DIETI UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II
DIPARTIMENTO DI INGEGNERIA ELETTRICA E TECNOLOGIE DELL'INFORMAZIONE
SCUOLA POLITECNICA E DELLE SCIENZE DI BASE

TECHNICAL PROJECT

PROJECT 1

POWERBOT AUTONOMOUS NAVIGATION

IN A LOGISTIC KNOWN ENVIRONMENT BASED ON RRT* ALGORITHM

Candidate
Antonio Caccavale
P38/18

Academic Year 2020/2021

Contents

List of Figures

Introduction

This technical project describes a robotic application implemented in ROS to solve a realistic problem set in a logistic environment and simulated in Gazebo. Particularly, chapter 1 presents robot typology, its modeling and the one of the scenario. Then, in chapter 2, the mission is stated: preliminary features to the given task are reported so that the final system architecture solution is described in the last section. This system is composed by three main players and each one has a chapter is enterely dedicated: the planner in chapter 3, the navigator in chapter 4 and the controller in chapter 5. Finally chapter 6 is about the final simulations, by showing all the plots related to the video attached to the project. At this regard, the final video is 2.3x speed up comparing to the plot time. Matlab software provided these figures importing the recorded rosbags for each simulation. All the additional image elaborations have been obtained with Gimp software.

Chapter 1

Modeling

1.1 The differential drive robot

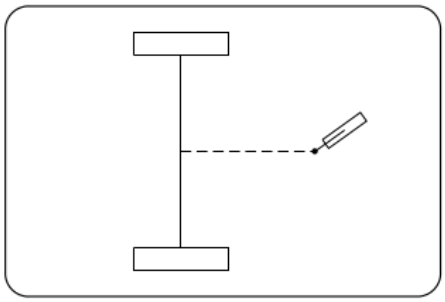

Figure 1.1: Differential drive robot

The differential drive robot has two fixed motion wheels with a common axis of rotation and one or more smaller caster wheels to keep the robot statically balanced. The two fixed wheels are separately controlled, in that different values of angular velocity may be arbitriarly imposed, while the caster wheels are passive [1]. This is a non-holonomic system for the presence of a pure rolling constraint for each motion wheel, expressed in the Pfaffian form by:

$$\dot{x}\, sin(\theta) - \dot{y}\, cos(\theta) = 0 \tag{1.1}$$

where x and y are the Cartesian coordinates in the plane, coincident with the ground, of the middle point between the two fixed wheels, while θ is the yaw angle which refers to robot orientation. So we can consider:

$$\boldsymbol{q} = \begin{bmatrix} x & y & \theta \end{bmatrix}^T, \tag{1.2}$$

as vector of generalized coordinates, whose time-derivative give us the kinematic model of the robot. By assuming v and ω as the heading velocity and the stearing one respectively, the kinematic model is:

$$\dot{\boldsymbol{q}} = \boldsymbol{G}(\boldsymbol{q})\,\boldsymbol{u} = \begin{bmatrix} cos(\theta) \\ sin(\theta) \\ 0 \end{bmatrix} v + \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \omega, \tag{1.3}$$

where

$$v = \frac{\rho(\omega_R + \omega_L)}{2} \qquad\qquad \omega = \frac{\rho(\omega_R - \omega_L)}{d}. \tag{1.4}$$

are the kinematic model inputs. Moreover, we can notice that the final inputs to the robot are the wheels speed, expressed in Eq.(1.4) as ω_R and ω_L, for the right wheel and for the left one respectively. Then d is the distance among them, while ρ is the common radius.

1.2 Robot model

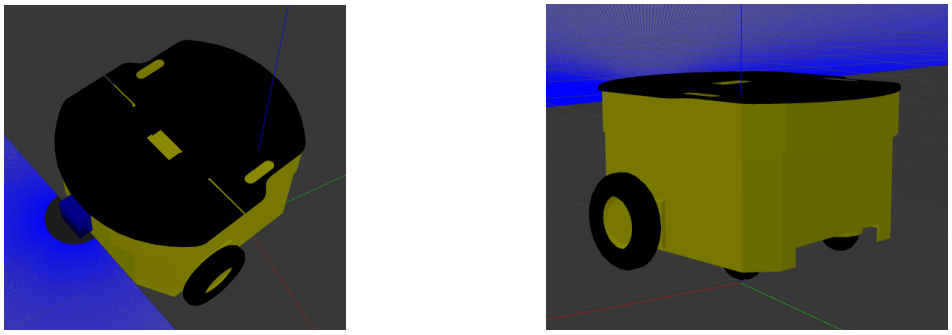

Figure 1.2: Robot model in Gazebo

Robot choice fell on the Omron Adept MobileRobots PowerBot [2], whose model has been imported [3]. To spawn the robot in the simulation environment, all the *.xacro* files are contained in the *urdf* folder. The typical modularity and reusability programming principles are preserved by using *left* and *right* suffixes attributes associated to opposite sides of the structure. So the total structure is composed from the top down by:

- 1 Top plate: Black surface on which an arm plate can be eventually fixed. The maximum dimensions are approximately represented by a $0.71 \times 0.83 \times 0.006\,m$ box.

- 1 Chassis: Yellow body of the robot. The maximum dimensions are approximately represented by a $0.61 \times 0.81 \times 0.42\,m$ box.
- 1 Sensor Link: $0.1 \times 0.1 \times 0.05\,m$ box fixed on the front side of the chassis. The position choice determines the reference for sensors.
- 2 Motion Wheels: Fixed wheels with a $0.133\,m$ radius and $0.065\,m$ thick.
- 2 Caster Wheels: Passive spherical wheels with $0.065\,m$ radius.

With these choices, it is possible to consider $\rho = 0.133\,m$ and $d = 0.61\,m$ for the kinematic model inputs in Eq.(1.4). Moreover, in Fig.1.3 it is possible to visualize the built structure, after the generation of the corrisponding *.urdf* file, thanks to *urd_to_graphiz* command.

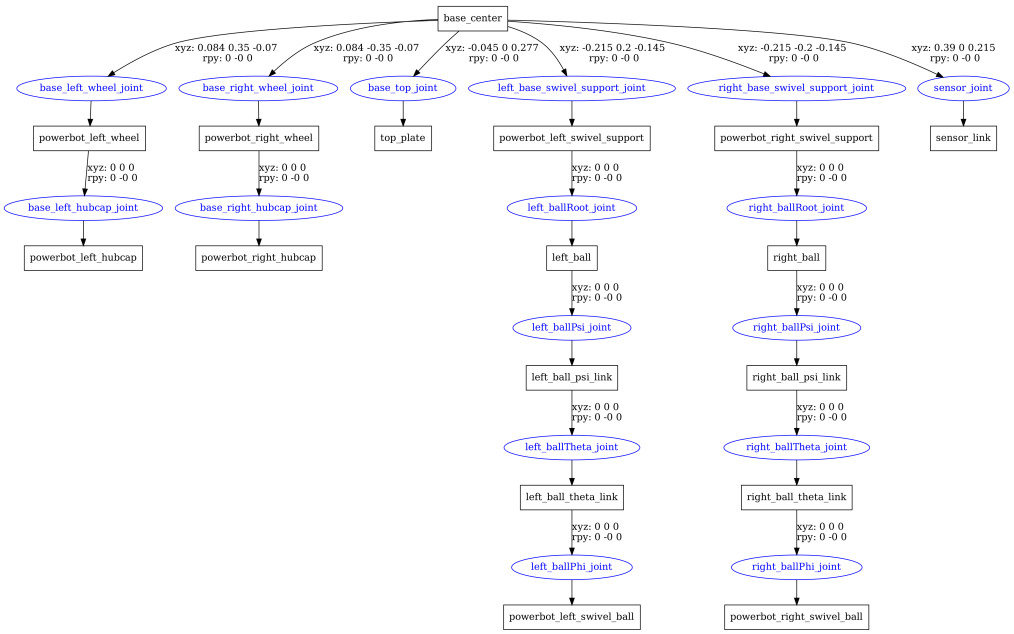

Figure 1.3: Structure of the robot from the urdf file

In addition, to performe the proposed tasks the robot is equipped with:

- 1 RGB camera: implemented by *libgazebo_ros_camera.so* plugin.
- 1 LIDAR sensor: implemented by *libgazebo_ros_laser.so* plugin.
- 1 IMU sensor: implemented by *libgazebo_ros_imu.so* plugin.

Finally the *libgazebo_ros_control.so* plugin ensures to control the the differential drive robot, after the inclusion of two different trasmissions to actuate each wheel with a own motor.

1.3 Scenario

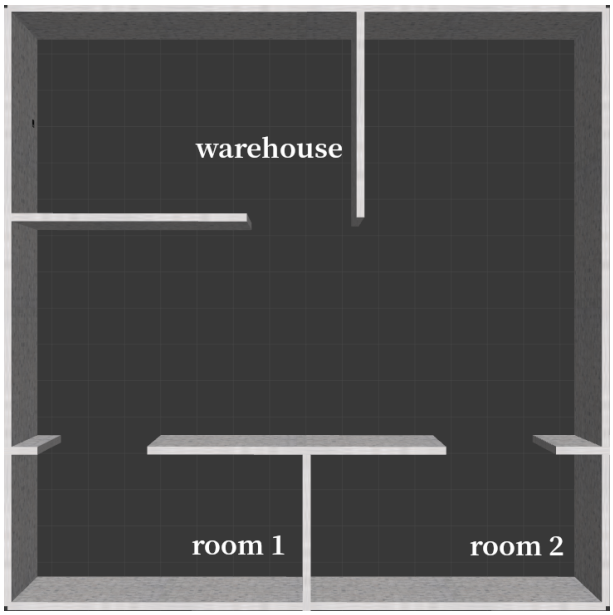

Figure 1.4: Scenario in gazebo

Covering a $15 \times 15\,m^2$ area, the scenario is a logistic closed environment composed by three rooms: a warehouse and two destination rooms. The walls, imported as Gazebo models, are $0.2\,m$ thick and are disposed so that each doorway is about $2.75\,m$ wide. Moreover, we state that the world frame corresponds to the one of the simulation environment, in the origin of the scenario. Respect to it a $0.2 \times 0.2\,m^2$ AR marker sides in $\boldsymbol{p} = \begin{bmatrix} 5.0 & 7.4 & 0.5 \end{bmatrix}^T m$ and $\boldsymbol{\eta} = \begin{bmatrix} 1.57 & 4.71 & 0 \end{bmatrix}^T rad$, in the warehouse.

Chapter 2

Mission

Robot mission consists in the simulation of an object transportation from the warehouse to a destination room on the basis of the AR marker detection. During the motion, since it is not possible to obtain directly the robot state from the sensors, we can refer to a procedure of odometric localization by designing a *odom* node. Moreover, this motion takes place in a closed scenario, so that before starting we can solve a mapping problem. In this way SLAM consists only in robot localization on a given map.

2.1 Odometric localization

By considering the kinematic model of the robot in Eq.(1.3) and a time interval $\left[t_k, t_k + T_s\right]$, where $T_s = 5\,ms$ is the choosen sampling time, the value of the robot configuration $q(t_{k+1})$ can be retrieved thanks thanks to the 2^{nd} order Runge-Kutta integration:

$$\begin{cases} x_{k+1} & = x_k + v_k\, T_s \cos(\theta_k + \frac{\omega_k\, T_s}{2}) \\ y_{k+1} & = y_k + v_k\, T_s \sin(\theta_k + \frac{\omega_k\, T_s}{2}) \\ \theta_{k+1} & = \theta_k + \omega_k\, T_s. \end{cases} \tag{2.1}$$

To obtain kinematic model inputs the two angular displacements

$$\Delta\phi_R = \omega_R\, T_s \qquad\qquad \Delta\phi_L = \omega_L\, T_s\,, \tag{2.2}$$

are retrieved from a callback to the $\backslash joint\_states$ topic, that provides the two wheels speed. So,

$$v_k = \frac{\rho}{2\, T_s}(\Delta\phi_R + \Delta\phi_L) \qquad\qquad \omega_k = \frac{\rho}{d\, T_s}(\Delta\phi_R - \Delta\phi_L) \tag{2.3}$$

are evaluated to obtain the desired state estimation in Eq.(2.1). Despite this, to minimize approximation errors, the estimation of robot orientation is entrusted to an IMU sensor, so that the implemented technique is not completely passive. The resulting *nav_msg::Odometry* message is published by the *odom* node on the $\backslash odom$ topic at $200\,Hz$ rate.

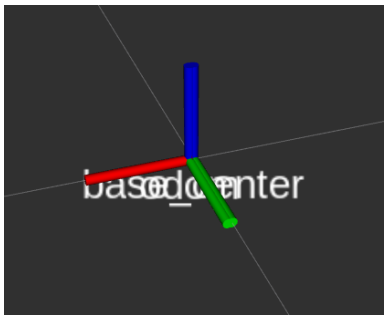

Figure 2.1: Overlap between *base_center* and *odom* frames in RViz

2.2 Mapping

Before starting to move, robot needs to know about the sorrounding environment. To reach this goal, a *key_teleop* node for each wheel has been designed to explore the logistic environmnet, so that a map can be obtained thanks to the *gmapping* package. It contains the *slam_gmapping* node that needs of *laser/scan* data provided by the LIDAR sensor, the *base_center* frame as reference of the robot structure and the *odom* one. RViz graphical interface allows us to visualize the generating map. Then, *map_server* node provides the *map_saver* command that builds a *map.yaml* file and the *map.pgm* image of the final generated map. By the same node, the result can be loaded into a $\backslash map$ topic as a *nav_msgs::OccupancyGrid* message. It contains two fields:

- MapMetaData info: available informations contained into the *map.yaml* file. They are the key to map pixels of *map.pgm* image in Cartesian coordinates of the workspace $W \equiv \mathbb{R}^2$.

```
1 image: map.pgm
2 resolution: 0.050000
3 origin: [-13.800000, -17.000000, 0.000000]
4 negate: 0
5 occupied_thresh: 0.65
6 free_thresh: 0.196
7
```

Figure 2.2: *Generated map.yaml* file content

- int8[] data: The map data, in row-major order, starting with (0,0). Occupancy probabilities are in the range [0,100]. Values lower than *free_tresh* (Fig.2.2) are related to "white" free regions, greater than *occupied_tresh* are related to "black" occupied regions, while -1 is a "grey" unexplorated zone, according to [4].

Starting from the result provided by the *map_server*, an additional image elaboration is made by using *Gimp* software. This is due to the need of a "fake" map to solve the planning problem: while the thickness of some walls has been exaggerated to be sure of collision avoidance, the doorways have been restricted to direct paths in desired directions. The final result is shown in Fig.2.3.

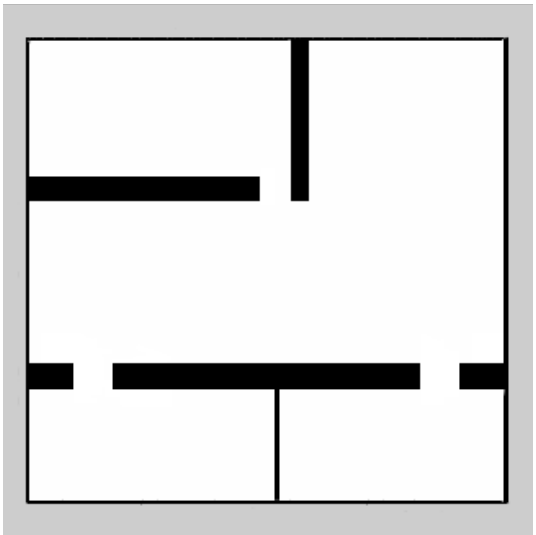

Figure 2.3: Final *map.pgm* elaboration

2.3 Localization

Once obtained the map of the environment, we can localize the robot by using a probabilistic localization system as *amcl*. It implements the adaptive (or KLD-sampling) Monte Carlo localization approach, which uses a particle filter to track the pose of a robot against a known map. The estimated pose in the map is published into *amcl_pose* by means of a *geometry_msgs::PoseWithCovarianceStamped* message. It needs of *map*, *tf*, *initial_pose* and *laser\scan* topics [4].

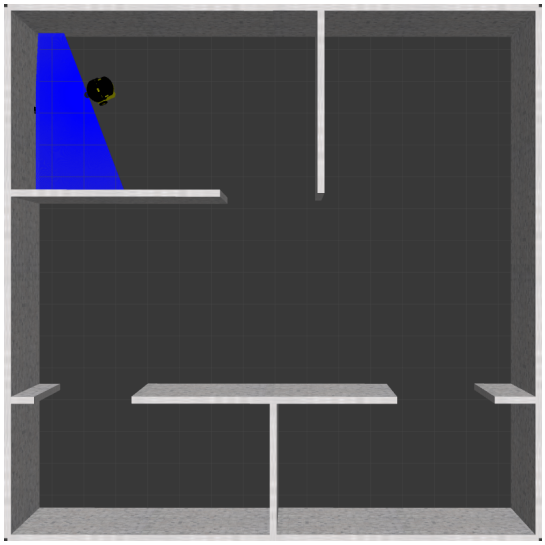

Figure 2.4: Robot motion in Gazebo

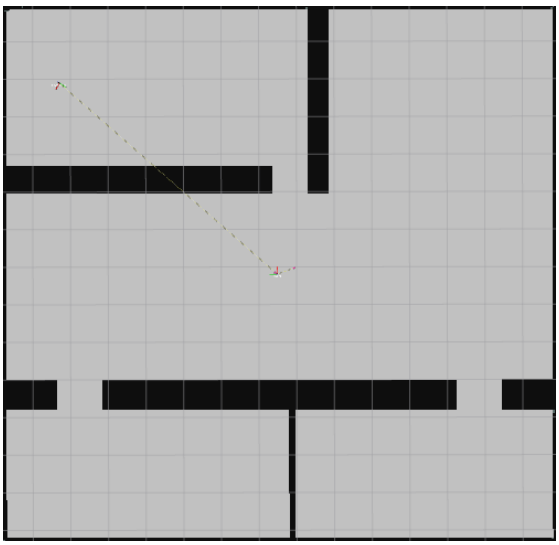

Figure 2.5: Localization in RViz

In RViz it is possible to visualize how the *base_center* frame is moving respect to the *odom* frame by considering the *map* frame as fixed, as Fig.2.5 shows, while simulation works in Fig.2.4.

2.4 System architecture

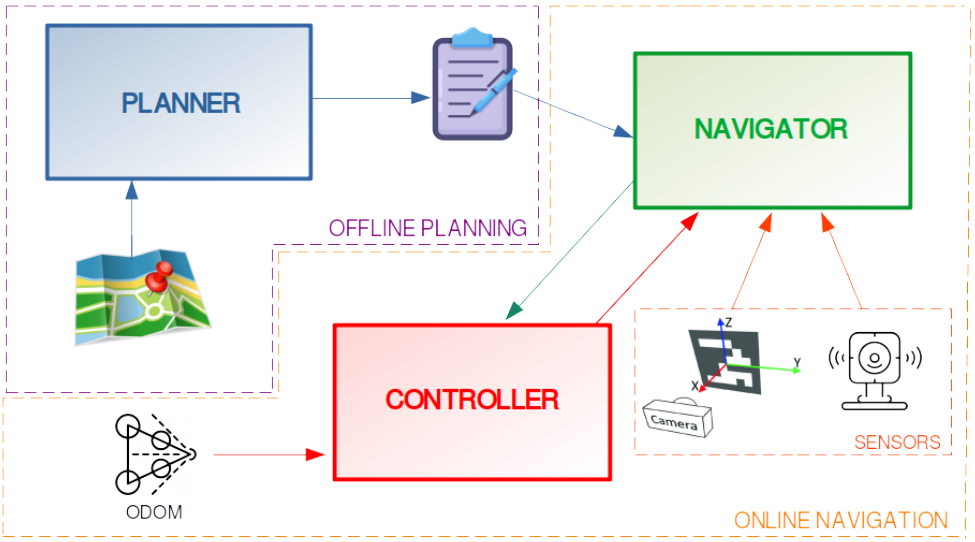

Figure 2.6: System architecture of mission solution

Solution to the given mission consists in the development of a system composed by three main players: the planner, the navigator and the controller. By obtaining informations about the environment from the map, the planner elaborates paths according to a probabilistic planning method. Once the work is finished, all the paths are reported into a *path.yaml* file. As opposite to this offline procedure, navigator works online to provide the current path the controller has to execute. Once initialized the navigation with the reaching of the warehouse, navigator has to detect the AR marker identifier to know what the next destination is. To do that it needs to read the *path.yaml* file and to get information from both RGB camera and LIDAR sensors. Moreover the controller, that receives the odometric localization, elaborates the control law allowing the robot to reach each waypoint of the path selected by the navigator. Then the controller responds with the number of paths sucessfully completed.

Chapter 3

Planning

3.1 Offline Planning

Assuming a closed scenario, the planner node has been implemented to run offline. All the functions needed for the proposed algorithms have been designed by scratch, so that it is possible to higlight differences and advantages of different versions. Particularly the probabilistic method adopted is the Rapidly-exploring Random Tree algorithm. The different versions we refer to are: RRT, bidirectional RRT, RRT^* and bidirectional RRT^*. First of all, the subscription of the map as a *nav_msgs::OccupancyGrid* message is needed, so that the conversion from *map.pgm* pixels to Cartesian coordinates takes place thanks to *readMapData()* function, a common element for all the versions. Since we expect algorithms results as sequences of points, the implementation of a *visualization* node makes it possible the rapresentation of the generated paths in RViz, very useful in planning algorithm debug especially. As soon as the map is read by the planner node, the set of points that forms C_{obst} is load on a *obstacle.yaml* file. In this way visualization is entrusted to *visualization_msgs::Marker* ROS messages: red *POINTS* for C_{obst}, while red *POINTS* and green *LINE_STRIP* for generated paths.

3.1.1 RRT and bid-RRT algorithms

According to motion planning problem, given a starting configuration q_s belonging to the C_{free} space, a final configuration q_g has to be reached. The implementation of the RRT probabilistic algorithm is described by considering the *RRT-node struct* as follow:

$$\begin{cases} int\ nodeID; \\ double\ posX; \\ double\ posY; \\ int\ parentID; \\ double\ cost; \end{cases} \tag{3.1}$$

So a final path will result as a *RRT-tree*, a vector of this structure. The algorithm starts from the desired q_s configuration and consists in generating a random configuration (thanks to the *generateRandNode()* function) whose distance respect to the nearest node of the tree is calculated (thanks to the *checkNearNode()* function). If it is lower than a choosen treshold of $\delta = 0.20\,m$, the node becomes a candidate to be included into the tree. Otherwise the choosen node is that along the straight line joining them at δ distance (thanks to the *checkNewNode()* function). So if the new candidate node has not to belong to C_{obst} space, its *nodeID* is set by an increasing counter, its position is set in *posX* and *posY*, while its *parentID* is the identifier of the nearest node of the tree. Finally, the tree can be pushed back by the new node. Iterations stop when the last included node of the tree is a neighbour of the q_g.
At this point we need a function which provides the final path, given the whole built tree. This task is entrusted to the *checkFinalPath()* function that, starting from the last node included in the vector, rebuild all the path sequence by logging in each *parentID* field, till the q_s.
Besides, a bidirectional version of this algorithm is proposed too. Now, considering the same implemented algorithm, starting from q_s a direct tree is generated. At the same time, starting from q_g a reverse one too. Obiouvsly, iterations stop when the the two respective last nodes are in a neighborhood.

Results

To test the two algorithms, in a workspace $W \equiv \mathbb{R}^2$ is possible to suppose a starting initial configuration $(0; 0)$ and a goal one $(-5; -6)$ respect to a fixed frame in the start, as Fig.3.1 and Fig.3.2 show.

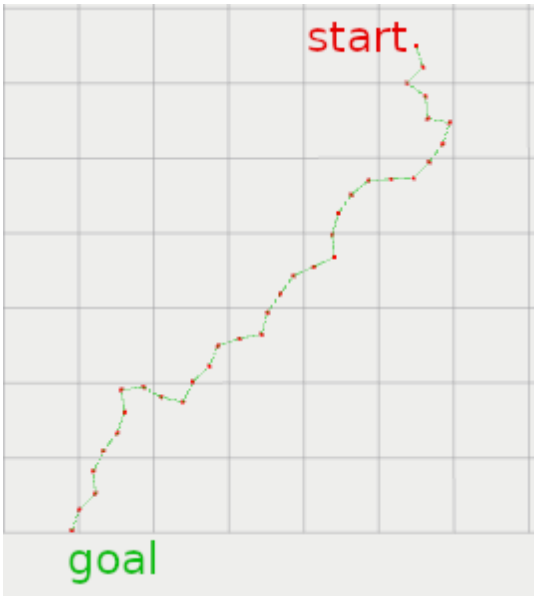

Figure 3.1: *RRT* path

Figure 3.2: *bid-RRT* path

It is possible to notice two different policies to reach the final destination: *RRT-tree* reaches a neighborhood of the goal (Fig.3.1), while each bidirectional *RRT-tree* search for a common one (Fig.3.2). This causes an extension of path lenght (13.76 instead of 10.17 m), but a high reduction in the number of iterations (327 instead of 2524), as Fig.3.3 and Fig.3.4 show. It would seem necessary to specify that the number of iterations correspond to each final tree size, certainly not equivalent to the number of points of each final path. So it is possible to point out that in bid-RRT algorithm the direct tree and the reverse one make the same number of iterations because of the no presence of obstacles.

Figure 3.3: *RRT* path info

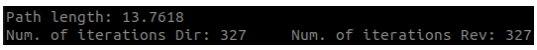

Figure 3.4: *bid-RRT* path info

3.1.2 RRT* and bid-RRT* algorithms

In the proposed algorithms the only *RRT-node* field that has not been considered is the *cost*, in the *struct* (3.1). This happens beacause the *checkFinalPath()* function doesn't care about it. So an upgraded version is represented by making the algorithm capable to reject the inclusion of a parent if there is a more convenient node in a given search neighborhood. If we call j and i the new node and the nearest of the tree identified by the *checkNewNode()* function respectively, we can consider:

$$j.cost = i.cost + getDistance(j, i), \tag{3.2}$$

where *getDistance()* function returns the Euclidean norm of the postions of the nodes. In this way *checkFinalPathStar()* function selects the parent with the minimum cost for each node, given a search radius (set as $\delta = 0.30\,m$). Thus, RRT* minimizes the total path length as a cost function.
Obiouvsly to obtain the optimized bidirectional version bid-RRT* the direct tree and the reverse one both refer to *checkFinalPathStar()* function.

Results

The same starting and goal positions of the previous versions (Fig.3.1 and Fig.3.2) are assumed. Now it is possible to notice a steeper trend in reaching respectively final destinations by final paths in both cases, as Fig.3.5 and Fig.3.6 show.

Figure 3.5: *RRT** path

Figure 3.6: *bid-RRT** path

This is testified by length reduction for both optimized paths in comparison to previous versions: for *RRT** 9.34 m instead of 10.17 m, for *bid-RRT** 12.49 m instead of 13.76 m, as Fig.3.7 and Fig.3.8 show.

Figure 3.7: *RRT** path's info

Figure 3.8: *bid-RRT** path's info

3.1.3 Final choice and results

An *obstacleFound()* function allows the planning algorithm to take into account the presence of a C_{obst} in the workspace. This increase in complexity causes so many iterations that benefits of the bidirectional versions cannot be appreciated anymore. Those of optimized versions remain unaffected by sure. So all the implemented simulations will concern *RRT**. By using this algorithm, all the generated paths of interest are publised by the planner node in the *path.yaml* file. So by loading the map using the *map_server* node, the *visualization* node reads from this file and from *obstacles.yaml* file to visualize planning results in RViz (Fig.3.9-3.14).

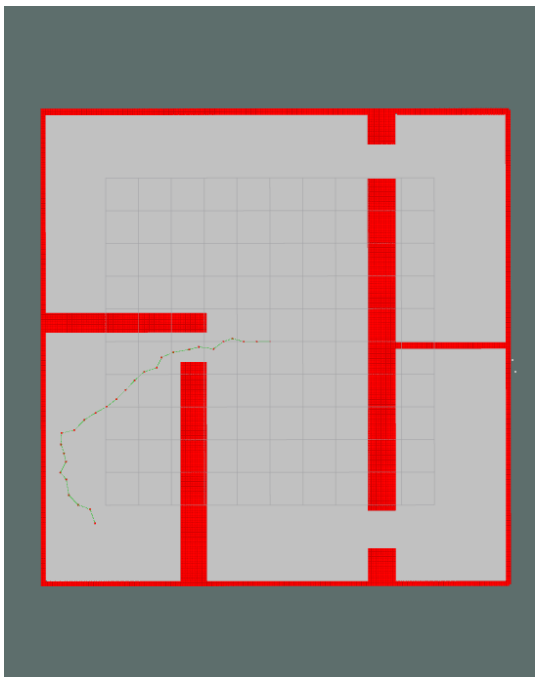

Figure 3.9: *Path 0*: resting position to warehouse

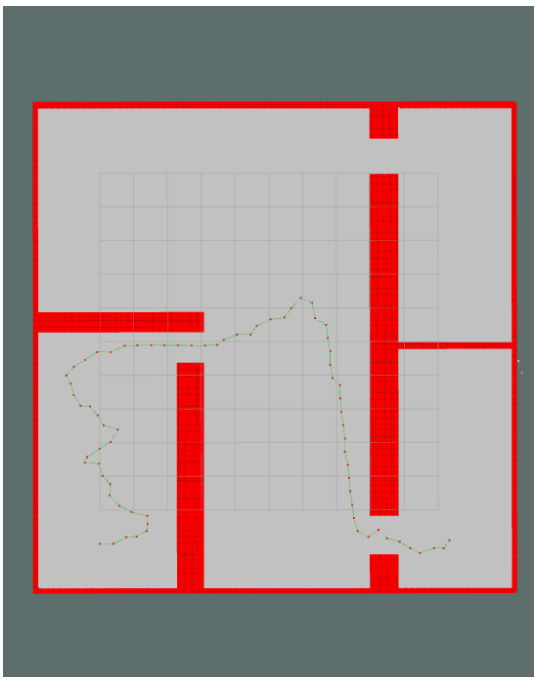

Figure 3.10: *Path 1*: warehouse to room 1

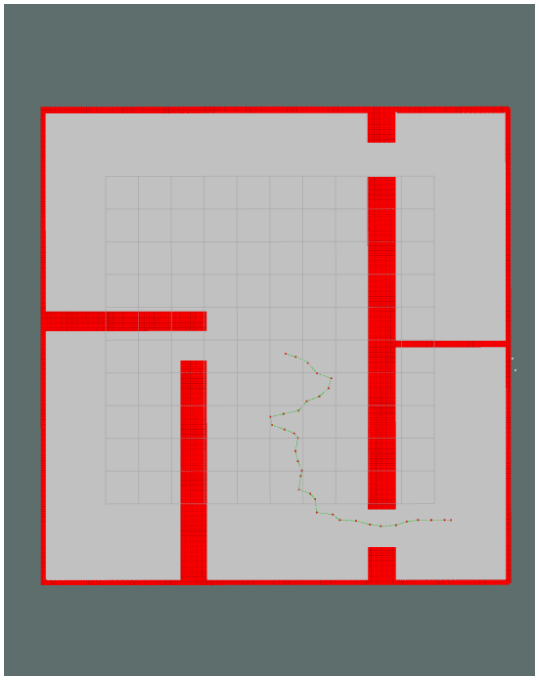

Figure 3.11: *Path 2*: room 1 to resting position

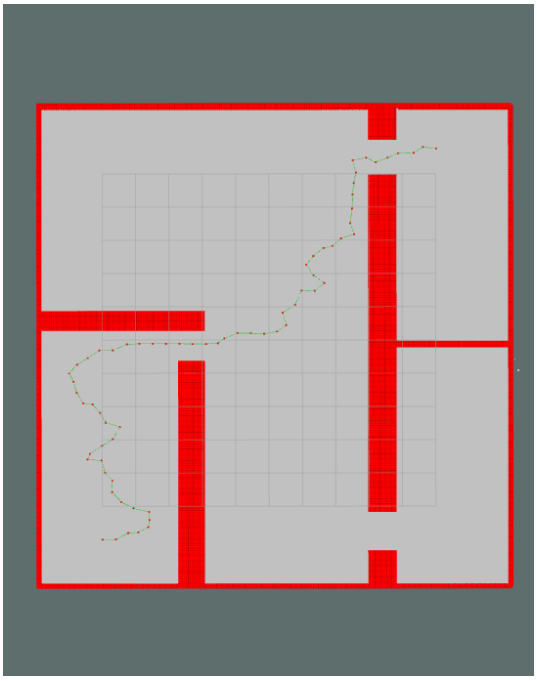

Figure 3.12: *Path 3*: warehouse to room 2

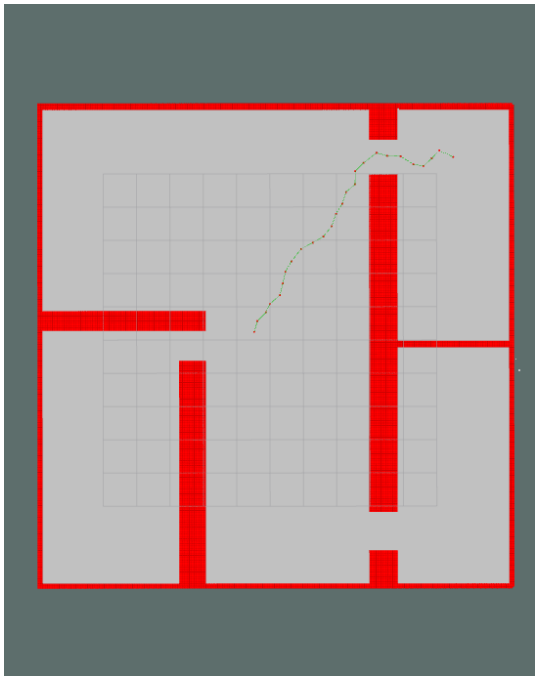

Figure 3.13: *Path 4*: room 2 to resting position

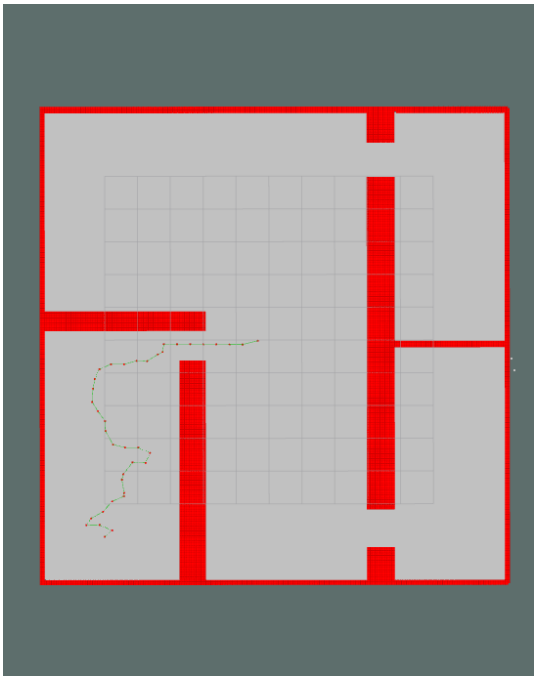

Figure 3.14: *Path 5*: warehouse to resting position

Chapter 4

Navigation

4.1 The navigator

The navigator node has been designed to choose the current path to be sent to the controller. For this reason, the navigator is made aware of the planned paths by *path.yaml* file. Then to make communication with the controller effective the current path is published on a */curr_path* topic with a *std_msgs::Float64MultiArray* message while a corrisponding identifier on */id_path* (*std_msgs::Float64*). At the same time the navigator is in charge of subscribing the number of successfully completed paths from */path_finished* topic. So navigation begins with a default initialization with a *path 0* that takes the robot to the warehouse starting from the resting position in the origin of the world frame. Once here, the robot has to detect an AR marker whose image is captured by the RGB camera sensor. The marker detection takes place using the ArUco library, related to OpenCV, which allows the navigator to access to an */aruco_marker_publisher/markers* in order to subscribe an *aruco_msgs::MarkerArray* message. Because of the presence of a single marker, it is interested only in reading the identifier of the first element of the subscribed array.

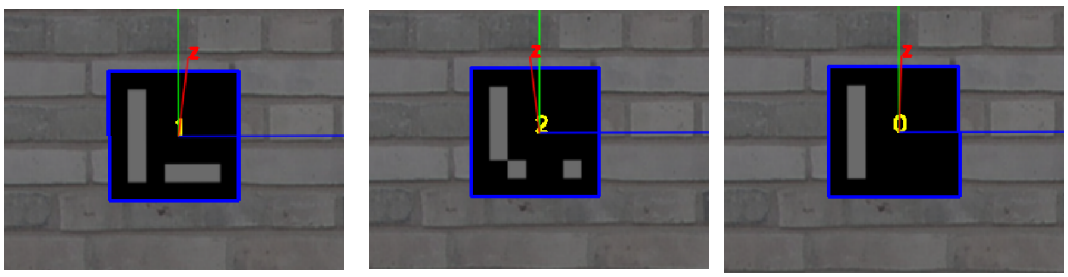

Figure 4.1: AR markers detection

As Fig.4.1 shows, visualizing *aruco_marker_publisher/result* in *rqt* framework, more scenarios have been provided:

- If the identifier is equal to *1* the robot has to reach room number *1*, and then returns back to the resting position (*Simulation 1*).
- If the identifier is equal to *2* the robot has to reach room number *2*, and then returns back to the resting position (*Simulation 2*).
- For any other reason, if *1* or *2* values detection fails, for example by replacing them with a *0* marker, the robot has to returns back to the resting position directly (*Simulation 3*).

4.2 Unexpected obstacle

Not only the RGB camera, but also the LIDAR sensor makes the navigation task online. Despite the assumption of a closed environment, the robot can catch the presence of an unexpected obstacle along one of the planned paths. To do that, the navigator node subscribes the */laser/scan* topic to gain access to the *ranges* and *min_range* fields of the sensor_msgs::LaserScan message published by the LIDAR. If an obstacle is detected closer than $0.12\,m$, a flag rises to be published on */obst* topic, so that the controller stops the motion to avoid collisions (Fig.4.2).

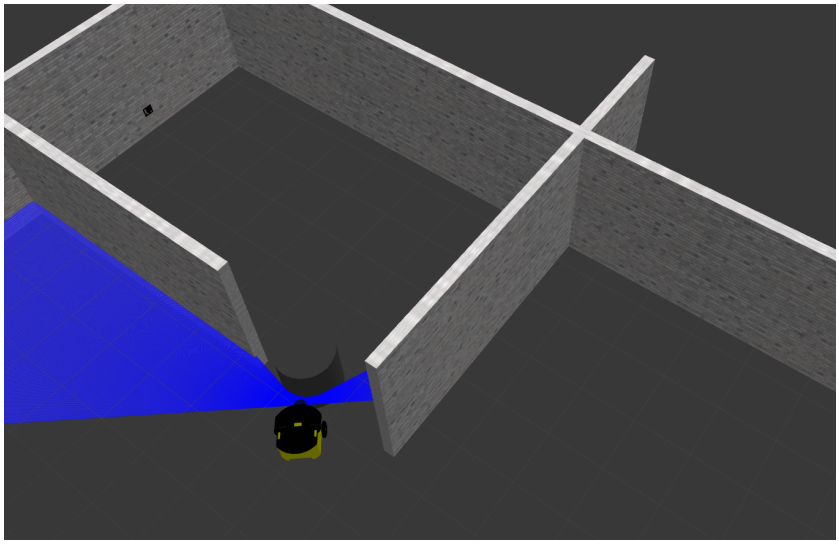

Figure 4.2: Unexpected obstacle

Chapter 5

Control

5.1 High level control

As already mentioned in sec.1.2, two main elements of the structure are the trasmissions for the joints wheels. By defining the relative hardware inteface as *hardware_interface/VelocityJointInterface* it is possible to consider a high-level velocity controller for the two motors. So in the *powerbot_controller.yaml* file they are defined as PID of *velocity_controllers/JointVelocityController* type whith a conventional gains tuning ($P : 100.0$, $I : 0.01$, $D : 10.0$). In this way, thanks to *libgazebo_ros_control.so* plugin it is possible to gain access to */left_wheel_velocity_controller/command* and */right_wheel_velocity_controller/command* topics.

5.2 Low level control

5.2.1 The controller

Publishing on the previous two command topics is the job of the controller node. It is responsible for the implementation of the solution for the low level control problem at $100\,Hz$ given rate. It requires the odometry from */odom* topic (pubished by the odom node), the current path from */curr_path*, the corrisponding id from */id_path* and a boolean flag from */obst* (all published by the navigator node).
At this point, we can separate the control problem in two steps: reaching the waypoints of each selected path and regulating the desired final orientation at each end. At this regard, by considering different proposed cases in sec.4.1, respect to the world frame, the robot has to reach a $\theta_d = \frac{\pi}{2}$ final orientation once in the warehouse (facing the AR marker); $\theta_d = \pi$ once in room 1 or room

2 (facing a potential object to grasp); and $\theta_d = 0$ once in the resting position.

5.2.2 The tracking control

Each planned path can be considered as a sequence of waypoints on the basis of which the robot has to regulate its pose. An initial observation regards the need of a control law that allows the robot to potentially reverse its motion, beacuse of the change of the selected path. Besides, thanks to the separation of the two control problems, we are not interested in the choice of a desired orientation during this tracking. These first aspects already lead to the choice of an I/O feedback linearization controller. At this regard, by considering the unicycle kinematic model in Eq.(1.3) and referring to a point B along the sagittal axis at a $b = 0.1\,m$ distance respect to the contact point with the ground, as Fig.5.1 shows, we can state that the new flat outputs of the system are:

$$\begin{aligned} y_1 &= x + b\cos(\theta) \\ y_2 &= y + b\sin(\theta) \end{aligned} \tag{5.1}$$

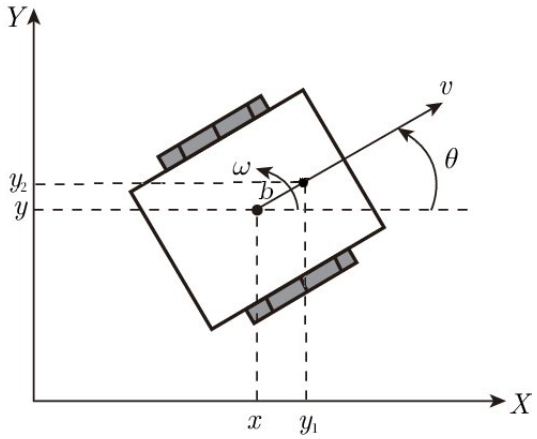

Figure 5.1: Unicycle representation

We can consider the relationship between our control inputs and the kinematic model ones:

$$\begin{bmatrix} v \\ \omega \end{bmatrix} = T^{-1}(\theta) \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} \tag{5.2}$$

,

where $T^{-1}(\theta) = \frac{1}{b}\begin{bmatrix} b\cos(\theta) & b\sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix}$. To write the final closed control loop system in the form:

$$\begin{cases} (\dot{y}_{1,d} - \dot{y}_1) + k_1(y_{1,d} - y_1) = 0 \\ (\dot{y}_{2,d} - \dot{y}_2) + k_2(y_{2,d} - y_2) = 0 \\ \dot{\theta} = \dfrac{(\dot{y}_{2,d} + k_2(y_{2,d} - y_2))\cos\theta - (\dot{y}_{1,d} + k_1(y_{1,d} - y_1))\sin\theta}{b} \end{cases} \tag{5.3}$$

that makes positional error converging to zero, we have to consider the control action as:

$$\begin{cases} u_1 = \dot{y}_{1,d} + k_1\,(y_{1,d} - y_1) \\ u_2 = \dot{y}_{2,d} + k_2\,(y_{2,d} - y_2), \end{cases} \tag{5.4}$$

with gains chosen as $k_1 = k_2 = 0.2$.
We realize that this controller allow us to take advantage of two degrees of freedom: by substituting ($y_{1,d}$ $y_{2,d}$) with each desired waypoint of the current path (by considering the traslation in Eq.(5.1)) and the relative desired velocities (without any trasformation because of the rigid body assumption). Despite this, to avoid a jerky gait for motion, that would cause impulsive wheels speed, a PD pre-filtering is introduced. So we can refer to:

$$\begin{cases} \bar{u}_1 = k_d(\dot{x}_d - \dot{x}) + k_p\,(x_d - x) \\ \bar{u}_2 = k_d(\dot{y}_d - \dot{x}) + k_p\,(y_d - y), \end{cases} \tag{5.5}$$

where $k_p = 0.2$, $k_d = 0.5$, (x_d ; y_d) is each waypoint of the selcted path and $\dot{x}_d = \dot{y}_d = 0$. By considering the PD control actions as linear accelerations, we can integrate them once to retrieve setpoints in velocity, two times to retrieve setpoints in position. Besides, for the designed algorithm, each waypoint is considered reached at $0.2\,m$ distance, as the generating node distance of the planner (sec.3.1.1), so that the next node can be assessed.

5.2.3 The orientation regulation

We can notice that because of the underactuation and nonholonomy of the system we can solve positional control problem only with the I/O feedback linearization technique. So, after reaching each path final destination, a good trick is to fix a regulation problem in a desired final orientation, by rotating the robot on the spot. To reach this goal a proportional controller is enough:

$$u = k_\theta(\theta - \theta_d), \tag{5.6}$$

with $k_\theta = 1$. Since this control law can be considered as an angular acceleration, the stearing velocity is retrived by integrating it one time. To slow down the rotation when the angular displacement is too high, a saturation at $|0.5|\, rad/s$ constant stearing velocity is introduced.

Chapter 6

Simulations

6.1 Simulation 1

From warehouse to room 1

The first proposed simulation describes the moving from the warehouse to the room number 1. Odometric error consists in the gap between the total path really tracked by the robot in the simulation environment (in red in Fig.6.1) and the one retrieved from odometric localization (in blue in Fig.6.1). The current yaw is directly provided from the IMU sensor, so that no error affects the orientation estimation.

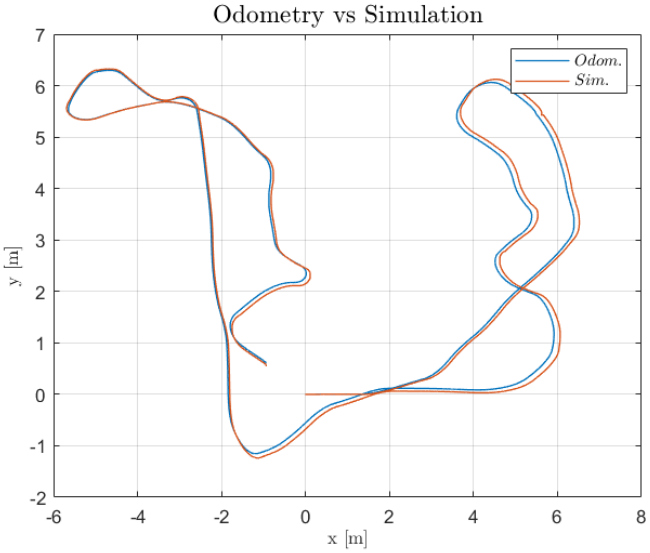

Figure 6.1: Odometry vs simulation

The evolution of the odometric error is characterized by an higher growth for the positional component along the x-axis of the world frame, with a peak higher then $0.20\,m$ when the robot is leaving the warehouse, as Fig.6.2 shows.

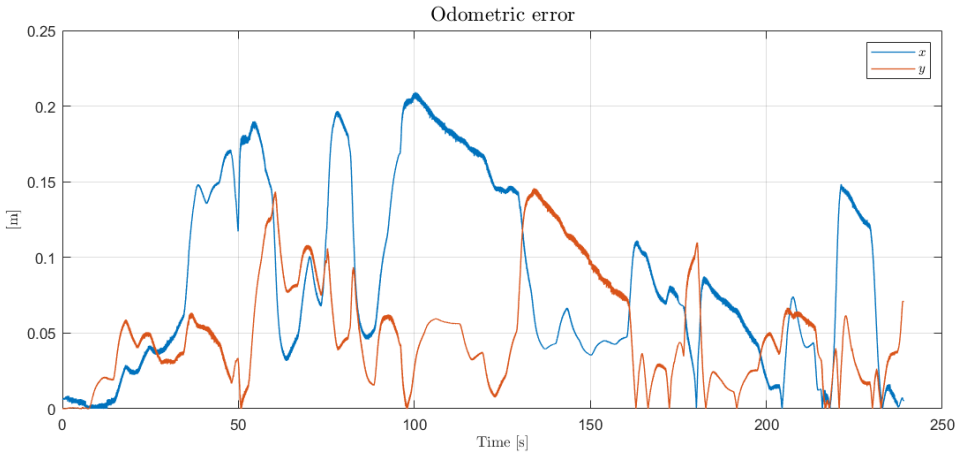

Figure 6.2: Odometric error

By considering the kinematic model inputs of the robot, the heading velocity in Fig.6.3 is presented. We can notice that motion never reverses and linear velocity becomes zero only when robot stops to rotate on the spot during orientation regulation ($47\,s$, $174\,s$ and $237\,s$). Besides, it is possible to consider a mean forward velocity of $0.22\,m/s$.

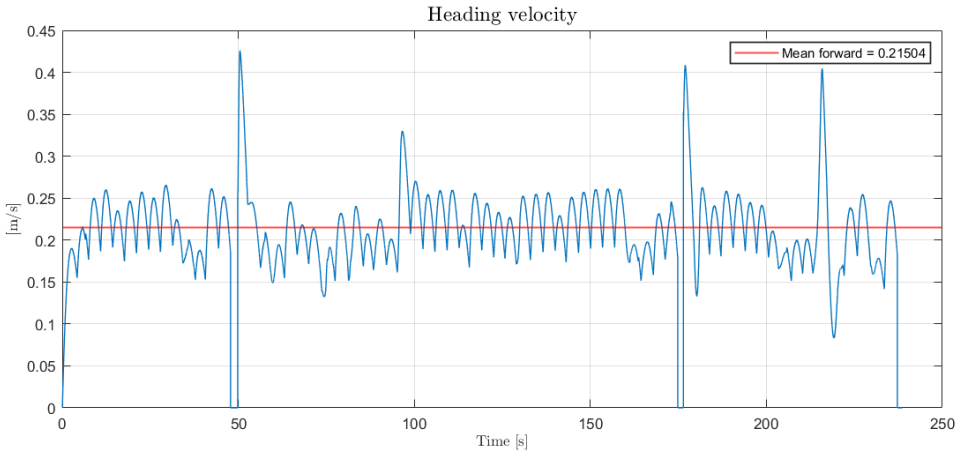

Figure 6.3: Heading velocity

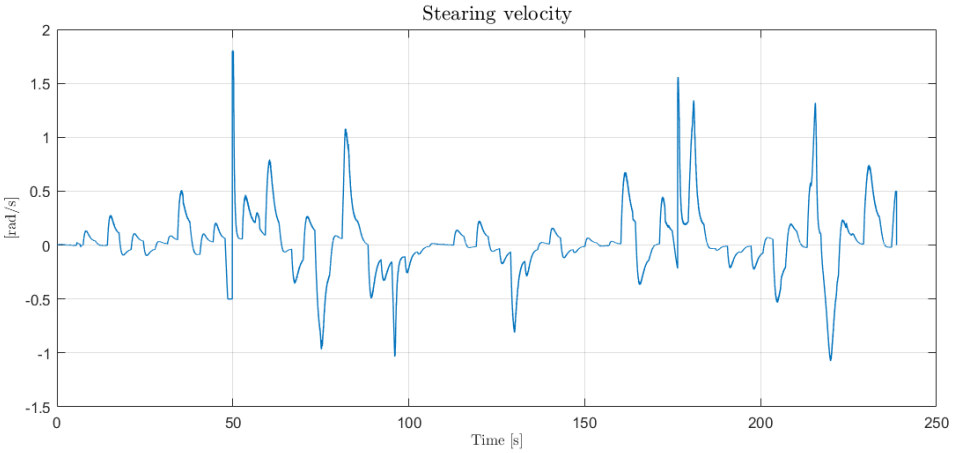

Figure 6.4: Stearing velocity

Moreover, we can notice that the first orientation regulation makes the robot rotate on the spot with a constant -0.5 rad/s stearing veocity, when the robot is targeted towards the AR marker, as Fig.6.4 shows. Right after the peak is reached and this causes an inevitable rapid increase in wheels speed too, reported in Fig.6.5. These final robot inputs are characterized by equal and opposite values when the robot rotates on the spot (47 s, 174 s and 237 s).

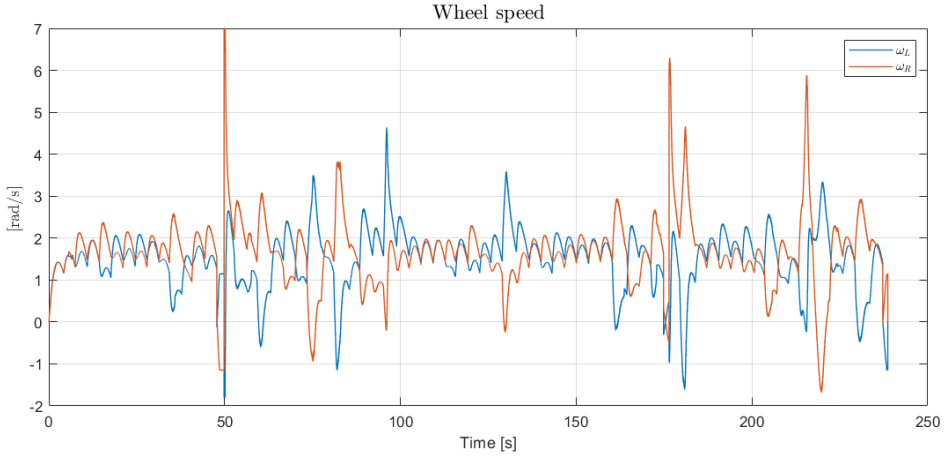

Figure 6.5: Wheels speed

Then about the solution to the tracking control problem, the plot of $\|\mathbf{p_{wp}} - \mathbf{p}\|$ and $\|\mathbf{p_{wp_i}} - \mathbf{p}\|$ are shown in Fig.6.6. The I/O feedback linearization works fine: the final goal destination of the total path (resting position in the origin of the world frame) is reached correctly at the end of the simulation. Then, the waypoints of each path are reached correctly, so local error converges to 0 too. Obiouvsly these values are relative to the offset of $0.2\,m$, according to the settings of the implemented control algorithm.

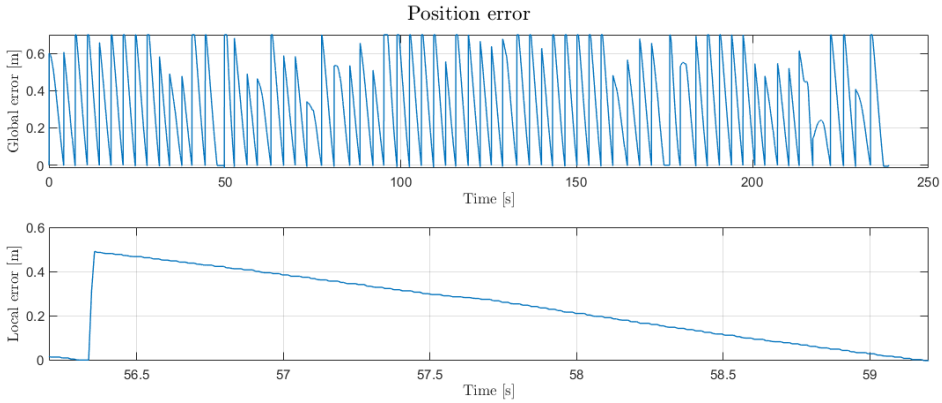

Figure 6.6: Position error

Finally, in Fig.6.7 the three rotations are highlighted, when the I/O feedback linearization controller gives way to the proportional orientation regulator.

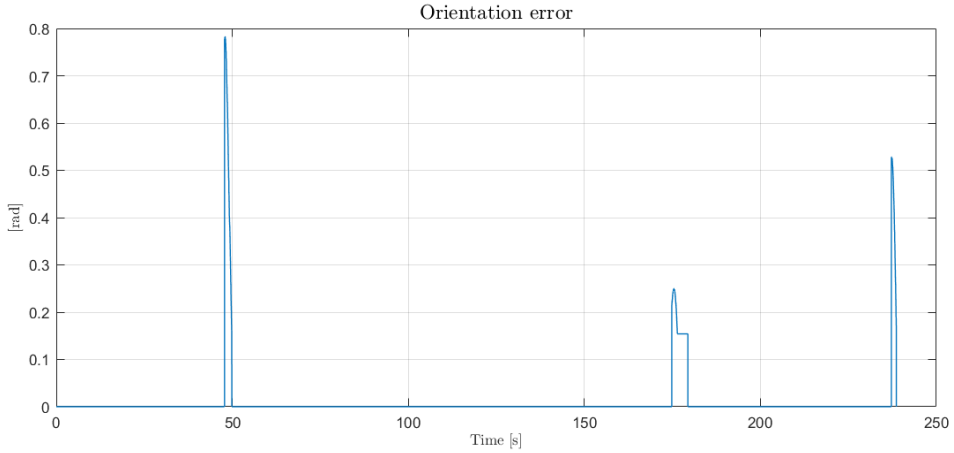

Figure 6.7: Orientation error

6.2 Simulation 2

From warehouse to room 2

In the second simulation robot moves from the warehouse to the room number 2. The gap caused by the odometric error in position is shown in Fig.6.8.

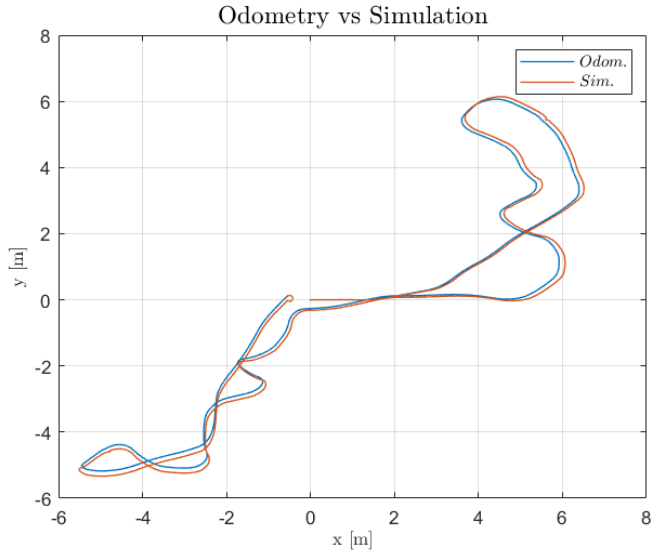

Figure 6.8: Odometry vs simulation

Now the peak of about $0.23\,m$ is reached by the error along the y-axis of the world frame, when the robot is reaching the resting position.

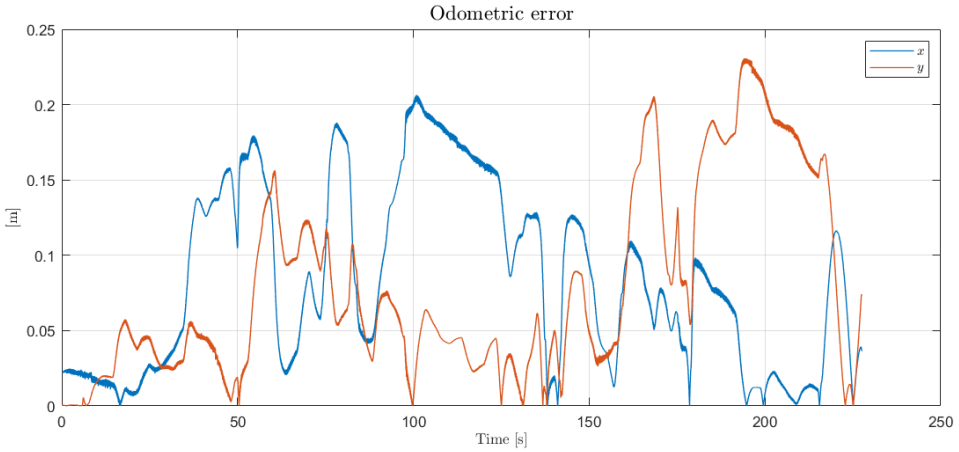

Figure 6.9: Odometric error

Then, the heading velocity shows that the robot never reverses its motion, goes to 0 when the robot rotates on the spot ($47\,s$, $173\,s$ and $214\,s$). The main forward velocity is about $0.22\,m/s$, as Fig.6.10 states.

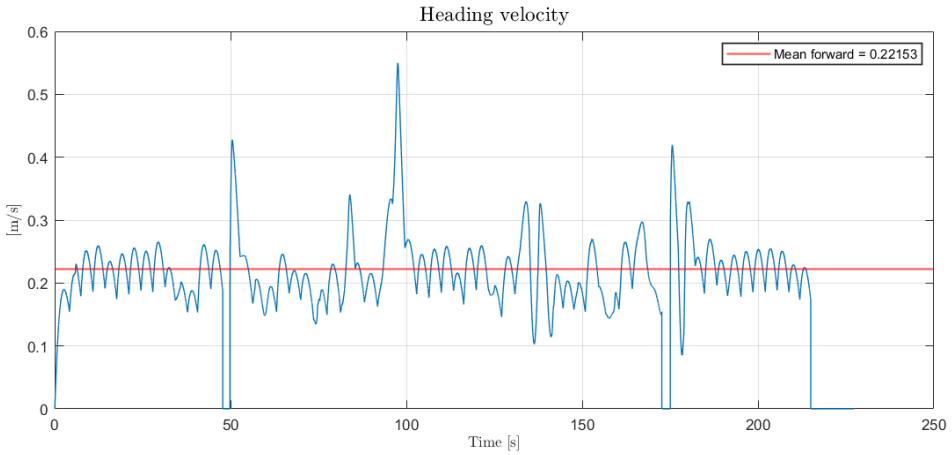

Figure 6.10: Heading velocity

During all the rotations the $|0.5|\,rad/s$ saturation takes place, especially at the end of the simulation where the rotation lasts more than $10\,s$ because of the high angular displacement.

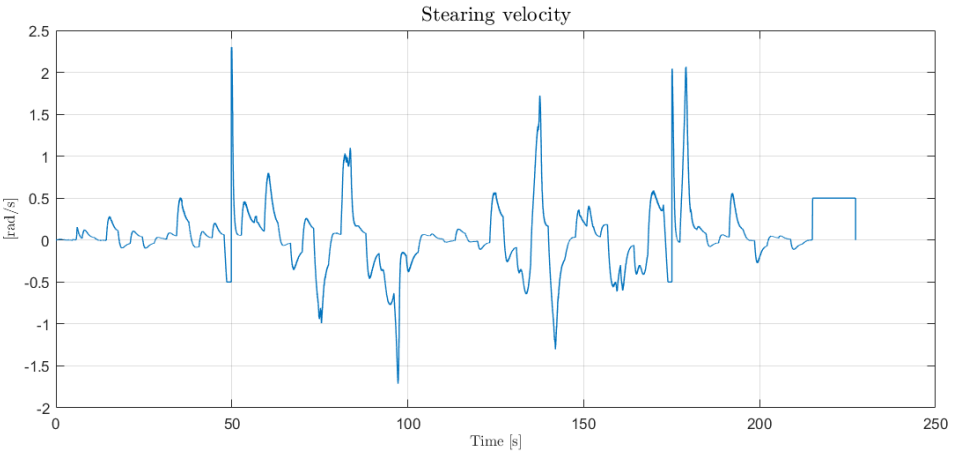

Figure 6.11: Stearing velocity

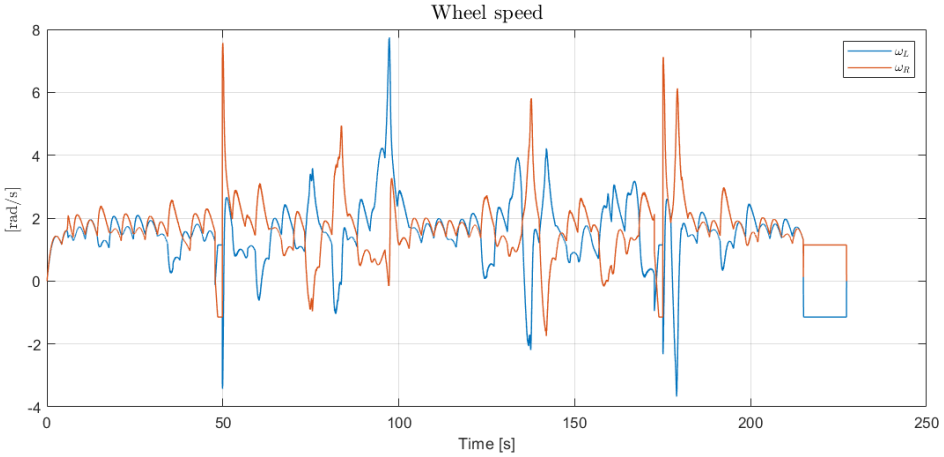

Figure 6.12: Wheels speed

The constant values assumed by the stearing velocity are reflected in the wheels speed (Fig.6.12) by considering three constant equal and opposite trends during the orientation regulations ($47\,s$, $173\,s$ and $214\,s$).
Then, about the tracking problem, the I/O feedback linearization controller works fine: positional error converges to 0 globally and locally, as Fig.6.13 shows.

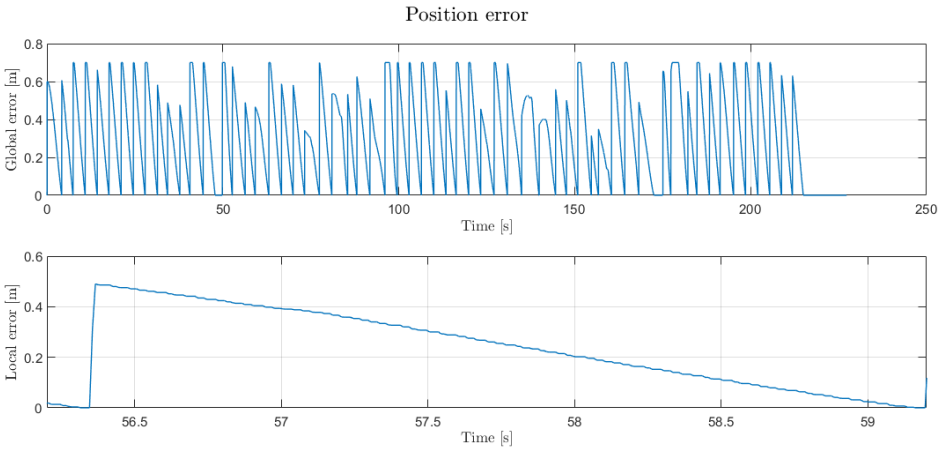

Figure 6.13: Position error

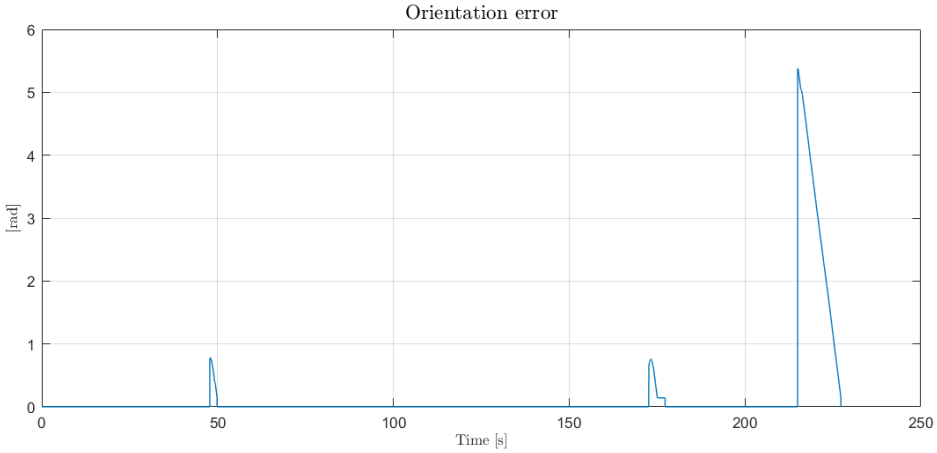

Figure 6.14: Orientation error

Finally in Fig.6.14 the three orientation regulations are shown, with a displacement larger than $5\,rad$ for the final one.

6.3 Simulation 3

Failure to detection

The last case is about the failure to AR marker detection: the identifier has been substituted with an unplanned one, for example. In these cases the robot has to come back to its resting position in the origin of the world frame.

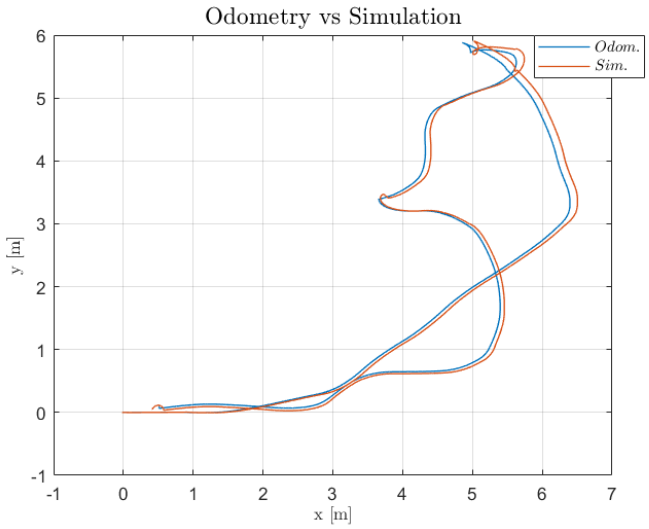

Figure 6.15: Odometry vs simulation

Fig.6.15 is about the odometric error in position while Fig.6.16 shows that the peak along the x-axis is reached immediately after AR marker detection.

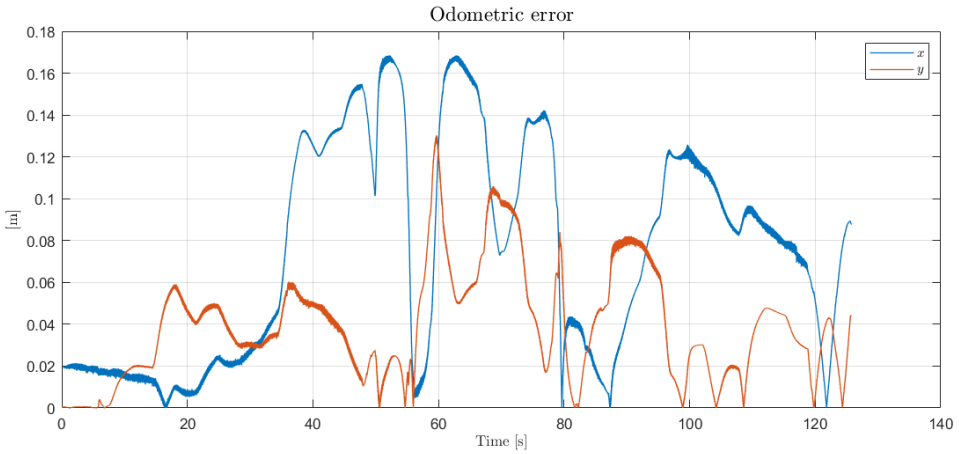

Figure 6.16: Odometric error

About kinematic model inputs, the heading velocity goes to 0 during the two orientation regulations (47 s and 119 s). Moreover, it assumes negative values too: robot reverses its motion.

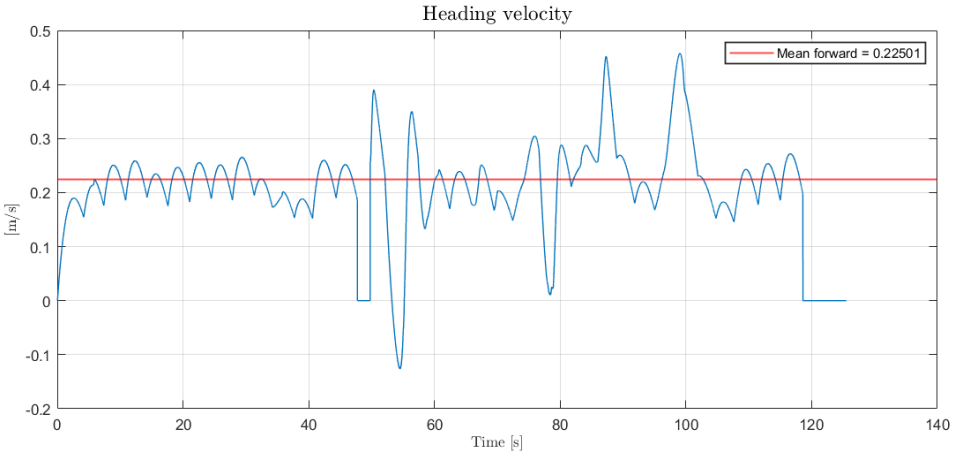

Figure 6.17: Heading velocity

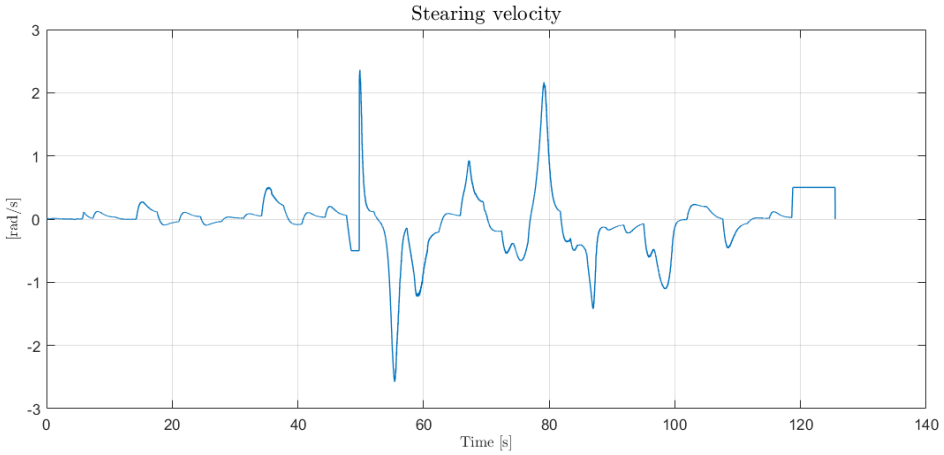

Figure 6.18: Stearing velocity

For both orientation regulations the stearing velocity assumes constant values, -0.5 rad/s and 0.5 rad/s respectively, as shown in Fig.6.18. This is reflected in wheels speed: two equal and opposite constant trends are shown at 47 s and 119 s.

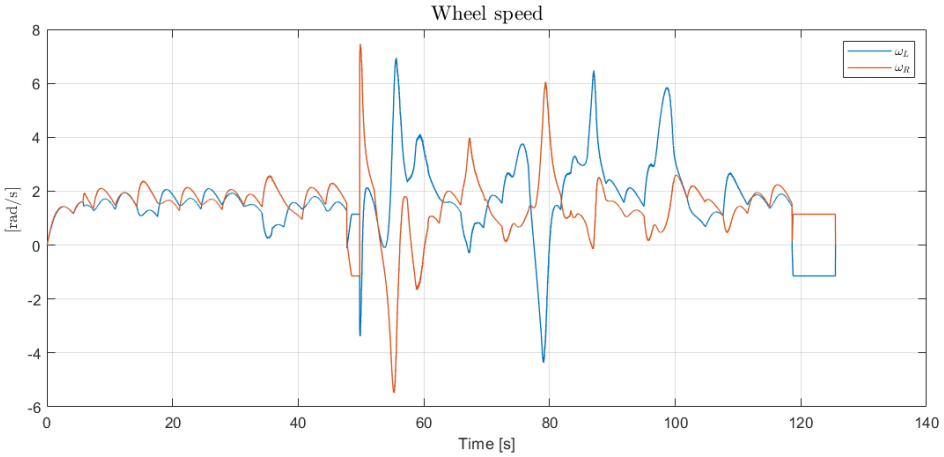

Figure 6.19: Wheels speed

About tracking control problem, the position error is reported in Fig.6.20: both globally and locally converges to 0.

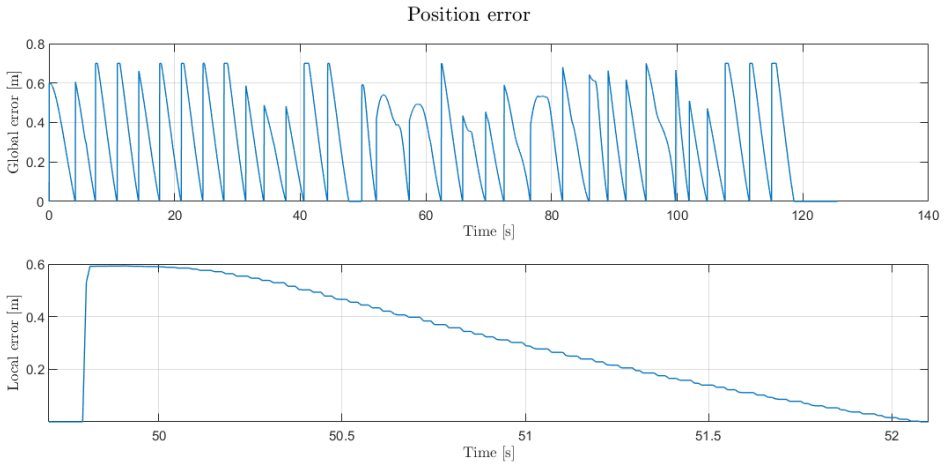

Figure 6.20: Position error

Finally, in Fig.6.16 orientation error schedules the round-trip: the robot reaches the goal destination from behind at the end of the motion, a final regulation of about $3\,rad$ is enough to terminate simulation.

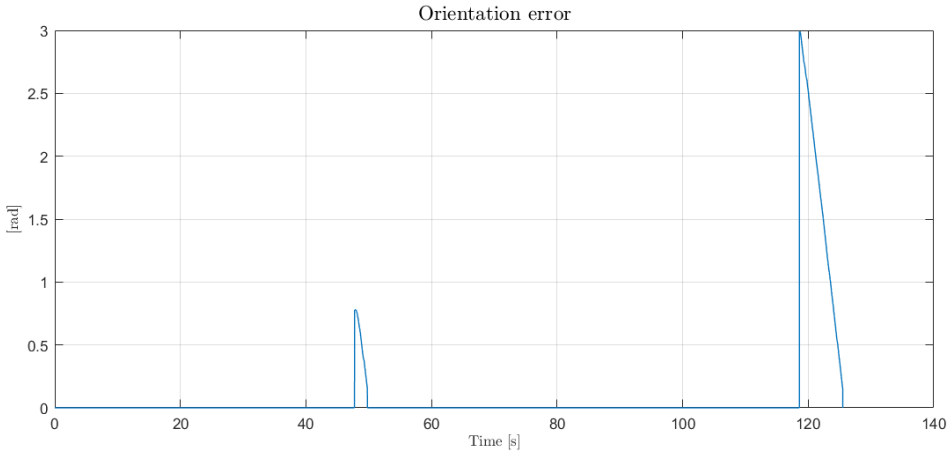

Figure 6.21: Orientation error

Bibliography and sitography

[1] B. Siciliano, L. Sciavicco, L. Villani, and G. Oriolo "*Robotics: Modelling, Planning and Control*". 1*st* edition. Springer, 2009.

[2] https://robots.ros.org/powerbot/.

[3] https://github.com/Sicelukwanda/champ_sim_pkgs/tree/master/champ_description/urdf/powerbot.

[4] http://wiki.ros.org.