \in C : B(q) \cap O_i \neq \emptyset\} \tag{2.1}$$

The subset CO_i is called C_obstacle and the union of all these is called

C-obstacle region, CO:

$$CO = \bigcup_{i=1}^{p} CO_i \tag{2.2}$$

Hence, the subset of configurations that do not cause collision, named free configuration space (C_{free}), is the complement of CO:

$$C_{free} = C - CO \tag{2.3}$$

The start configuration, the goal configuration and the path that connect them can be considered valid if they belong to C_{free}.
In this project the obstacles in W are assumed to be polygonal, so the boundary of the C-obstacles refers to CO_i is the locus of the configurations that put the robot in contact with the obstacle. The union of these point can be achieved easily by an isotropically growth (fig 2.1) of the obstacles area of a quantity equal to the robot radius.

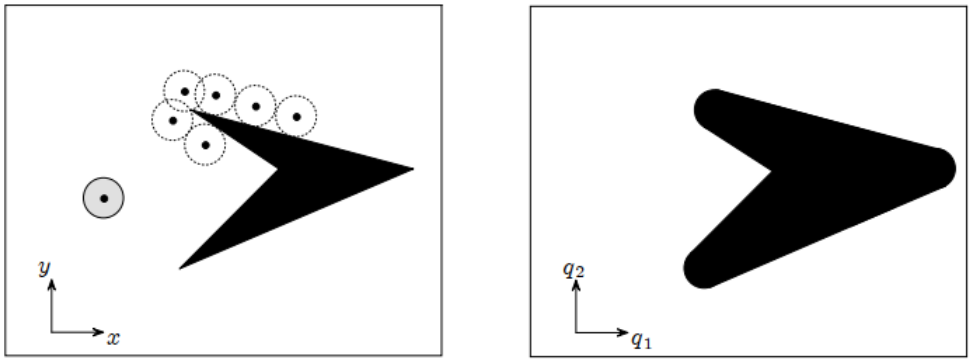

Figure 2.1: C-obstacles for a circular robot in $\mathbb{R}^2$

2.2 Scenario

The working area for this project is a square room with an edge of 10m. Inside this space are placed three obstacles, two cylinders and a rectangular-shaped. The former have a radius $r_c = 0.5m$ and an high $h_c = 2m$, the latter have a length $l_{rec} = 2.5m$ a width $w_{rec} = 1m$ and an high $h_{rec} = 2m$.

In fig 2.2 and 2.3 are shown two different kind of working area that have been used for the simulation purpose. These scenarios were thought with

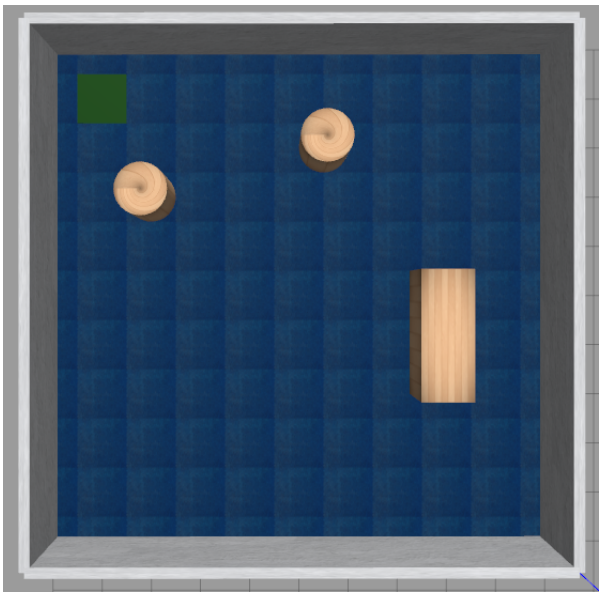

Figure 2.2: First simulation scene

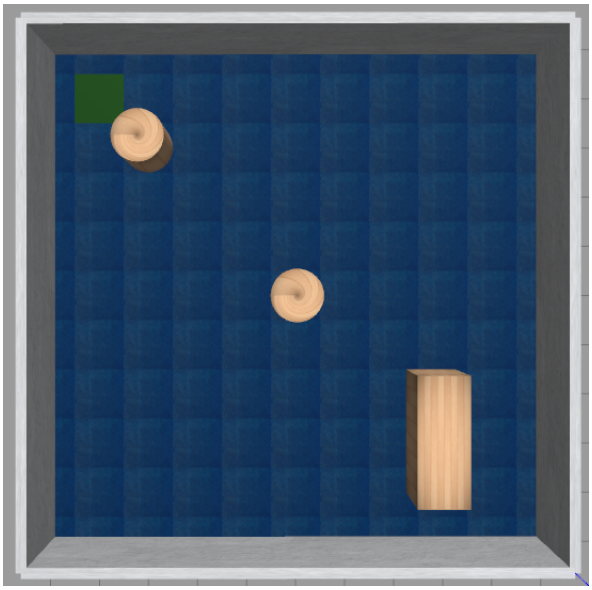

Figure 2.3: Second simulation scene

the aim to challenge the capabilities of the robot to reach the goal point (green square) from the starting point that is in down-right corner. The fist working area (fig. 2.2) has been thought with the purpose to block the straight line from the start point to the goal and also a possible alternative

way, in addition the goal has been placed behind an obstacle. The second one (fig. 2.3) want to test the skill of the robot to find a way toward the goal when the optimal path is blocked for all.

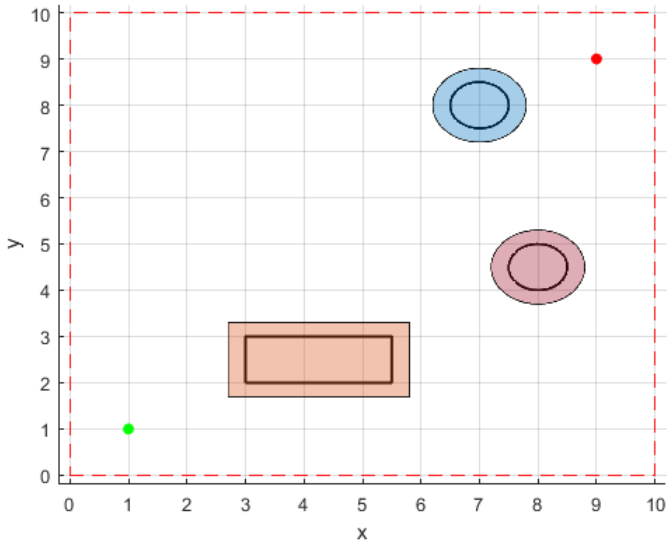

Figure 2.4: *CO*-obstacles for the fist scenario

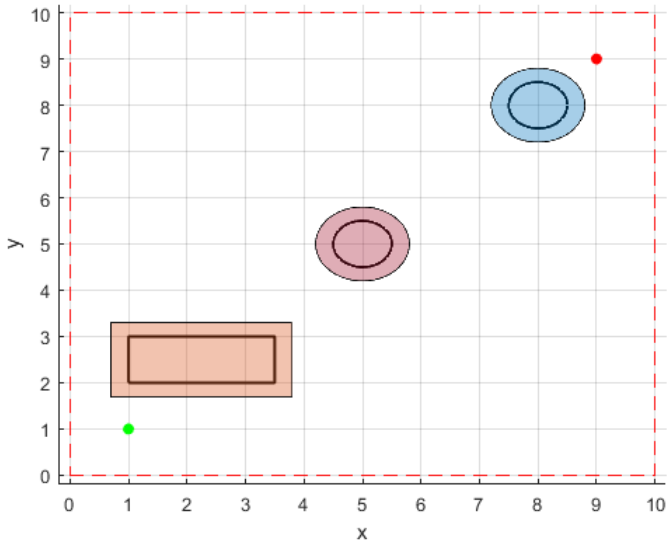

Figure 2.5: *CO*-obstacles for the second scenario

In fig. 2.4 and 2.5 are shown a graphic representation of *CO*-obstacles

for the first and second scenario. Each obstacle have two edges, the inner edge correspond to the real obstacle, the outer edge is the result of the isotropically growth. The robot radius $r_{bot} = 0.22m$, as write in previous chapter, for the growing process has been considered equal to $0.3m$. In this way the robot has a lit of bit of tolerance when it will be near an obstacle that compensate the position error during the navigation. Hence, if the cylinders have radius equal to r_c, the C-obstacle radius will be $r_c + r_{bot}$, while the C-obstacle for the rectangular-shaped will be $l_{rec} + r_{bot}$ and $w_{rec} + r_{bot}$. The green and red point are respectively the start point and the goal point.

2.3 RRT

RRT is a single-query probabilistic algorithm with the aim of rapidly explore the subset C_{free} to find a solution path of the motion planning problem. The fig. 2.6 show its pseudo-code.

```
RRT(q_start, q_goal)
1    T.add(q_start)
2    q_new ← q_start
3    while(DISTANCE(q_new, q_goal) > d_threshold)
4       q_target = RANDOM_NODE()
5       q_nearest = T.NEAREST_NEIGHBOR(q_target)
6       q_new = EXTEND(q_nearest, q_target,expansion_time)
7       if(q_new != NULL)
8             q_new.setParent(q_nearest)
9             T.add(q_new)
10   ResultingPath ← T.TraceBack(q_new)
11   return ResultingPath
```

Figure 2.6: RRT pseudo-code

The RRT relies on a simple randomized expansion of a tree T at each iteration until a maximum number of iterations or the goal are reached. The first step is the generation of a random configuration q_{rand} according to a uniform probability distribution in C. q_{near} is the closest node of T to q_{rand}, so a new configuration q_{new} is generated at a distance δ from q_{near} on the segment that join it with q_{rand}. At this point a collision check procedure is run and if the result is positive q_{new} is added to T with the link between it

and q_{near}.
In the case of nonholonomic robot, like the unicycle, instead of extend q_{near} on a straight line, is possible to use motion primitives produced by a specific choice of the velocity inputs in the kinematic model. In this project it has been considered the Waffle robot like a Dubins car, that is the backward motion and the rotation on spot is not allowed. Hence a constant positive linear velocity and bounded angular velocity have been chosen:

$$\begin{gathered} v = v_{max} = 0.26m/s \\ w = \{-\omega_{max}, 0, \omega_{max}\} = \{-1.82rad/s, 0, 1.82rad/s\} \end{gathered} \tag{2.4}$$

The v_{max} and ω_{max} value are provided by the Waffle specifications (tab. 1.1). Thanks to the combination of these three values, three different primitives can be used. One for the forward motion and two for turning the robot on left or on right.

- $[\, v_{max}, w_{max} \,] \rightarrow$ Turn left
- $[\, v_{max}, -w_{max} \,] \rightarrow$ Turn right
- $[\, v_{max}, 0 \,] \rightarrow$ Forward

In fig. 2.7 is possible to see the primitives used in this project. The forward motion and the turning motion have different duration Δt, the first one keep the maximum linear velocity of $0.26m/s$ with $\Delta t = 1s$ so the robot translate of $0.26m$, the other two keep both the maximum velocities for $\Delta t = 0.44s$ because it was thought for a linear variation $\Delta\theta = 0.81rad \approx 45°$ of the robot angle. In fig. 2.7 and 2.8 is possible to see the primitives and the angle displacement.

An RRT algorithm was developed for this project and the source code in provided with this report (`RRT_function.m`). The code offer a standard implementation of the RRT with ending condition on the maximum number of iterations and the achieving of the goal. The maximum number of iteration is set to 10000, while a primitive is considerate a solution if it falls inside a neighborhood of the goal point of radius equal to $0.1m$. In addition, the primitive is truncate at the nearest point from the goal, and the robot configuration at that point is added to the Tree. Example of Tree generate by the RRT algorithm are shown in fig 2.9 and 2.10.

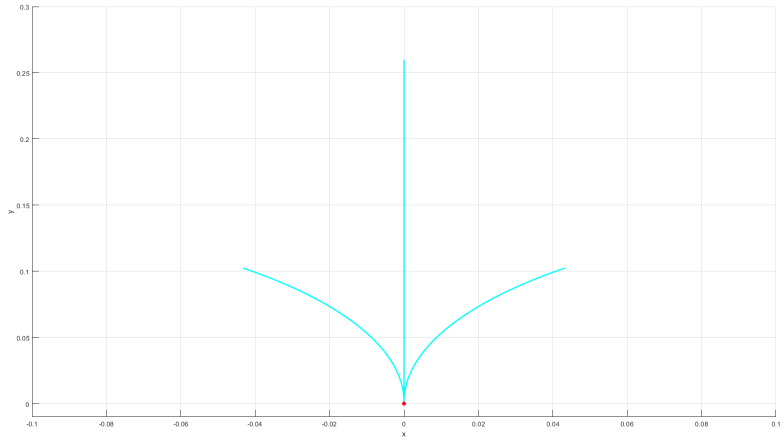

Figure 2.7: Motion primitives

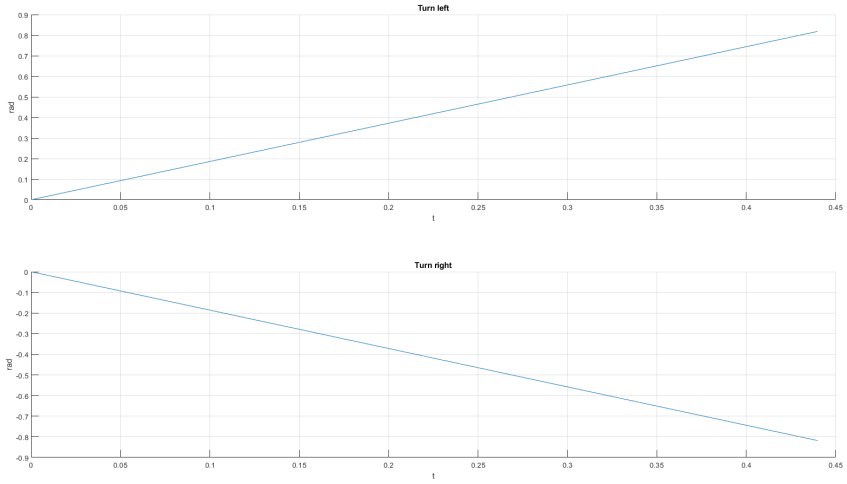

Figure 2.8: Robot orientation while making a turn

2.4 $A^\star$ Algorithm

$A^\star$ is a graph traversal and path search algorithm, so it connect the start node N_s to the goal node N_g of a graph G by the minimum cost path. Given G, like the Tree generated by RRT, $A^\star$ iteratively visit the node of the graph starting from N_s. At each step it store the minimum path from N_s to the visited node in a second Tree T. To calculate the cost of each node, $A^\star$ use

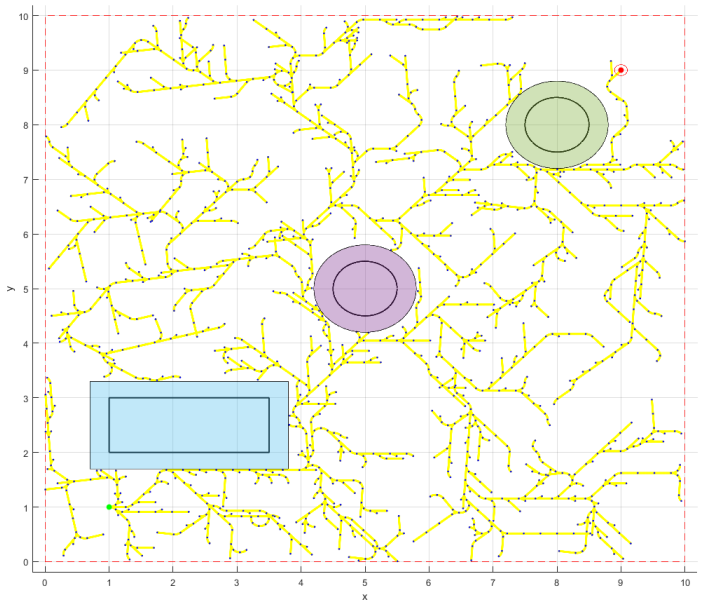

Figure 2.9: Fist example of a Tree generate by an RRT that use motion primitives

a cost function $f(N_i)$, where N_i is the node visited.

$$f(N_i) = g(N_i) + h(N_i) \tag{2.5}$$

$g(N_i)$ is the cost of the path from N_s to N_i and $h(N_i)$ is a heuristic estimate of the real cost $h^\star(N_i)$ between N_i and N_g. Frequently the $h(N_i)$ is the euclidean distance from the two node, because if the heuristic function does not overestimate the real cost $h^\star(N_i)$, $A^\star$ is complete.

An implementation of this algorithm was developed in this project and can be found in the file `Astar_function.m` where $g(N_i)$ and $h(N_i)$ are based on the euclidean distance from the nodes. The results path from the graphs in fig 2.9 and 2.10 are respectively in fig. 2.13 and 2.14.

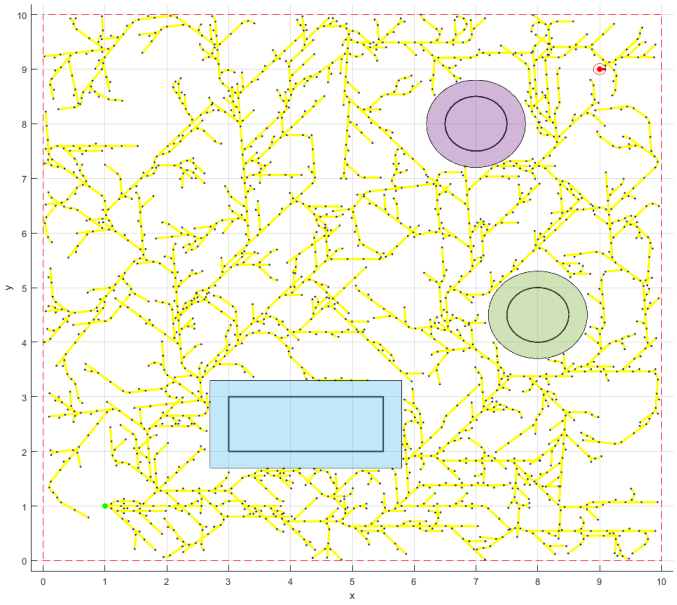

Figure 2.10: Second example of a Tree generate by an RRT that use motion primitives

```
while (iteration < max_iteration) && (~find_goal)
  %Generate random position
  q_rand(1) =  height*rand();
  q_rand(2) =  width*rand();

  %Find the nearest point
  index(iteration) = dsearchn(vert(:,1:2), q_rand);

  %Draw the primitive
  if rem(round(rand()),2) == 0
      [x_temp, y_temp, theta_temp, ~, ~] = rectilinear_primitive (vert(index(iteration),1),
      vert(index(iteration),2), vert(index(iteration),3));
  else
      if rem(round(rand()),2) == 0
          [x_temp, y_temp, theta_temp, ~, ~] = circular_primitive (vert(index(iteration),1),
          vert(index(iteration),2), vert(index(iteration),3), 1);
      else
          [x_temp, y_temp, theta_temp, ~, ~] = circular_primitive (vert(index(iteration),1),
          vert(index(iteration),2), vert(index(iteration),3), -1);
      end
  end

  %Check validity
  bound_condition = (prod(x_temp<height) && prod(y_temp<height) && prod(x_temp>0) && prod(y_temp>0));

  [in, on] = inpolygon(x_temp,y_temp,circle1.Vertices(:,1),circle1.Vertices(:,2));
  circle1_condition = (numel(x_temp(in)) == 0) && (numel(y_temp(in)) == 0) && (numel(x_temp(on)) == 0)
                      && (numel(y_temp(on)) == 0);

  [in, on] = inpolygon(x_temp,y_temp,square1.Vertices(:,1),square1.Vertices(:,2));
  square1_condition = (numel(x_temp(in)) == 0) && (numel(y_temp(in)) == 0) && (numel(x_temp(on)) == 0)
                      && (numel(y_temp(on)) == 0);
  [in, on] = inpolygon(x_temp,y_temp,circle2.Vertices(:,1),circle2.Vertices(:,2));
  circle2_condition = (numel(x_temp(in)) == 0) && (numel(y_temp(in)) == 0) && (numel(x_temp(on)) == 0)
                      && (numel(y_temp(on)) == 0);

  duplicate_condition = 0;
  i = 1;
  while ~duplicate_condition && i<=size(vert(:,1),1)
      dist = norm(vert(i,1:2)-[x_temp(end) y_temp(end)]);
      if dist<0.001
          duplicate_condition=1;
      end
      i = i+1;
  end

  %Update Tree
  if (bound_condition && circle1_condition && square1_condition && circle2_condition &&
                                                                  ~duplicate_condition)

      [in, on] = inpolygon(x_temp,y_temp,goal_circle.Vertices(:,1),goal_circle.Vertices(:,2));
      if ((numel(x_temp(in)) > 0) && (numel(y_temp(in)) > 0)) || ((numel(x_temp(on)) > 0) &&
         (numel(y_temp(on)) > 0))
          find_goal=1;
          k = dsearchn([x_temp' y_temp'], [x_g y_g]);
          x_temp = x_temp(1,1:k);
          y_temp = y_temp(1,1:k);
          theta_temp = theta_temp(1,1:k);
          [x_patch, y_patch, theta_patch, ~, ~] = rectilinear_patch(x_temp(end), y_temp(end), x_g, y_g);
      end


      %Add a primitive to the Tree
      y_tree = [y_tree y_temp];
      theta_tree = [theta_tree theta_temp];


      vert_count = vert_count + 1;
      cost = Astar_FN(vert(index(iteration),1:2), [x_temp(end) y_temp(end)], [x_g y_g],
                                                                  vert(index(iteration),4));
      vert(vert_count,:) = [x_temp(end), y_temp(end), theta_temp(end), cost];

      %Build graph matrix
      A(vert_count,:)=0;
      A(:,vert_count)=0;
      A(index(iteration), vert_count)=1;

  end

  if find_goal
      x_tree = [x_tree x_patch];
      y_tree = [y_tree y_patch];
      theta_tree = [theta_tree theta_patch];

      vert_count = vert_count + 1;
      vert(vert_count,:) = [x_g, y_g, 0, 0];

      %Build graph matrix
      A(vert_count,:)=0;
      A(:,vert_count)=0;
      A(vert_count-1, vert_count)=1;

  end

  iteration = iteration+1;
end
```

Figure 2.11: RRT source code

$A^\star$ algorithm
1 **repeat**
2 find and extract N_{best} from OPEN
3 **if** $N_{\text{best}} = N_g$ **then** exit
4 **for** each node N_i in ADJ(N_{best}) **do**
5 **if** N_i is *unvisited* **then**
6 add N_i to T with a pointer toward N_{best}
7 insert N_i in OPEN; mark N_i *visited*
8 **else if** $g(N_{\text{best}}) + c(N_{\text{best}}, N_i) < g(N_i)$ **then**
9 redirect the pointer of N_i in T toward N_{best}
10 **if** N_i is not in OPEN **then**
10 insert N_i in OPEN
10 **else** update $f(N_i)$
10 **end if**
11 **end if**
12 **until** OPEN is empty

Figure 2.12: $A^\star$ pseudo-code

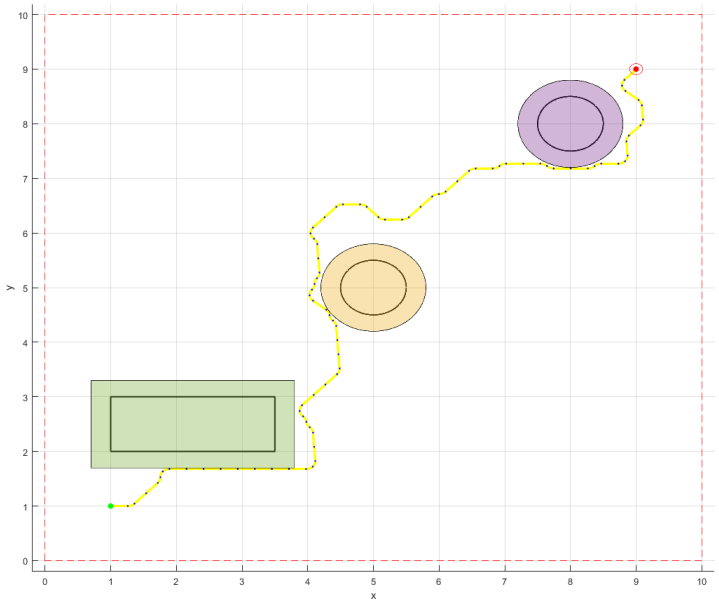

Figure 2.13: Path generated from the graph in fig 2.9

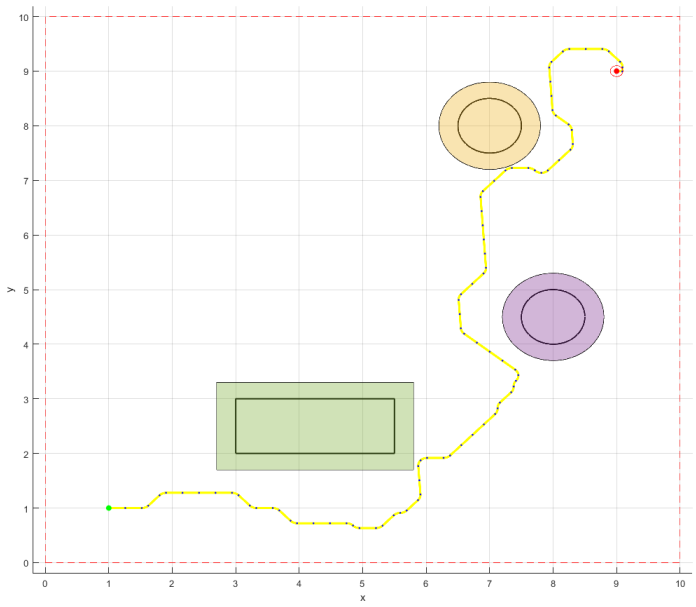

Figure 2.14: Path generated from the graph in fig 2.10

```
while ~isempty(open) && prod(n_best(1,1:2)~=[x_g y_g])
    i_best = find(open(:,4)==min(open(:,4)));
    n_best = open(i_best(1),:);
    temp = vert == n_best;
    [k, ~, ~] = find (prod(temp,2) == 1);
    row = A(k(1),:);
    i_adj = find(row==1);

    if ~isempty(i_adj)
        for i=1:size(i_adj,2)
            adj(i,:)=vert(i_adj(i),:);

            if visited(i_adj(i)) == 0
                visited(i_adj(i))=1;
                T(end+1,1:4) = adj(i,:);
                T(end,5) = k(1);
                open = [open; adj(i,:)];
            else
                if sum(vert(i_adj(i),1:3) ~= [x_s y_s theta_s])==3
                    gN_best = n_best(1,4) - norm([x_g y_g] - n_best(1,1:2));
                    gN_i = adj(i,4) - norm([x_g y_g] - adj(i,1:2));

                    if (gN_best + norm(adj(i,1:2)) - n_best(1,1:2))<gN_i
                        i_T = find(T == adj(i,:));
                        T(i_T,5) = k(1);

                        if isempty(find(open==adj(i,:),1))
                            open = [open; adj(i,:)];
                        else
                            open(find(open==adj(i,:)),4)= Astar_FN (n_best(1:2), adj(i,1:2), [x_g y_g],
                                                                                    n_best(4))
                        end

                    end
                end

            end
        end
    end

    open(i_best(1),:) = [];
    adj = [];

end
```

Figure 2.15: $A^\star$ source code

Chapter 3

Motion Control

3.1 Input/Output Feedback Linearization

The motion control problem use as reference the kinematic model of the robot and take as input the driving and steering velocity v and w to determine the generalized velocities $\dot{q}$. The main motion control problem are:

- Trajectory tracking: the robot must asymptotically track a desired trajectory from an initial configuration q_0;
- Posture Regulation: the robot must asymptotically reach a given posture q_d from an initial configuration q_0;

In this project the Input/Output Feedback Linearization has been used for solving the trajectory tracking problem. The outputs of the linearized feedback (3.1), in the case of the unicycle, are the Cartesian coordinates of a contact point located along the sagittal axis at a distance $|b|$ from the contact point, the sign of b determine if the point is ahead or behind the robot.

$$\begin{aligned} y_1 &= x + b\cos\theta \\ y_2 &= y + b\sin\theta \end{aligned} \tag{3.1}$$

In this project b has been set as the Euclidean distance between two consecutive samples of the trajectory generated by $A^\star$. The time derivative

of y_1 and y_2 are:

$$\begin{bmatrix} \dot{y_1} \\ \dot{y_2} \end{bmatrix} = \begin{bmatrix} \cos\theta & -b\sin\theta \\ \sin\theta & b\cos\theta \end{bmatrix} \begin{bmatrix} v \\ \omega \end{bmatrix} = T(\theta) \begin{bmatrix} v \\ \omega \end{bmatrix} \tag{3.2}$$

b is the determinant of $T(\theta)$, so if $b \neq 0$ the matrix T can be inverted and derive the following input transformation:

$$\begin{bmatrix} v \\ \omega \end{bmatrix} = T^{-1}(\theta) \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = \begin{bmatrix} \cos\theta & \sin\theta \\ \frac{-\sin\theta}{b} & \frac{\cos\theta}{b} \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} \tag{3.3}$$

The control input of the controller have the following form:

$$\begin{aligned} u_1 &= \dot{y}_{1d} + k_1(y_{1d} - y_1) \\ u_2 &= \dot{y}_{2d} + k_2(y_{2d} - y_2) \end{aligned} \tag{3.4}$$

where k_1 and k_2 are strictly positive. This assumption ensures exponential convergence to zero of the tracking error. Using this controller, the robot orientation is not controlled because this scheme does not use the orientation error but only the position error. The file `main.cpp` at the lines 149-182 provide an implementation of this controller used to track the trajectory achieved by the $A^\star$ algorithm applied on the Tree generated by the RRT algorithm.

3.2 Posture Regulation

The Input/Output Feedback Linearization does not control the orientation, hence to achieve perfectly a goal configuration it has been necessary to include a posture regulation at the end of the tracking phase.

```
//INPUT/OUTPUT LINEARIZATION
   while (i<(x_path.size()-1)){
       x_diff = x_path.at(i+1) - x_path.at(i);
       y_diff = y_path.at(i+1) - y_path.at(i);

       b = sqrt(pow(x_diff,2) + pow(y_diff,2));

       y1 = x_real + b*cos(theta_real);
       y2 = y_real + b*sin(theta_real);

       y1d_dot = lin_vel.at(i)*cos(theta_path.at(i)) - ang_vel.at(i)*b*sin(theta_path.at(i));
       y2d_dot = lin_vel.at(i)*sin(theta_path.at(i)) + ang_vel.at(i)*b*cos(theta_path.at(i));

       u1 = y1d_dot + k1*(x_path.at(i+1) - y1);
       u2 = y2d_dot + k2*(y_path.at(i+1) - y2);

       v = u1*cos(theta_real) + u2*sin(theta_real);
       w = -u1*sin(theta_real)/b + u2*cos(theta_real)/b;


       pos_error_temp = sqrt(pow(x_real - x_path.at(i),2) + pow(y_real - y_path.at(i),2));
       pos_error.push_back(pos_error_temp);

       vel_ctrl(v, w);
       i++;
       rate.sleep();
   }
```

Figure 3.1: Implementation code of the Input/Output Feedback Linearization controller

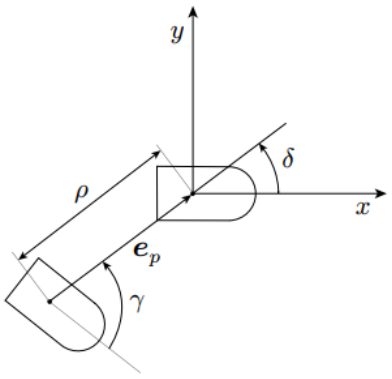

Figure 3.2: Definition of the polar coordinate for the unicycle

To simplify the problem is convenient to express the unicycle configura-

tion in polar coordinate:

$$\begin{aligned} \rho &= \sqrt{x^2 + y^2} \\ \gamma &= Atan2(y, x) - \theta + \pi \\ \delta &= \gamma + \theta \end{aligned} \tag{3.5}$$

At this point the kinematic model can be expressed using this coordinate and the feedback control can be defined as follow:

$$\begin{aligned} v &= k_1 \rho \cos\gamma \\ \omega &= k_2\gamma + k_1 \frac{\sin\gamma\cos\gamma}{\gamma}(\gamma + k_3\delta) \end{aligned} \tag{3.6}$$

Under this control action, with $k_1 > 0$ and $k_2 > 0$, the system converge asymptotically to the desired configuration that coincide, unless of traslations or rotations, with the origin $[\,\rho \quad \gamma \quad \delta\,]^T = [\,0 \quad 0 \quad 0\,]$. This kind of controller has been implemented in the same file of the previous one (`main.cpp`) at the lines 193-216 (fig. 3.3).

```
1  //POSTURE REGULATION
2      while( (abs(pos_error_temp)>0.05) || ((abs(ang_error_temp)>0.15))){
3          x_diff = x_real - x_g;
4          y_diff = y_real - y_g;
5
6          rho = sqrt(pow(x_diff,2) + pow(y_diff,2));
7
8          gamma = atan2(y_diff, x_diff) - theta_real + M_PI;
9          delta = gamma + theta_real;
10
11         v = k1*rho*cos(gamma);
12         w = (k2*gamma)+k1*((sin(gamma)*cos(gamma))/gamma)*(gamma + k3*delta);
13
14         pos_error_temp = rho;
15         ang_error_temp = theta_g - theta_real;
16         pos_error.push_back(pos_error_temp);
17         ang_error.push_back(ang_error_temp);
18
19         vel_ctrl(v, w);
20         rate.sleep();
21     }
```

Figure 3.3: Implementation code of the posture controller

Chapter 4

Tests

4.1 Implementation

In this section will be described the tools used to develop this project. The main software have been Matlab and ROS. The former was used to write the code of the RRT and $A^\star$ because it offer a great flexibility and powerful functions for working with the matrix and data structure with a lot of samples. The latter give the possibility to easily communicate with the physics engine simulator, Gazebo, and elaborate the data from it for executing the controlling action. In fig. 4.1 is possible to see a schema of the workflow implemented for this project. The first step is to define the motion problem: start and goal configuration, the working area dimension, the maximum number of iterations, the goal range and the robot radius. All the parameters used in the tests that will be show in this report can be found in the table 4.1.

height	$10m$
width	$10m$
x_start	$1m$
y_start	$1m$
θ_start	$0rad$
x_goal	$9m$
y_start	$9m$
θ_start	$0rad$
max iteration	10000
goal range	$0.1m$
robot radius	$0.3m$

Table 4.1: Parameters of the Motion Problem

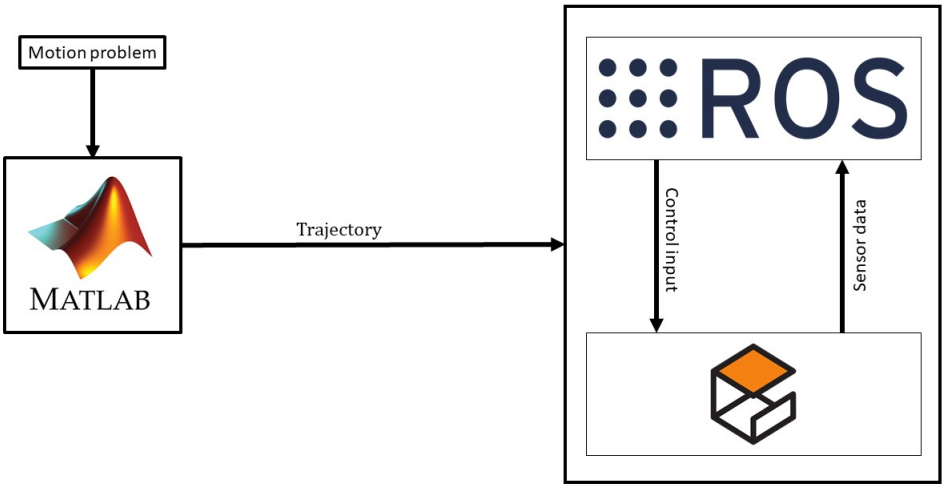

Figure 4.1: Schema of the workflow of the project

The solution generated by RRT and $A^\star$ is a sequence of samples where each of these represent a robot configuration until the goal. Furthermore, the set of driving and steering velocities is generated for each path sample. This information are saved in text file and they will be load in ROS for running the controllers.
At this point ROS use the trajectory and velocity data and the measurement

provided by the sensor simulated in Gazebo to execute the controller based on the Input/Output Feedback Linearization for first and the Posture Controller at the last. The desired value of the solution path and the odometry data, provided by the Gazebo sensors, are constantly compared to achieve the position and orientation error. These errors are used to compute the control action moment by moment and after for analyzing the general performance. In order to achieve the best control performance several attempts have been made and at the end the best gain value are:

- Input/Output Feedback Linearization controller
  - $k_1 = 2.5$
  - $k_2 = 1$
- Posture Controller
  - $k_1 = 2$
  - $k_2 = 1$
  - $k_3 = 3$

In the next sections will be presented the result of the simulation tests.

4.2 Test 1

The first test has been executed in the first scenario (the fig. **??** is repeated here in fig. 4.2). The aim of the simulation is to test the robot skill into an environment that block the line that join directly the start point and the goal point and also a possible alternative way. In fig. 4.3 and 4.4 are shown respectively the results of the RRT and $A^\star$ algorithm.

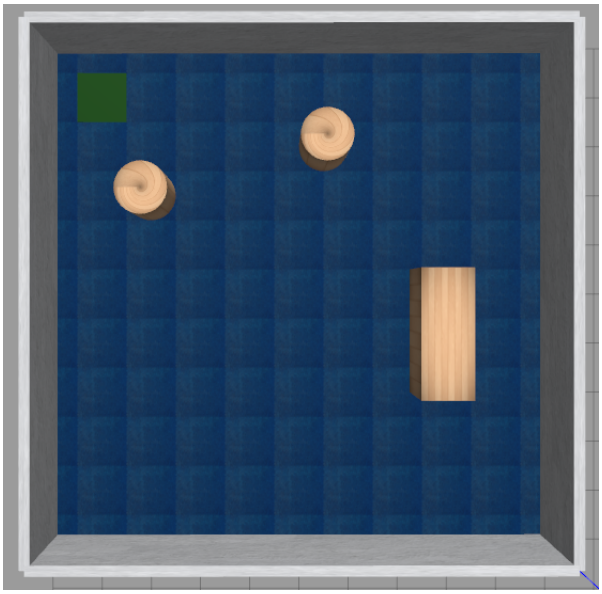

Figure 4.2: First simulation scene

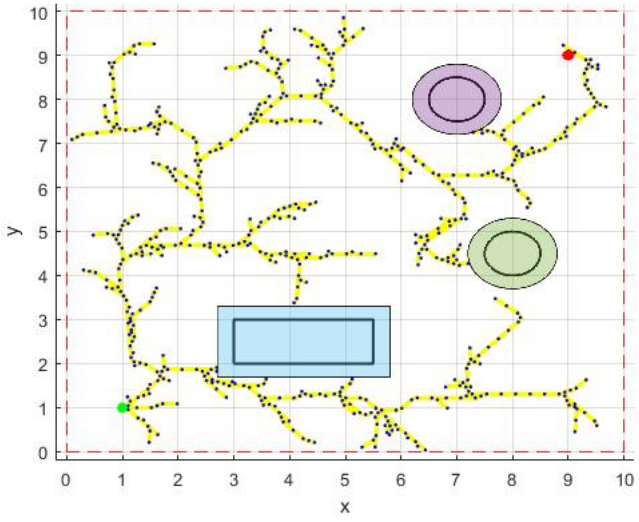

Figure 4.3: RRT solution in the first scenario

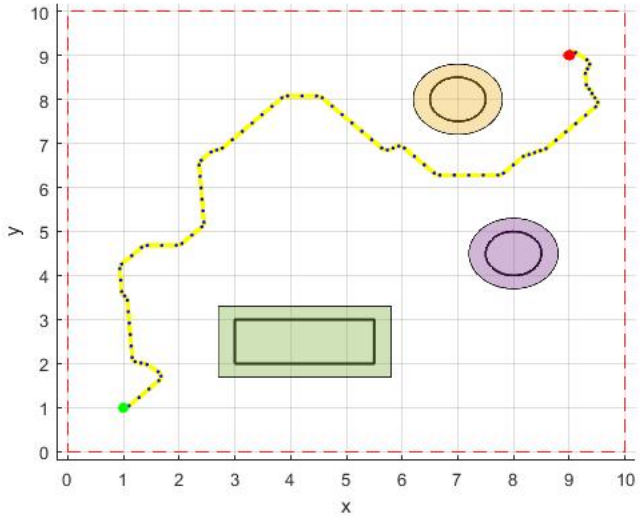

Figure 4.4: $A^\star$ solution in the first scenario

In fig. 4.5 is possible to see the trend of the position error and its mean (red line). It take in count both the error on x and y axis. This graph show that the controller works fine, the peak is equal to $0.33m$ and the mean to 0.07, so this value ensures a good execution of the task without any risk of collision with the obstacles. Furthermore, the posture controller act to reduce the position error and bring the robot in correct position with an error less than $0.01m$. In fig. 4.6 is shown the orientation error that have at a certain point a jump because the robot do a complete rotation. However is important to highlight the good trend of the error that arrest its value around $0.15rad$.

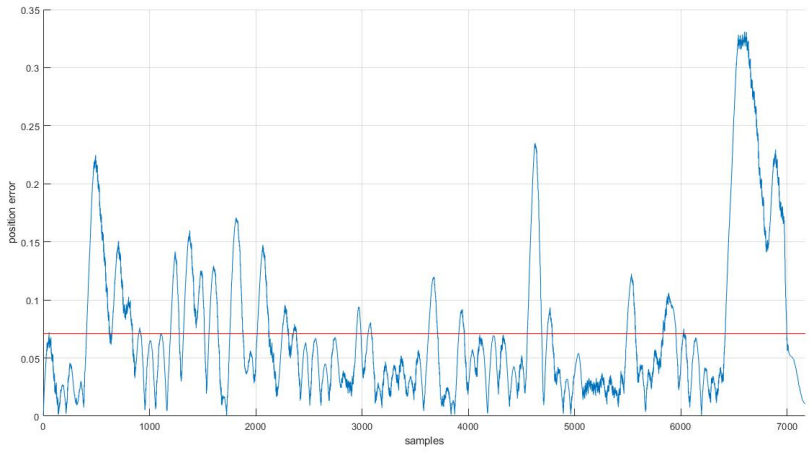

Figure 4.5: Position error for the test 1

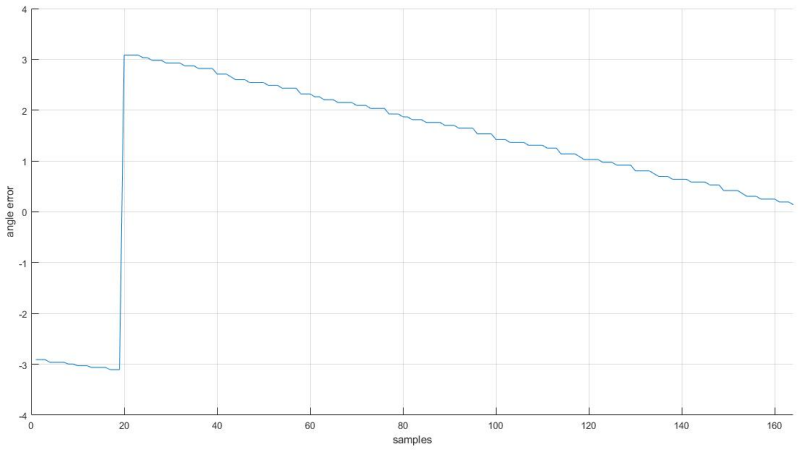

Figure 4.6: Orientation error for the test 1

In fig. 4.7 and 4.8 are shown respectively the driving and steering velocity that the robot assume during the simulation. Both the velocity are limited to their maximum and minimum value, these is $\pm 0.26 m/s$ and $\pm 1.82 rad/s$ (tab. 1.1). Analyzing the graphs, point out that while the trend of the driving velocity is good, the angular velocity graph suggest that the robot oscillate around the trajectory for all the time. It try to keep center of the path but obviously this is impossible because two samples will never be perfectly identical, then the control action compensate the error with this behaviour.

4.3 Test 2

The second test is simulated in the second scenario, the fig. 2.2 is repeated here in fig. 4.9. In this case the purpose is to completely block the direct way from the start point to the goal point and force the robot to pass near the wall or near the obstacle. The RRT solution and the path finding by $A^\star$ are in fig. 4.10 - 4.11. As is possible to see, the solution trajectory pass near the rectangular-shaped and the circle obstacle.

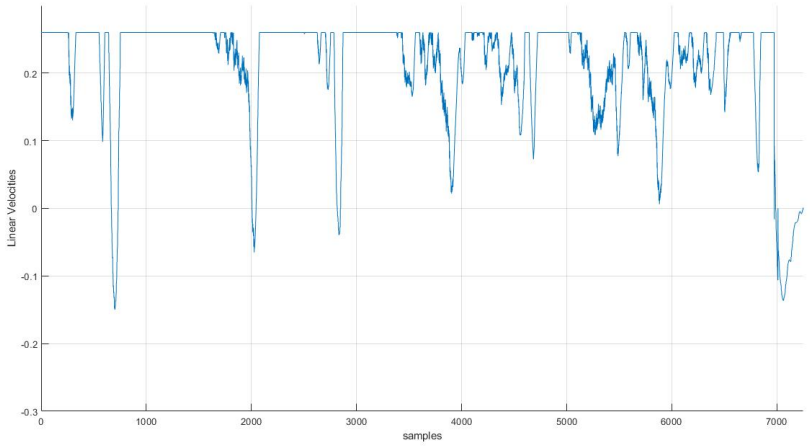

Figure 4.7: Driving velocity during the test 1

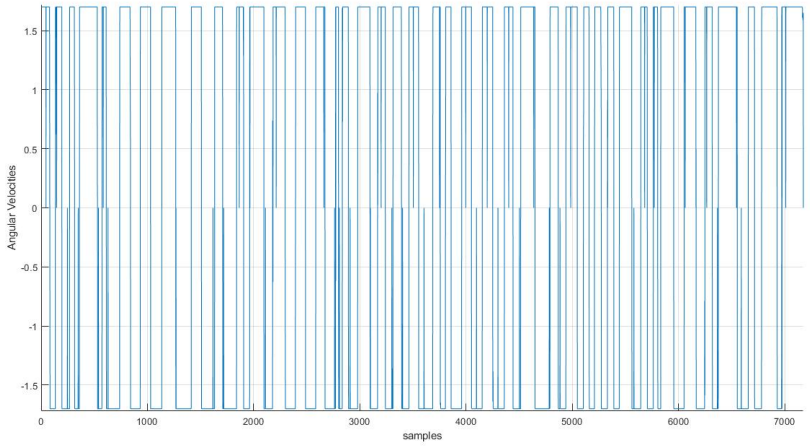

Figure 4.8: Angular velocity during the test 1

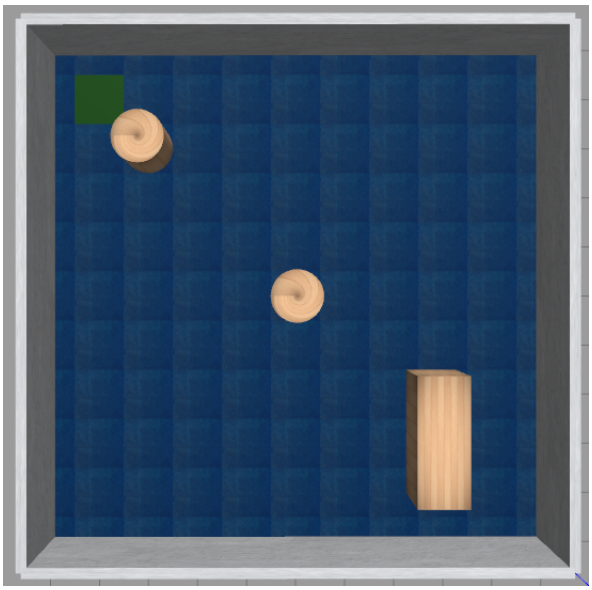

Figure 4.9: Second simulation scene

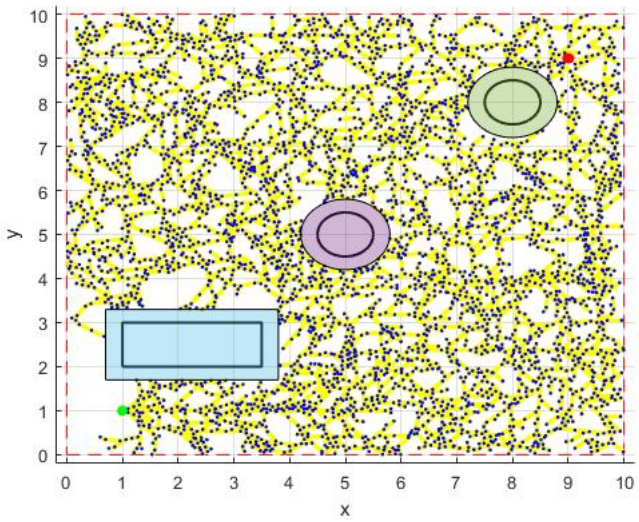

Figure 4.10: RRT solution in the second scenario

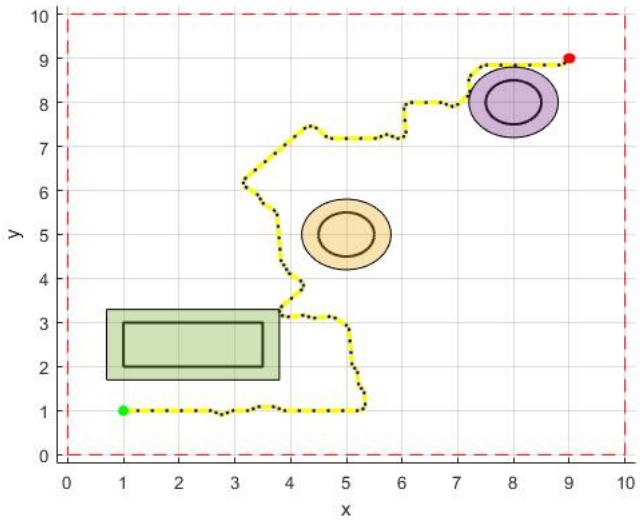

Figure 4.11: $A^\star$ solution in the second scenario

In this case the position error (fig. 4.12) has a mean a lit of bit greater then before $0.077m$, but the peak do not pas the $0.3m$. This value ensure that robot even if it pass very close to the obstacle, the collision is avoided and the trajectory can be completed without any problem. Furthermore, in the last part of the graph is possible to see the exponential decrease of the position error due to the posture controller when the robot reach the surrounding of the goal point. The same trend point out for the orientation error (fig. 4.13).

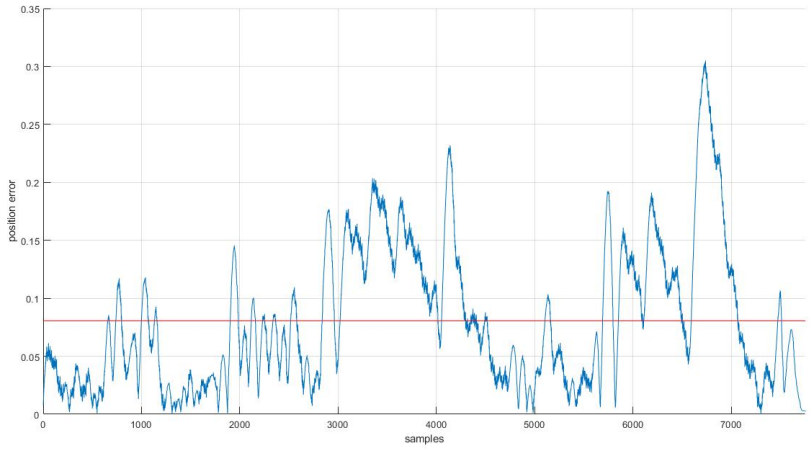

Figure 4.12: Position error for the test 2

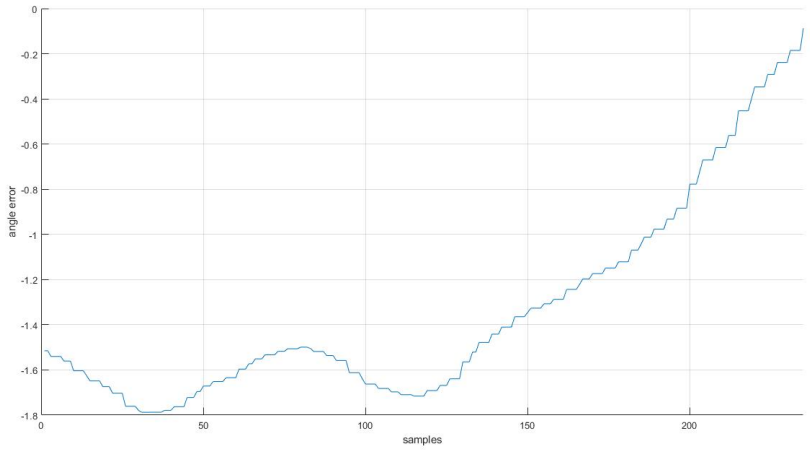

Figure 4.13: Orientation error for the test 2

Fig. 4.14 and 4.15 are respectively the driving and steering velocity. They are quite similar to the previous test, then also in this case the robot oscillate around the trajectory.

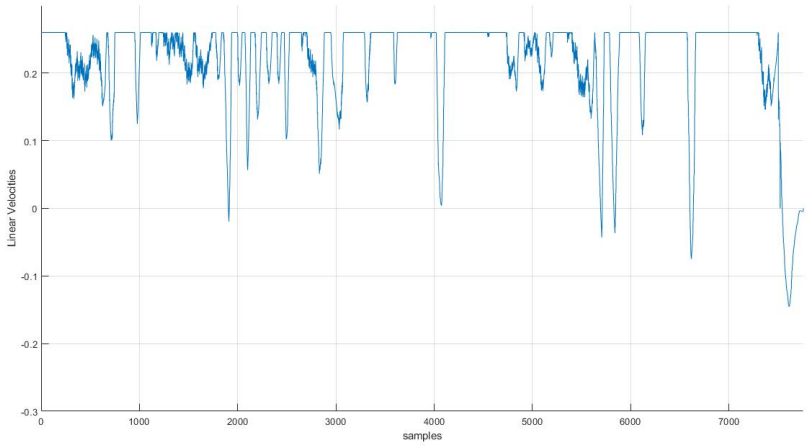

Figure 4.14: Driving velocity during the test 2

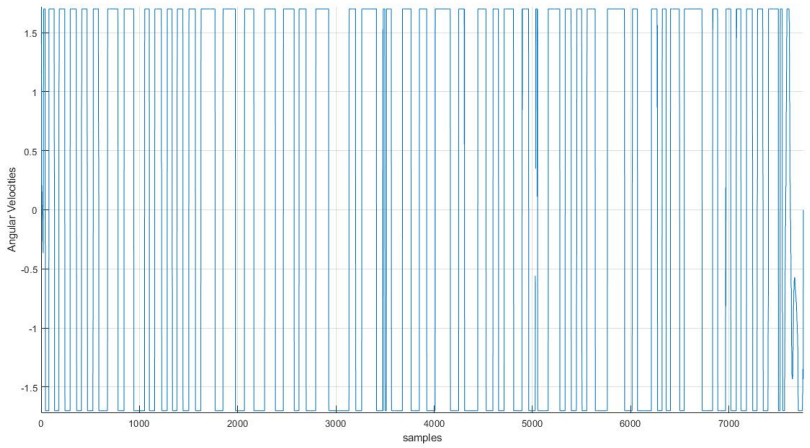

Figure 4.15: Angular velocity during the test 2

Conclusion

In this report has been shown a possible solution to the motion planning problem for a differential-drive robot. In the chapter have been analyzed the RRT and $A^\star$ algorithm useful for creating a valid path from a start configuration to a goal configuration and two control technique, Input/Output Feedback Linearization and Posture Regulation, to solve the motion control problem. The tests presented in this report have shown how the combination of these two controller can produce quite good results: the norm of error remain limited for all the trajectory under acceptable value and the orientation error tend to zero one that the goal is reached.

Bibliography

[1] L. Villani B. Siciliano L. Sciavicco and G. Oriolo. *Robotics, Modelling, Planning and Control*. Springer, 2008. ISBN: 9781846286414.