DIPARTIMENTO DI INGEGNERIA ELETTRICA E TECNOLOGIE DELL'INFORMAZIONE

SCUOLA POLITECNICA E DELLE SCIENZE DI BASE

Field and Service Robotics/Robotics Lab
Technical Report

PLANNING (APF), CONTROL, ODOMETRY AND COMPUTER VISION-BASED PROJECT FOR A DIFFERENTIAL-DRIVE ROBOT

Profs. F.Ruggiero J.Cacace

Paolino De Risi
P38/062

Academic Year 2020/2021

Table of Contents

Abstract

The following technical report will present the various implementation choices to resolve the given task.
This academic essay will be displayed singularly: it will show the implemented project and argue every employed technical solution.
The instructions and limitations provided by the rules and the project description are considered mandatory. So, at the beginning of each chapter, the most relevant sentences from the assignment will be quoted to demonstrate how that specific point has been achieved regarding the given directions.

Each chapter will deal with a theoretical and an implementation section.

Note: where not specified, the length measures are expressed in meters.

Foreword

The following is a study about computer vision, kinematics, planning algorithms, control and odometry for a differential drive robot given an autonomous navigation capability in a Gazebo-ROS environment.

The first chapter will deal with the implemented environment, presenting all the implementation choices on it to get the best out of the whole navigation.
In the second chapter there will be a treatment about the differential wheeled robot's kinematic model and the implemented robot model will be presented.
The third chapter is dealing with a first chart solution to the whole project. There will be also presented the most important nodes to the project in the ROS network.
In the fourth chapter, the implemented odometry will be presented, and there will be, also, a discussion about its systematic and non-systematic errors according to [Reference 2].
The fifth chapter will deal with the solution to the AR markers recognition problem.
The sixth chapter will present the implemented planner and the controller. There will be a theoretical presentation of the artificial potential approach to plan trajectories, then a mathematical formulation of the attractive and repulsive potential and there will be treated the implemented solution to explore the whole room and to the local minima traps. For the controller, there will be presented the high-level controller and the low-level one, with particular attention to motivate why an I/O linearization controller has been employed.
The seventh chapter will present the solution to the mapping problem.
Finally, the eighth chapter will present the results of a simulation, with different plots retrieved by the *Matlab* suite.

1 Scenario

"The environment is composed of 4 main locations: a starting location and three other locations to be explored [...] The coordinates of the location are known"

The scenario has been defined very strictly. The resulting *world* file [Fig.1] has been implemented to stress the whole navigation algorithm, to better appreciate its capabilities.

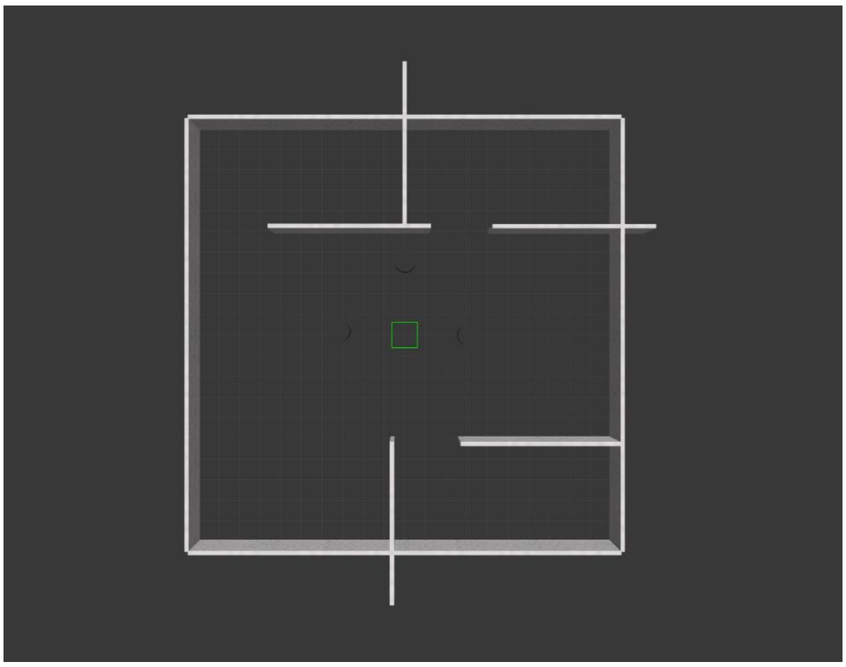


Figure 1: Plan view of the implemented environment

At first, ample space has been designed to strain the planning algorithm and the odometry: the robot's area is a 20X20 square.

The positions of the doors are not trivial: the door is not at the centre of the room, but it is at the tip. In this way, since an online version of artificial potential performs the planning, the probability of getting in a local minimum trap is increased.

The adoption of the wall model to build the *world* file has been very conscientious. The selected wall texture is uneven with grey-scale tones.

As the robot searches for AR-markers in the rooms, the purpose was also to stress the exploration task. Indeed, this kind of texture offers a low contrast to make the marker less recognizable [Fig.2], so the robot has to get very close to it to scan it.

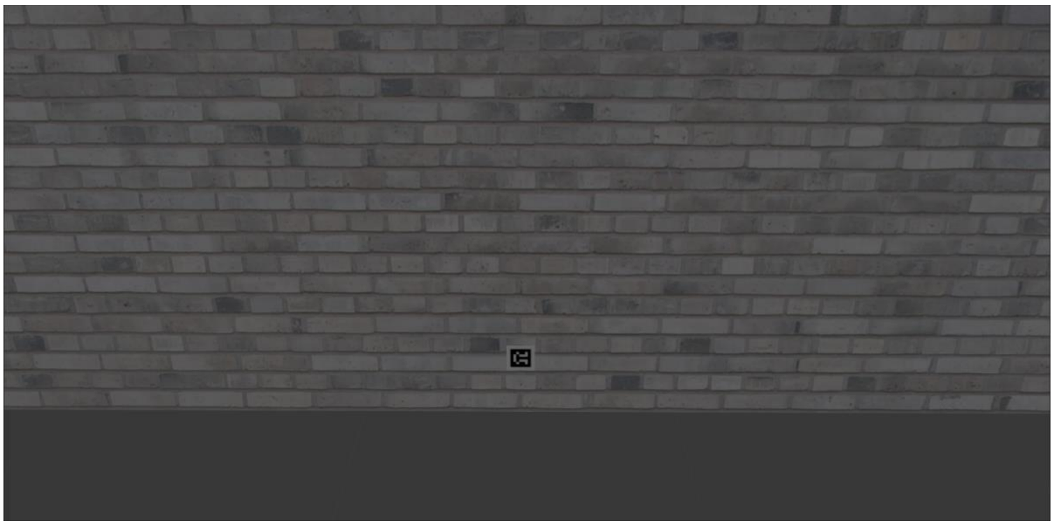


Figure 2 The wall texture and an AR-marker applied to it

The markers' positions are not mean; they have been placed on the opposite wall to the room entry, moreover they have a small dimension (0.09X0.09 meters), to make the robot explore the whole chamber.
Finally, three cylindric obstacles occur in a region close to the base, interfering with the navigation.

The entire environment has been designed by importing the wall model available in the Gazebo model library and replicating it, varying its pose. Then the Ar markers models were imported and placed in the desired positions. Finally, the cylindric obstacles have been set in the *world*.

The *world* file has been created through the *save world* Gazebo command and saved in the world folder in the *examprj* directory.

2 Robot model description

"Perform a state-of-the-art research, or commercial research, to find out a differential drive robot from whose datasheet it is possible to extract the necessary kinematic and dynamic parameters. It is possible to use already existing imported models."

A differential drive robot comprises two actuated fixed wheels and passive caster wheels to allow statical balance.
In order to introduce the robot's kinematic model, as is well known, a differential drive robot (two control inputs ω_R and ω_L) is kinematically equivalent to a unicycle (two control inputs v and ω) thanks to the mapping:

$$v = \frac{\rho_w(\omega_R + \omega_L)}{2} \tag{1}$$

$$\omega = \frac{\rho_w(\omega_R - \omega_L)}{d_w}, \tag{2}$$

where ρ_w is the wheels' radius and d_w the wheelbase.

Therefore, it is possible to employ the equivalent unicycle kinematic model for the differential robot.

The unicycle configuration space is described by:

$$\boldsymbol{q} = [x \quad y \quad \vartheta]^T, \tag{3}$$

where x and y are the cartesian coordinates of the wheel's contact point C with the ground and ϑ is the vehicle's yaw, dealing with the angle between the sagittal axis and the x-axis.

The pure rolling constraint is represented in the Plaffain form as:

$$\dot{x}\sin(\vartheta) - \dot{y}\cos(\vartheta) = 0, \tag{3}$$

representing that the velocity in the contact point with the ground has no component along the normal direction to the sagittal axis.
By this non-holonomic constraint, it is built the robot's kinematic model:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\vartheta} \end{bmatrix} = \begin{bmatrix} \cos(\vartheta) \\ \sin(\vartheta) \\ 0 \end{bmatrix} v + \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \omega, \tag{4}$$

where v is the heading velocity, and ω is the steering one.

The unicycle is a differentially flat system with x and y flat outputs. So does the differential drive robot; this means that the state and input vectors are functions of the so-called *flat outputs* and their derivatives. So, given a cartesian path through the flat outputs $(x(s), y(s))$, the whole $\boldsymbol{q}(s)$ and the $\tilde{v}(s)$ and $\widetilde{\omega}(s)$ are completely defined:

$$\boldsymbol{q}(s) = \begin{bmatrix} x(s) \\ y(s) \\ \vartheta(s) \end{bmatrix} = \begin{bmatrix} x(s) \\ y(s) \\ Atan2\big(y'(s), x'(s)\big) + k\pi \end{bmatrix}, \tag{5}$$

with $k = 0$ for a forward-moving or $k = 1$ for a backward one,

$$\tilde{v}(s) = \pm\sqrt{\big(x'(s)\big)^2 + \big(y'(s)\big)^2} \tag{6}$$

$$\widetilde{\omega}(s) = \frac{y''(s)x'(s) - x''(s)y'(s)}{\big(x'(s)\big)^2 + \big(y'(s)\big)^2}. \tag{7}$$

Thanks to the robot's differential flatness, the planning algorithm has to provide the two desired cartesian coordinates, systematically abiding by the non-holonomic pure rolling constraint.

Coming to the implementation part, one of the most employed high dimensioned differential drive robots for exploration tasks is the *Pioneer 2dx/3dx*. Therefore, this robot model has been the most important reference to the user-defined *xacro* model [Fig.3]. Indeed, the differential drive robot *xacro* model has been coded from scratch.

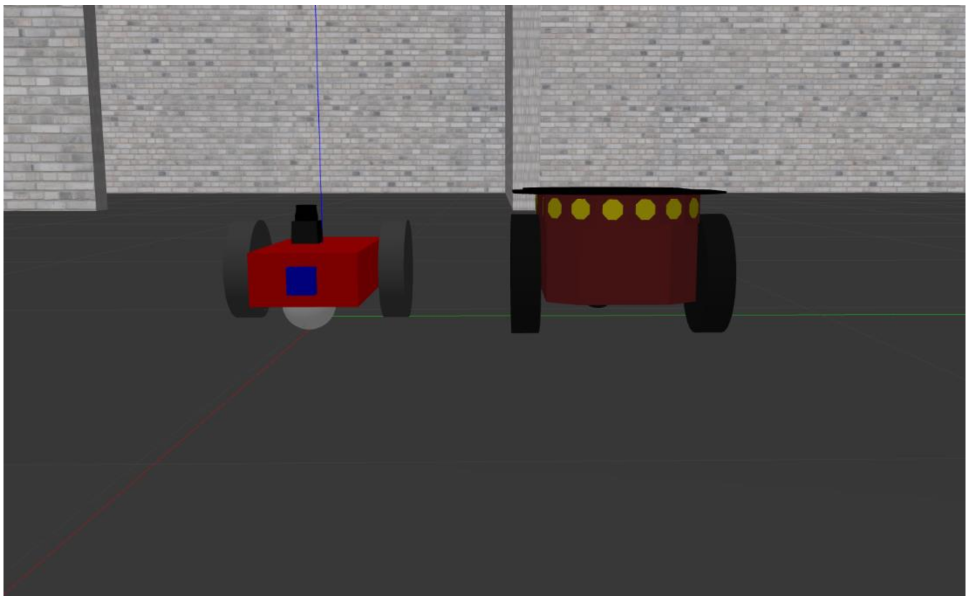


Figure 3 The implemented robot, compared to a Pioneer 2dx

The choice of a large robot has been adopted due to the environmental dimensions and simplifying the localization problem. A high dimensioned wheeled robot, indeed, allows being closer to a pure rolling motion hypothesis, avoiding slippage.
At the same time, the coding approach from scratch is due to two reasons. The former cause is that the (few) *Pioneer 2dx xacro* models available on the web are full of high-level built-in navigation libraries. Therefore, much effort would have been necessary to eliminate them than configuring a whole robot with similar kinematic and dynamic parameters on self-own, using the *Pioneer* model as a reference. The latter cause is that it could be seen as an excellent exercise to improve XML-coding skills. The references to the Pioneer model are highlighted in the following table.

Relevant parameters	**Pioneer 2dx**	**The built robot**
Wheel diameter (mm)	195	200
Mass value (kg)	9	9
Height (mm)	237	200
Wheel separation (mm)	381	400

Clearly, except for the lidar sensor, the employed geometries are elementary:

- the robot's chassis is a 0.4X0.2X0.1 box;
- the wheels are two cylinders with a 0.1 radius and a 0.05 height;
- back and front casters have been configured as two hemispheres of 0.05 radius to allow static balance.

Through the *urdf_to_graphiz* tool, it is possible to view the connection graph of the implemented robot [Fig.4].

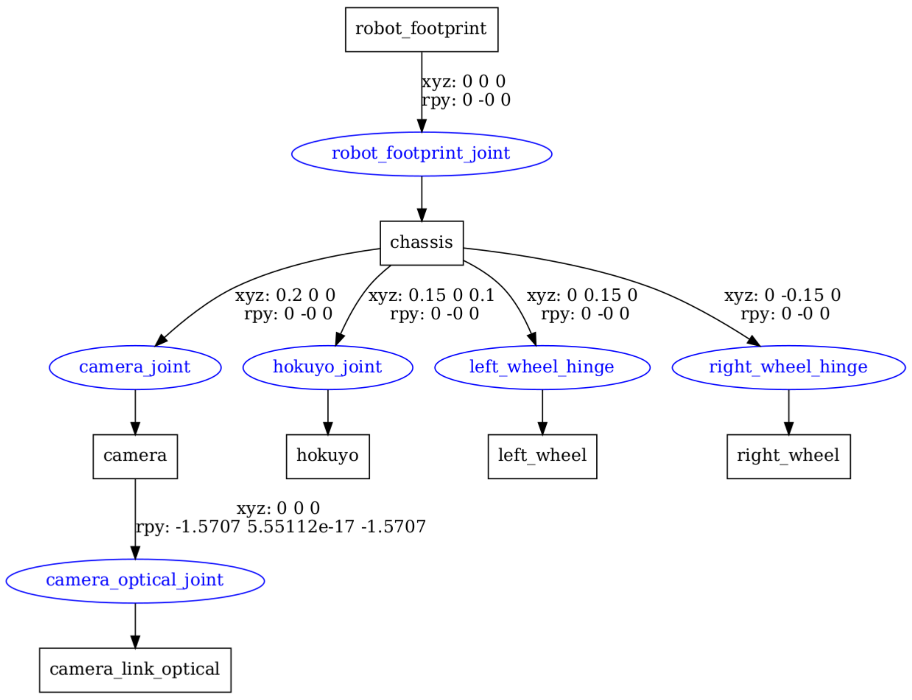


Figure 4: Connection graph of the implemented robot

Five Gazebo plugins have been configured in the *.macro (.gazebo)* file: RGBD camera, Hokuyo lidar sensor, IMU sensor, "wheels' encoders," and Gazebo ROS controller.
These plugins were essential to robot navigation, and they will be explained further in the following chapters.

The coding style was not the classical used for *xacro* files. In this case, the modelling was similar to a *urdf* coding style: in the *.macro* file there is no link or joint replicable in the *.xacro* one, but it was used to define plugins and elements' colours.

3 Conceptual solution

The sketch solution has been implemented considering the following conceptual scheme [Fig.5]: an odometry block, a computer vision block, a planning block and a control one.

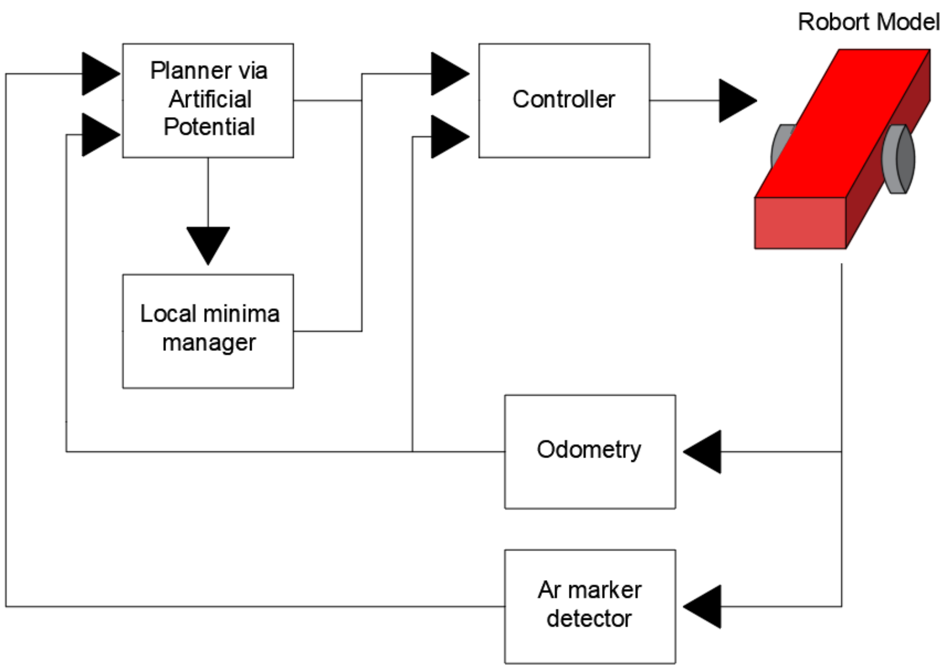
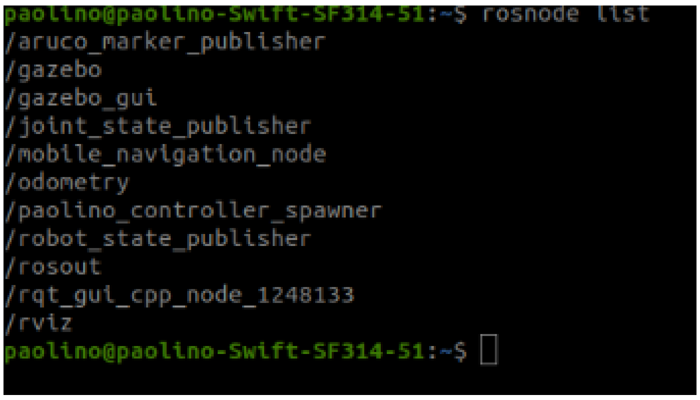


Figure 5: Conceptual block scheme

The conceptual sketch of the solution became the essential structure of the Ros network.
Each block has become a *rosnode*, with the planning and control ones joining in a single node named *navigation*.

The ROS middleware has been used to create communication among these nodes, through the *Publish/Subscribe* communication protocol. Referring to [Fig.5], it is understandable why this is the only communication method employed: since the *Publish/Subscribe* is a one-direction communication protocol, it perfectly suits the conceptual solution. It also offers a security guarantee, making the programmer constantly aware of which node is modifying the others' data.

In [Fig.6], it is possible to recognize all the ROS network nodes once the executables are run.



```
paolino@paolino-Swift-SF314-51:~$ rosnode list
/aruco_marker_publisher
/gazebo
/gazebo_gui
/joint_state_publisher
/mobile_navigation_node
/odometry
/paolino_controller_spawner
/robot_state_publisher
/rosout
/rqt_gui_cpp_node_1248133
/rviz
paolino@paolino-Swift-SF314-51:~$ 
```

Figure 6: List of the rosnodes in the network

4 Odometry

"During the whole task, the robot must be able to localize itself into the map without using precise information provided by the simulator (i.e., odometry and additional sensors must be used to solve localization and/or simultaneous localization and mapping problem)"

In a control loop for a mobile robot, odometry represents the control's and the planner's feedback.
Estimating current pose value, odometry allows the planner (the controller) to compute the position error based on the global (local) goal.

As proven, differently from an industrial robot, it is much more complex to obtain a precise pose estimation for a mobile wheeled robot. The set of joints encoders does not provide the pose of the whole robot but it could be used to evaluate the robot's movements. Moreover, the direct kinematic problem is blind to unmodelled real-world effects, such as sliding motion. As said before, a large wheeled robot allows mitigation of the sliding effects.

The *odom.cpp* file is the odometry computing executable. It is defined as a periodic task with a rate of 200Hz, and this rate is precisely double the controller's one.

An odometry system has to receive information about the wheels' velocities during the sample time. There are two possible approaches to retrieve this data.
The former solution is to retrieve the wheels' velocities by the controller's outputs. Even if it is a mild solution, there is no guarantee that the commanded speed is the same as the robot's actual one, considering that each actuation system is not ideal, and pose estimation errors could occur.
The latter solution relies on wheels' encoders. Through these proprioceptive sensors, it is possible to retrieve the effective average wheels' angular velocities.

However, there is no real encoder in the proposed solution: a real encoder, placed on the motor's camshaft, pulls the angular displacement between the current sample and the previous one, retrieving the angular velocity using the sample time. The solution has been simplified directly

taking the wheels' speeds through a self-implemented Gazebo plugin (*wheel_vel_plugin.cpp*).
This plugin publishes on the *"/diff_wheels/vel"* topic subscribed by the odometry.
Odometry takes the two wheels' velocities, mapping them in the unicycle's heading and steering velocities through (1) and (2).

Then, by a second-order Runge-Kutta approximation (more accurate than the standard Euler integration technique, since it considers an average yaw value during the sample time interval), it is possible to retrieve the linear x and y coordinates and the yaw one:

$$x_{k+1} = x_k + v_k T_s \cos\left(\vartheta_k + \frac{\omega_k T_s}{2}\right) \tag{8}$$

$$y_{k+1} = y_k + v_k T_s \sin\left(\vartheta_k + \frac{\omega_k T_s}{2}\right) \tag{9}$$

$$\vartheta_{k+1} = \vartheta_k + \omega_k T_s, \tag{10}$$

where T_s, the sample time is the inverse of the odometry task rate, 5ms.
Giving as initial conditions the starting position values,

$$x_0 = 0 \tag{11}$$
$$y_0 = 0 \tag{12}$$
$$\vartheta_0 = 0, \tag{13}$$

the odometry can correctly estimate the robot's pose during the full navigation. It works, even if a cumulative drifting error, originated by the integration, could become relevant since the navigation takes much time.

However, that is not the only error affecting odometry. As in *IEE Transactions on Robotics and Automation, Vol 12, No 5, October 1996*, odometry is also affected by systematic and non-systematic errors.
Using the above solution, indeed, several significant pose estimation errors are encountered.
These are commonly due to many non modelled effects: *unequal wheel diameters, misalignment of wheels, uncertainty about the effective wheelbase, wheel-slippage due to slippery floors, over-acceleration, fast turning, external forces (interaction with external bodies), internal forces (castor wheels).*

In specific, the central problem is represented by the orientation errors because once incurred, they degenerate in no-bounded linear position errors, as intuitable from (8)-(9)-(10).

In the case at hand, non-systematic odometry errors are not at all questioned. They derive from irregular ground morphologies (not modelled in *Gazebo*) or external forces (as guessable if the robot gets in contact with a fixed obstacle, its wheel will keep on turning, causing a slip, but the implemented planner is employed to avoid obstacles).

The project odometry is, actually, mainly hampered by systematic odometry errors. These are broadly due to unequal wheel diameters and uncertainty about the effective wheelbase.
While unequal wheel diameters could solely occur in real robots due to the rubber tires compressing differently under asymmetric load distribution and merely affect straight-line motion, uncertainty about the effective wheelbase also affects simulated models.
The wheelbase is the distance between the right and the left contact points of the two drive wheels of a differential robot.
This dimension is affected by uncertainty because the contact between wheels and ground during the robot's navigation is not punctual (as said before, wheels have been modelled as cylinders).
This kind of error concerns only turning, and therefore it could be very dangerous to the odometry.

A confirmation on the occurring of the wheelbase uncertainty error has been provided through a *uni-directional square path benchmark* [Fig.7].

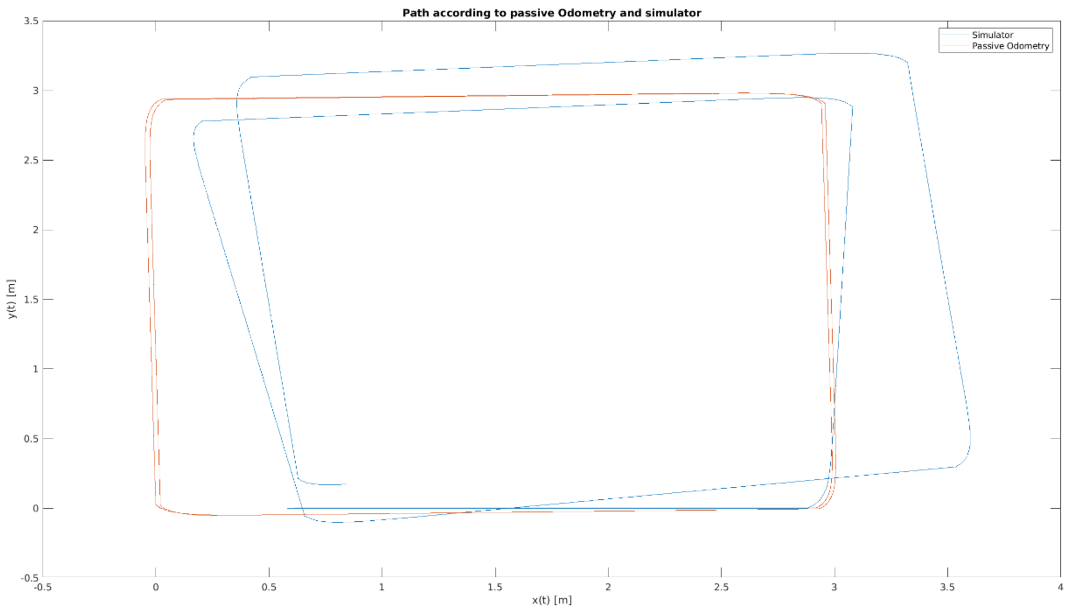


Figure 7: Uni- directional square path benchmark. The real path of the robot is an inclined square. This proves the occurency of uncertainty about wheelbase error.

Because of this occurring error, the robot's navigation could not go on correctly.

Even if the systematic errors could be compensated, a different solution has been implemented.
Since the wheelbase uncertainty error affects only steering motion, an IMU sensor, implemented through the *imu_plugin* from Gazebo, has been employed to obtain the yaw value.
The IMU sensor publishes the computed data to the */imu* topic with an average 155Hz rate, even if the update of the wheels velocity happens through an event-based call, using a *sensor_msgs/Imu* and the */imu* topic is subscribed by the odometry through the *imu_sub_callback*.
This callback receives the orientation quaternion and converts it into an RPY rotation matrix; as pitch and roll angles are not considered, the only variable of interest is the yaw angle, which is much more accurate with this implemented code design.

Once fixed the yaw error, it is possible to use (8) and (9) to retrieve the linear positional coordinates. Via this solution, odometry error has decreased remarkably [Fig.8].

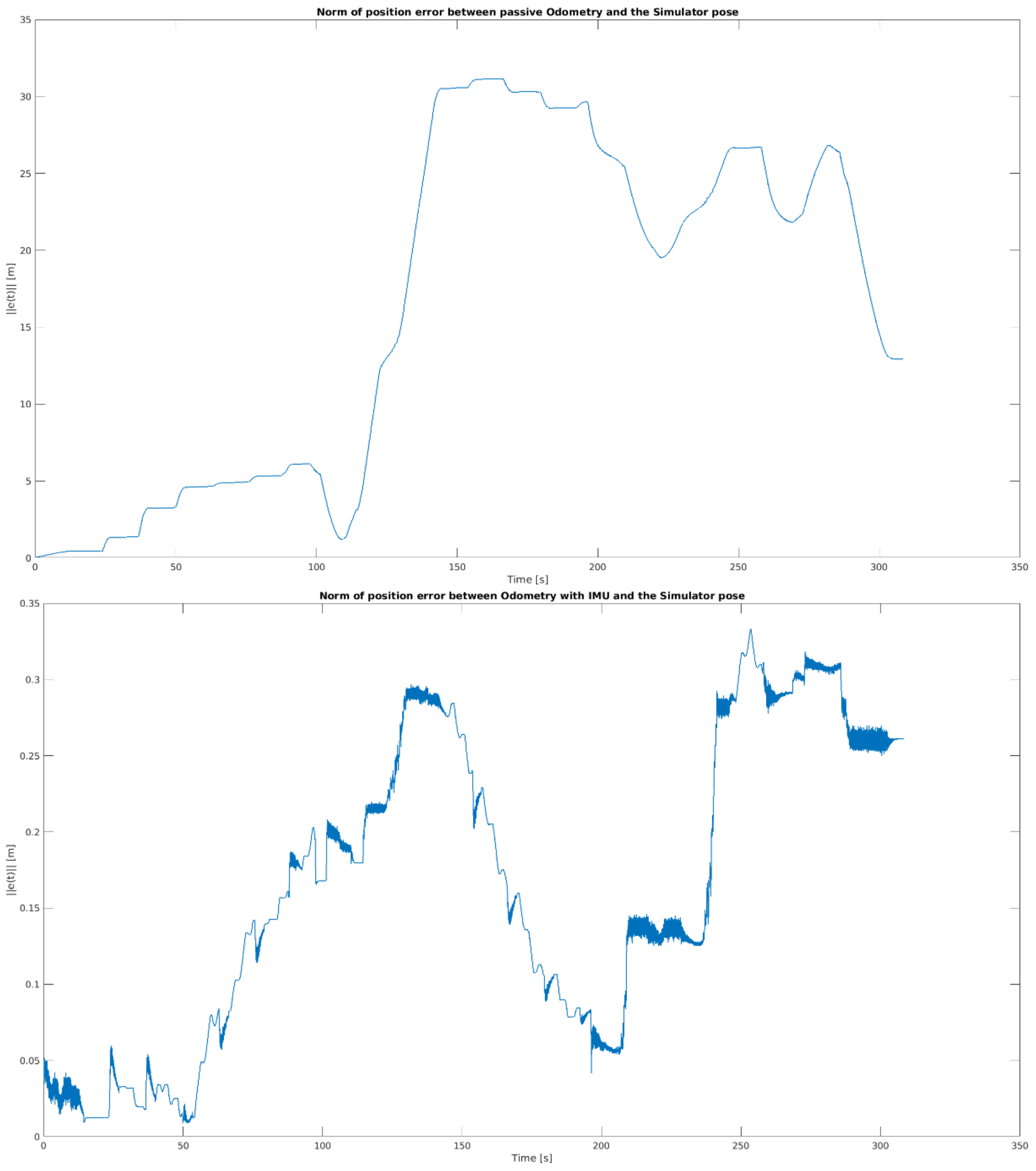


Figure 8: Norm of the position errors. The odometry taken by Gazebo is assumed as a reference. It is clear how much the IMU sensor benefits odometry.

The odometry also computes the robots velocities as:

$$\dot{x}_{k+1} = v_k \cos(\vartheta_k) \tag{14}$$
$$\dot{y}_{k+1} = v_k \sin(\vartheta_k) \tag{15}$$
$$\dot{\theta}_{k+1} = \omega_k. \tag{16}$$

The executable does not just publish the computed odometry to the */odom* topic but also models the odometry transform frame available in Rviz [Fig.9].

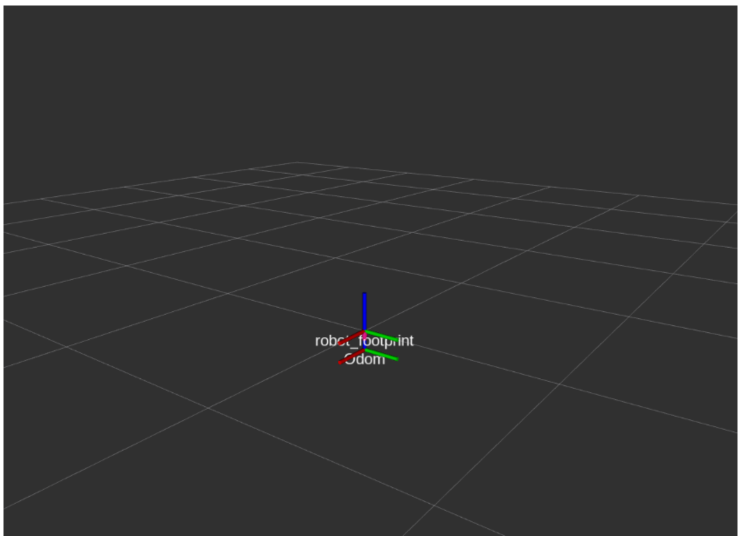


Figure 9: Robot's frame and odometry frame

5 Ar-markers detecting

"Within each location, a QR-code/AR-marker is placed in an unknown position. At the beginning, the user inserts an id among the ones present in the QR-codes within the exploring locations. The robot must navigate and explore all the locations until the id requested by the user has been found or all the locations have been explored."

An Ar-marker is a fiducial symbol, usually introduced to simplify object pose estimation by a camera.
It is represented as a square image with a black border, and an inner region encodes a binary pattern, determining it as a unique identifier.
Even if the ar-markers primary purposes are in the augmented reality and camera pose estimation fields, in this case, the only relevant aspect to the implemented work is to detect a *marker_id*.

In order to scan a marker, a library to detect squared fiducial markers in images and a marker dictionary are necessary.
ArUco has represented the former. It is a free source c++ library relying on OpenCV.

There are several types of markers, each of them belonging to a dictionary. Each library has proposed its own markers, like ArToolKit+, Chilitags, AprilTags, and ArUco dictionary. Even if ArUco could detect markers belonging to different libraries, it has been preferred the proprietary dictionary in order to move in a safer environment.

Once cloned the ArUco library, three markers' textures (ids 8, 26 and 582) are already provided.
Therefore, these markers have been added to the environment in the following positions[Fig.10]:

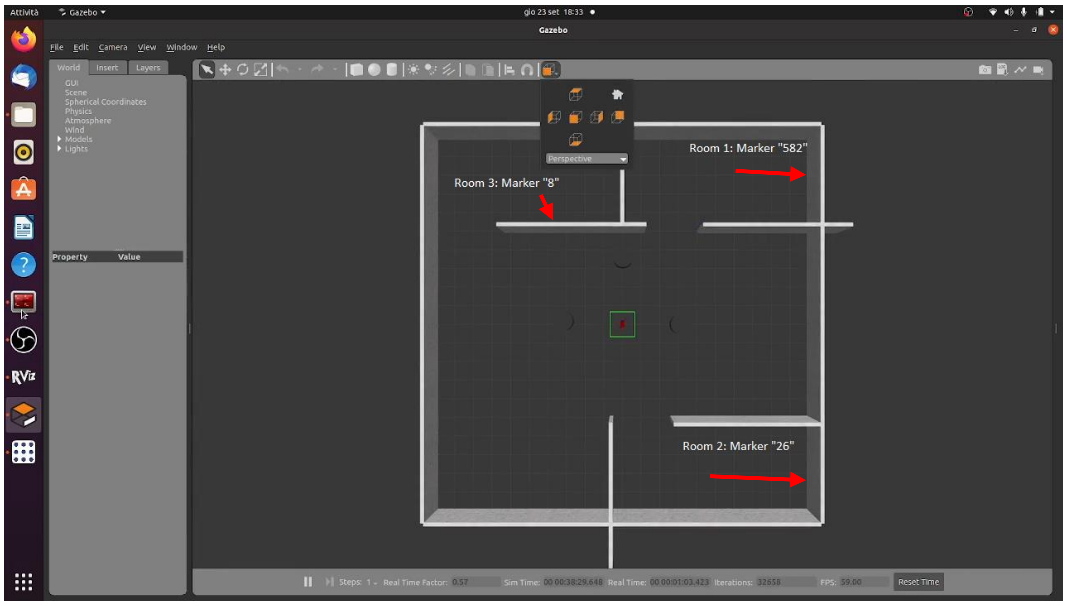


Figure 10: Ar markers positions: Ar marker with 582 id in the first room, in the second the one with the 26 id and in the third the 8 one

In order to scan the Ar marker id, the *aruco.launch* file is executed. This file defines the *markerSize* argument as 0.09 meters; it is run, by the ArUco library, the *aruco_marker_publisher.cpp* executable, allowing to detect any marker in the ArUco dictionary, and two camera topics are remapped:

/camera/rgb/camera_info	to	/camera_info;
/camera/rgb/image_raw	to	/image.

After this launch, the robot can scan ar markers' ids [Fig.11]:

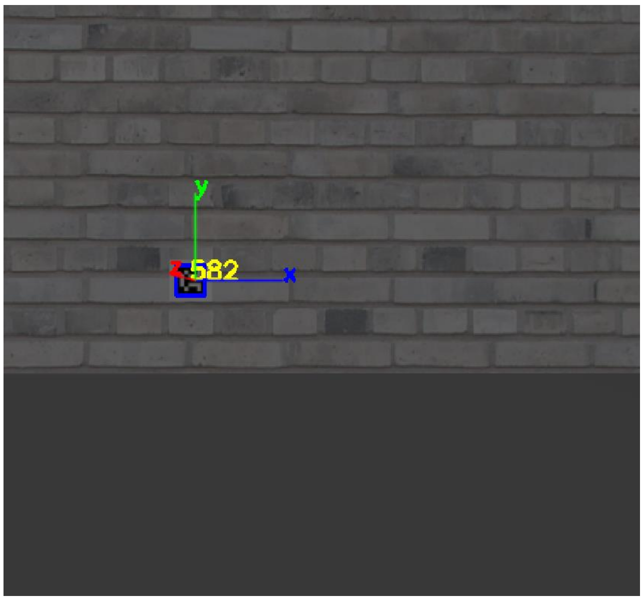


Figure 11: Through the rqt tool, it is possible to view images from topics: selecting the /aruco_marker_publisher/result topic it is showed the detection of an Ar marker with an id of 582.

After the launch, several ArUco topics were published.
The most important to the project purpose is the */aruco_marker_publisher/markers_list*, publishing the currently scanned ids through a *std_msgs::UInt32MultiArray*. Since there is only one marker in each room, it is assumed that the only non-empty item of the array is the one with the 0 index.
So this element is checked by the *marker_cb*, and it will provide the id of the scanned marker.

6 Planning and Control

The planner and the controller have both been implemented in the *navigation.cpp* file. There a *NAVIGATION* class has been developed, including the planner, the controller and three callbacks.

The planning and the control are two threads managed by the *run()* function through the *boost::thread c++* library. According to the project requirements, the planner and the controller are both threads working at 100Hz.
The planner's outputs are a velocity setpoint and a position one, both defined for the x-axis and the y-axis. The controller takes these setpoints as inputs and, after a few passages, it can command each robot's wheel.

The *run()* function also calls a *ros::spin()* function to check for the incoming messages from the odometry, lidar and vision callbacks.

The *main* function declares a *NAVIGATION* class instance and displays a message asking for the id marker to find. Then it calls the *run()* function, and the two main threads start working, blocking themselves if the odometry has not been already run. In the worst case, the navigation will proceed in the following order [Fig.12]:

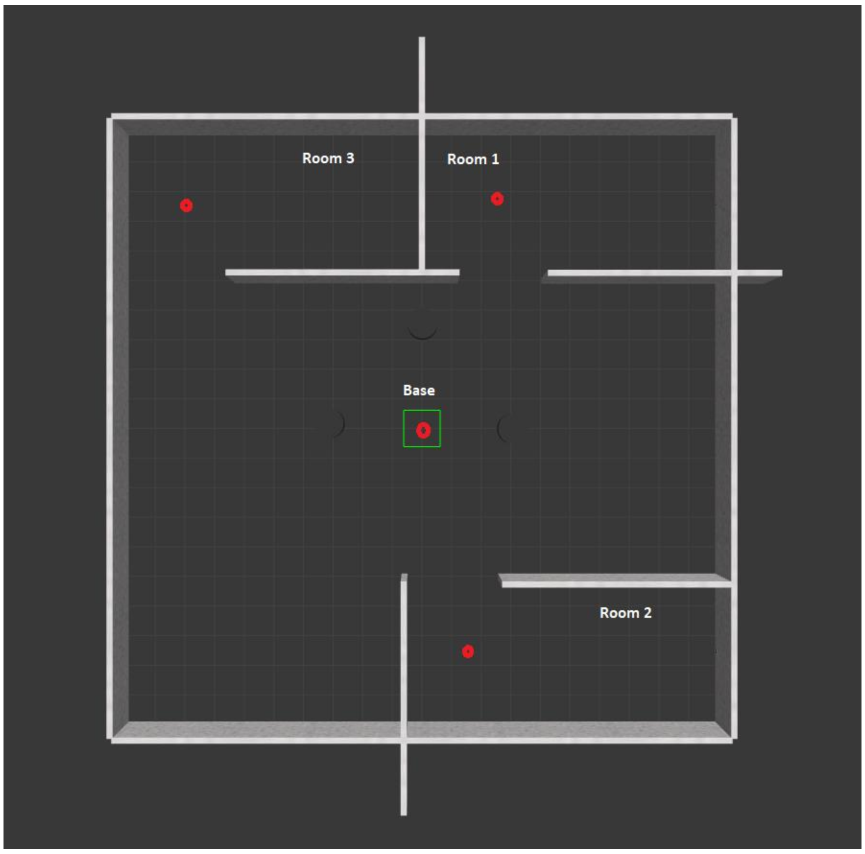


Figure 12: Rooms sequence, in red the global goals positions,

6.1 Planning

"Planning must be carried out with an online version of the artificial potential method. The force field is seen as an acceleration vector. Use random solutions to avoid possible local minima.
The robot must travel towards the locations [...] The coordinates of the locations are known."

The planner assigns the controller the trajectory to track.
As said before, the final outputs of the implemented planner are two setpoints: one in velocity and one in position. The planner has been configured as a 100Hz thread calling the artificial potential function (*art_pot()*) and the local minimum solver (*local_minima_manager()*) if a trigger is set.

The implemented solution comprises a global goals vector (*_des_q*) for the three rooms' first arrival points and the base coordinate.
Whenever an ar-marker is detected, the global goal is changed, incrementing the vector's current index.

The artificial potential method makes it possible to build an utterly online planner without any previously known environmental information, except the coordinates of the global goals.
This technique is conceptually straightforward. It allows moving the robot in the *C-free* space, driven by potential field, minimizing it by steps, through the steepest descent method.
The potential field is composed of two potentials: an attractive potential and a repulsive potential.

6.1.1 Attractive potential

The attractive potential drives the robot to the global goal according to the position error between the current robot's position, retrieved by subscribing the */odom* topic, and the goal configuration.

The attractive artificial potential field has been modelled as a union of paraboloidal and conic surfaces, taking vertex in the global goal.
The conic field has been defined as the positive definite function:

$$U_{a2}(\boldsymbol{q}) = k_a \| e(\boldsymbol{q}) \|, \tag{17}$$

where $e(\boldsymbol{q}) = \boldsymbol{q}_g - \boldsymbol{q}_{odom}$, $\boldsymbol{q}_g$ standing for the goal's position, $\boldsymbol{q}_{odom}$ standing for the robot's position retrieved by odometry, and k_a is a positive gain: it will be discussed about later.

Differentiating it, the following attractive force field is retrieved:

$$\boldsymbol{f}_{a2}(\boldsymbol{q}) = -\nabla U_{a2}(\boldsymbol{q}) = k_a \frac{e(\boldsymbol{q})}{\| e(\boldsymbol{q}) \|}. \tag{18}$$

The paraboloidal field is defined by the following:

$$U_{a1}(\boldsymbol{q}) = k_a \| e(\boldsymbol{q}) \|^2, \tag{19}$$

it is positive definite too. Besides, the related attractive force is:

$$\boldsymbol{f}_{a1}(\boldsymbol{q}) = k_a e(\boldsymbol{q}). \tag{20}$$

It is possible to aggregate the two potentials advantages: when the planning starts, $\boldsymbol{q}_g$ is far from the robot's pose and the parabolic force (seen as an acceleration vector), linearly depending on the error, would come out in a rash, assigning to the robot too high accelerations.
Instead, the attractive conic forces are constant in the module being unit vectors, allowing a more careful navigation start. However, $\boldsymbol{f}_{a2}(\boldsymbol{q})$ is not defined in the vertex ($\boldsymbol{q}_g$) so a transition to the paraboloid is necessary to take the robot to the global goal.

This switch happens when:

$$\|\boldsymbol{e}(\boldsymbol{q})\| = 1 \tag{21}$$

So that a continuous attractive force is provided for each value of $\boldsymbol{q}$.

6.1.2 Repulsive potential

Its role is to modify the attractive potential to avoid the nearby obstacles in the environment.

Since the planning must be entirely online, the repulsive potential could not be computed as shown in the classical artificial potential implementations because of the underdetermination about the *C-obstacle* region.

In the proposed solution, a lidar sensor has been employed to detect the obstacles through a *head_hokuyo_sensor* Gazebo Plugin.
This sensor projects a 180 degrees beam composed of 720 singular rays. The sensor max range value is 30 meters. For each ray, the plugin publishes on the */scan* topic the distances between the robot and a detected obstacle.
If a ray does not find any obstacle within the max range, a symbolic distance of *inf* is published.

The approach to compute the repulsive potential starts considering a symmetric 90 degrees angular sector in front of the robot, where to detect obstacles. This angular range is particularly ample due to the footprint of the wheels. If narrower, the robot would contact obstacles with one of the wheels, generating an external force and corrupting the odometry working.
It is not trivial to define the minimum distance from each obstacle because the environment is unknown, so the employed approach considers each of the ranges below an established threshold η_o of 2 meters, as a fake obstacle, giving repulsive potential.
So, when a range value in the angular sector is below η_o, i.e. the i-th range, this data is pushed in the *_min_range* vector. It can be considered as a vector composed of all the fake $\eta_i(\boldsymbol{q})$.

$\eta_i(\boldsymbol{q})$ represents:

$$\eta_i(\boldsymbol{q}) = \|\boldsymbol{q}_{robot} - \boldsymbol{q}_{obstacle}\| \tag{22}$$

Where $\boldsymbol{q}_{obstacle}$ is the fake obstacle pose.

In order to correctly retrieve $\boldsymbol{q}_{obstacle}$ a *_range_angle* data is computed as follows:

$$rangeangle_i = (i * inc) - \frac{\pi}{2} + \vartheta, \tag{23}$$

where i is the range's index which is below η_o, *inc* is the angle increment between two following rays, retrieved by the *laser_cb* as the value *laser.angle_increment* and equal to 0.004 radians, and ϑ is the robot's current yaw.

This formula allows defining an angular value for the range below the threshold coherent to the robot orientation. It is possible to demonstrate it by the following images [Fig.13, Fig.14]:

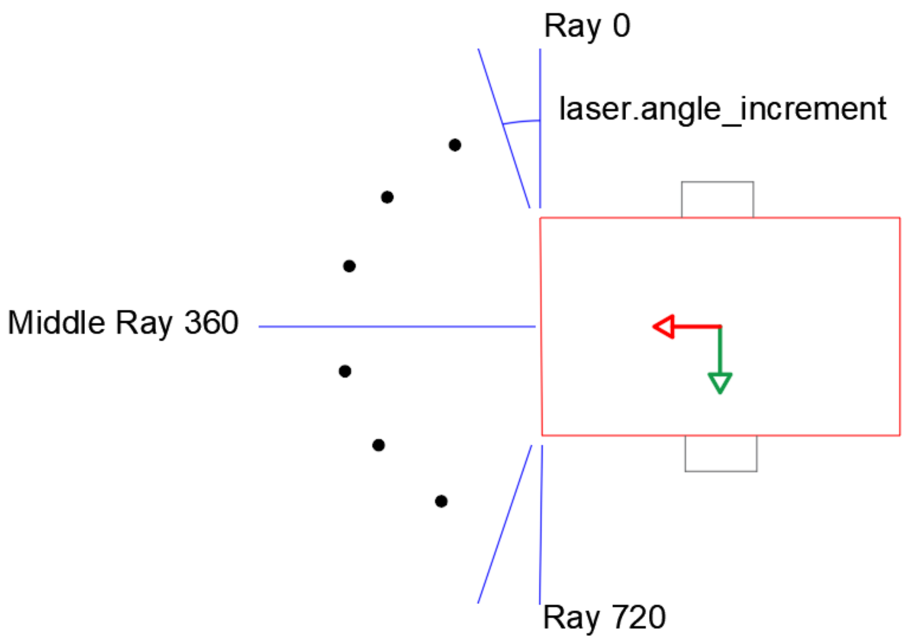


Figure 13: Example 1

In [Fig.13], the robot's yaw is 0 radians. It is convenient to focus on the middle range. Its index is 360, and its angle should be equal to the robot's yaw, while considering the angular value as $i * inc$ it results as $\frac{\pi}{2}$ radians. So, the middle range angular value could be computed as:

$$i * laser.angle_{increment} - \frac{\pi}{2} \tag{23.1}$$

It is missing another term that is not visible by this robot's pose.

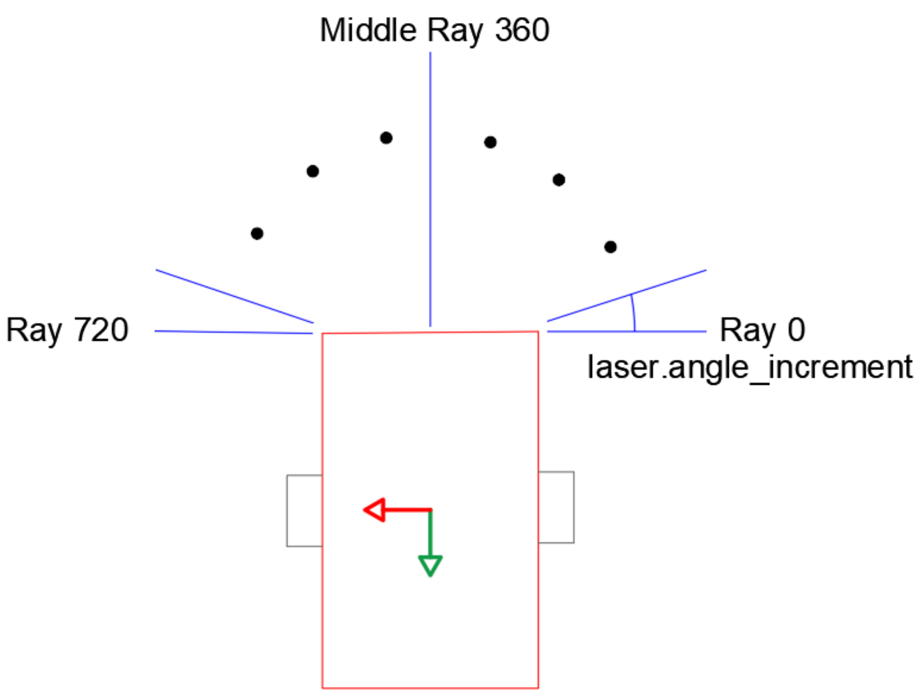


Figure 14 Example 2

In [Fig.14], the robot's yaw is $-\frac{\pi}{2}$ radians, but, using (23.1) the middle range's angle is still 0 radians. So, it is necessary to add the robot's current yaw value to the (23.1) to provide the range angle correctly. This yields to (23). This formula works for the middle range, and it will work for each range.

The *laser_cb* makes all the above calculations and provides essential data to the repulsive potential computing.

For each data in the *_min_range* vector, it is defined an aliquot of the repulsive potential.

In order to compute it, it is defined the position of the fake obstacle (remind: fake obstacle has to be referred to the i-th lidar range below the η_0 threshold), as:

$$\boldsymbol{q}_{obstacle,i} = \begin{matrix} x_{obst,i} = x_{robot} + \eta_i(\boldsymbol{q})\cos(rangeangle_i) \\ y_{obst,i} = y_{robot} + \eta_i(\boldsymbol{q})\sin(rangeangle_i) \end{matrix} \tag{24}$$

Then, for each of the fake obstacles it is defined the repulsive potential:

$$U_{r,i}(\boldsymbol{q}) = \frac{k_r}{\gamma}\left(\frac{1}{\eta_i(\boldsymbol{q})} - \frac{1}{\eta_o}\right)^{\gamma}, \tag{25}$$

where k_r is a positive gain that will be explained about later and $\gamma = 2$.

By the gradient, the associate repulsive force field is:

$$\boldsymbol{f}_{r,i}(\boldsymbol{q}) = \frac{k_r}{\eta_i(\boldsymbol{q})^2}\left(\frac{1}{\eta_i(\boldsymbol{q})} - \frac{1}{\eta_o}\right)^{\gamma-1} \nabla\eta_i(\boldsymbol{q}) \tag{26}$$

Where, given the (22):

$$\nabla\eta_i(\boldsymbol{q}) = \frac{\boldsymbol{q}_{robot} - \boldsymbol{q}_{obstacle}}{\eta_i(\boldsymbol{q})} \tag{27}$$

So, to retrieve the total repulsive potential, it is sufficient to sum the results of the above calculations for each fake obstacle:

$$U_r(\boldsymbol{q}) = \sum_{i=1}^{p} U_{r,i}(\boldsymbol{q}) \tag{28}$$

$$\boldsymbol{f}_r(\boldsymbol{q}) = \sum_{i=1}^{p} \boldsymbol{f}_{r,i}(\boldsymbol{q}) \tag{29}$$

Where p is the number of the ranges below the range of influence η_o.

6.1.3 Total potential & planning technique

The total force field is computed as the sum of the attractive and repulsive force fields at each iteration, and it is considered an acceleration vector:

$$U_t(\boldsymbol{q}) = U_a(\boldsymbol{q}) + \sum_{i=1}^{p} U_{r,i}(\boldsymbol{q}) \tag{30}$$

$$\boldsymbol{f}_t(\boldsymbol{q}) = \boldsymbol{f}_a(\boldsymbol{q}) + \sum_{i=1}^{p} \boldsymbol{f}_{r,i}(\boldsymbol{q}) \tag{31}$$

Since the implemented controller requires velocities and positions as inputs, it is necessary to integrate the accelerations twice.

In this way, the planner defines two set points for the controller, one in velocity, $\boldsymbol{sp}_{\dot{\boldsymbol{q}}}$ and one in position, $\boldsymbol{sp}_{\boldsymbol{q}}$ (the local goal):

$$\boldsymbol{sp}_{\dot{\boldsymbol{q}},k+1} = \boldsymbol{sp}_{\dot{\boldsymbol{q}},k} + \boldsymbol{f}_t(\boldsymbol{q})dt \tag{32}$$

$$\boldsymbol{sp}_{\boldsymbol{q},k+1} = \boldsymbol{sp}_{\boldsymbol{q},k} + \boldsymbol{sp}_{\dot{\boldsymbol{q}},k}dt \tag{33}$$

Where dt is the planner’s and controller’s sample time 10 ms.

It is intuitable that the robot would stop as $\boldsymbol{sp}_{\dot{\boldsymbol{q}}}$ is 0, but the robot is also wanted to stop at $\boldsymbol{q}_g$. At the global goal the acceleration field computed by the planner becomes 0, but the $\boldsymbol{sp}_{\dot{\boldsymbol{q}}}$ is not 0. So the robot would go on, and as the acceleration field gets negative, the integrator discharges itself. When the $\boldsymbol{sp}_{\dot{\boldsymbol{q}}}$ is 0, there would be a new attractive potential since the norm of the position error is not null, and the robot would move back. Besides, in brief, the robot would fluctuate around $\boldsymbol{q}_g$ without stopping.

To avoid this effect and to reach asymptotic stability, it is necessary to add a damping term to the acceleration field; this is proportional to the current robot's velocity, and it is defined as follows:

$$damping = -k\dot{\boldsymbol{q}} \tag{34}$$

Where k is a positive gain that will be discussed further.

Thanks to the damping term, the robot can stop at the global goal.

6.1.4 Gains tuning

In the section below there have been presented three different gains: k_a (attractive potential), k_r (repulsive potential), and k (damping term). These gains have been tuned through a trial and error approach. After various simulations, they have been set as:

$$k_a = 1.6$$
$$k_r = 0.08$$
$$k = 2.5$$

With these values, the robot navigates quite quickly and falls in local minima (Section 6.1.6) at a 0.6 distance from a wall.

6.1.5 Room exploration

Once the robot gets into the global goal, it reaches a room.
At this time, the robot has to find the ar-marker.

In order to make the exploration robust regardless of the marker position, a set of manoeuvres has been designed.
The former one is a 360 degrees turning implemented through the *rotate()* function. It is a blocking function checking when the current orientation becomes equal to the starting orientation plus the given angle, 2π in this case.
The latter one is a straight motion obtained overwriting the global goal with a new one in which the y value is decreased by 1. In this way, the robot will alternate turning 360 degrees and moving straight of 1 meter until the ar marker is detected.
This manoeuvre works well in the given environment and is designed thanks to the rooms' coordinates information: the door's position is always higher in y value than the end of the room.

Since the straight motion is obtained by modifying the *art_pot's* function goal and planned with artificial potential, the exploration is also obstacle safe.

6.1.6 Local minima problem

Until now, the artificial potential planning method could seem the perfect planning technique.
It allows online planning without any information about the environment or the obstacles poses; it is also easy to implement and does not require SLAM.

However, this planning technique suffers from a critical downside: local minima.

Since the gradient of the sum between an attractive and a repulsive potential gives the force field, the two forces could be the same, providing a null force field while the total potential field is still positive.
It could also happen in positions far from the global goal. These potions in which the robot could stop its navigation are defined as local minima.

In order to manage local minima, literature provides different solutions, some of which are exceptionally ingenious, but, in this case, it has to be employed the random solutions one.
With this approach, it is possible not to use any environmental information and the robot is commanded to do a random movement, after which the potential computing restarts.

Wishing a lower simulation time, this manoeuvre is not implemented as a completely random movement. Indeed, it is undetermined if the robot would fall in other local minima as the manoeuvre is completed.
Therefore, to lower this probability, it has been designed a manoeuver bringing the robot closer to the door.

First of all, it is necessary to recognize when the robot is in the attraction basin of a local minimum: at each iteration, the *art_pot()* function checks if the robot is in a local minimum by the following condition:

$$\begin{cases} |\boldsymbol{f}_t(\boldsymbol{q})| < 0.05 \\ U_t(\boldsymbol{q}) > 0.5 \\ the\ robot\ is\ not\ at\ the\ navigation'start \end{cases} \tag{35}$$

The last condition allows not to mistake the navigation's start with a local minimum. The second allows differentiating a local minimum from the global goal $\boldsymbol{q}_g$ arrival.

When the robot falls in this trap, a *_local_minima* boolean variable is set to true.
As this happens, the planner thread would not call the *art_pot()* function but the *local_minima_manager()* function, performing a maneuver to drive the robot away from the local minimum.
This manoeuvre is a "[" or a "]" trajectory shape, composed of three *move* functions. The *move* function is a blocking function defining fixed setpoints to the controller.

It is assumed that at a global minimum, the robot is in front of an obstacle, so it has to move back to get away from it: a first *move* function on the x-axis is called until the robot gets 1 meter back.
The whole manoeuvre aim is to drive the robot closer to the door. After various simulations, it is noticed that the robot falls in a local minimum in an upper position than the door's one, so the robot has to go down. A second *move* function is called to move the robot down on the y-axis of 3 meters.
Finally, to make the x-axis value equal to the one at the start, a third symmetric move is performed on the x-axis forward of 1 meter.
The move function works overwriting the setpoints also provided by the *art_pot()* function.

To allow, also during a global minimum managing, an obstacle safe navigation, the move function would stop if there is an obstacle below 0.4 meters through a *_local_obstacle* class trigger valued in the *laser_cb* callback.

In order to reset the artificial potential computing, at the end of the *local_minima_manager()* function, the velocity setpoint is set to 0, the position one to the current position.

Finally, the *_local_minima* variable is set to false, allowing the planner thread to call the *art_pot()* function again.

6.2 Control

"Free to be chosen among the tracking and/or the regulation controllers illustrated during the course. The controller sample time is 10 ms."

The implemented controller is a double-degree controller. A low-level controller computes the wheels' velocities according to the planning, and a high-level controller integrated into each actuator cranks them up.

6.2.1 High-level controller

The high-level controller allows moving the robot in the Gazebo-ROS environment. It has been implemented through a *gazebo_ros_control* plugin. It could be seen as the wheel's servo motor controller or a motor driver. Since the robot is differential driven, two motors are required separately actuating the wheels.

In order to implement them, two transmissions have been added to the robot description *.xacro* file. They are two *SimpleTransmission* named *left/right_wheel_hinge_trans*.
They are applied to the *left/right_wheel_hinge* joints, configured as two *hardware_interface/VelocityJointInterface,* so the robot could be controlled in velocity and not in torque, and actuated by the *left/right_wheel_joint_motor* with a mechanical reduction of 1.

Even if this mechanical transmission has a neautral reduction ratio, the whole engine is symbolic since it is employed just to move the robot in the Gazebo environment, and the mechanical effects are not deeply analyzed.
In the *conf* folder of the package, it has been coded a *.yaml* file to define the high-level controllers' specs.

They are two PID controllers with gains:

$$P = 100; \qquad I = 0.01; \qquad D = 10.$$

These values are suggested by the Gazebo *Tutorial: ROS Control.*

Moreover, it is also configured a *joint_state_controller* publishing, with a 50Hz rate, the wheels' angular positions and velocities. It could be

employed as encoders for odometry, but it has been better to develop the specific plugin for encoders to completely avoid Gazebo's information. Once the robot is launched in its environment three topics arise in the *rostopic list*:
/left_wheel_velocity_controller/command,
/right_wheel_velocity_controller/command,
/joint_states.

The first couple of topics allow sending independent velocities to the two wheels accepting as inputs types *std_msgs/Float64*, and those could be published from shell through a *rostopic pub* command or from a publishing topic (the low-level controller).

The */joint_states* topic has been employed to compare the wheels velocities published by the self-implemented plugin and check for its correctness [Fig.15].

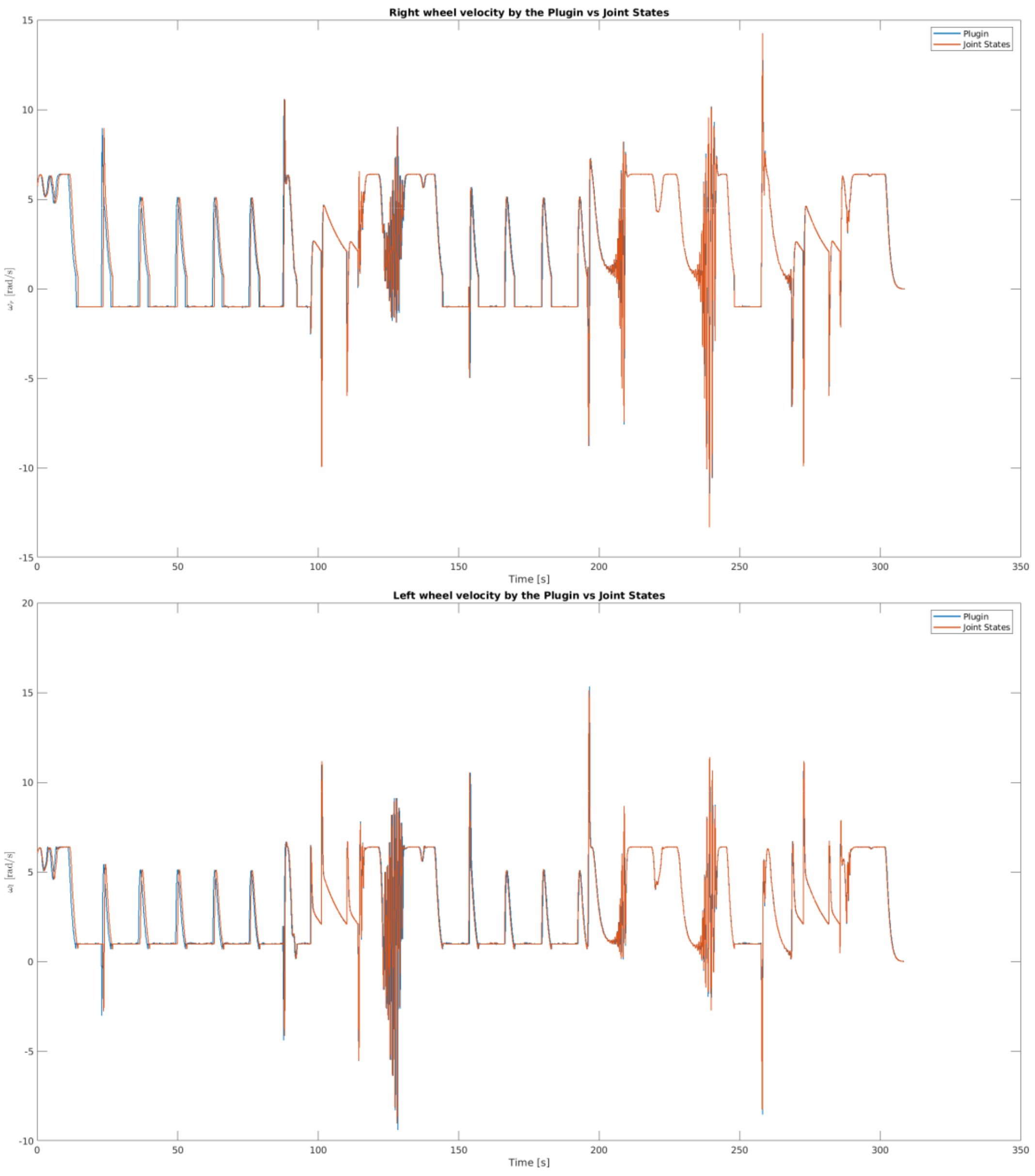


Figure 15: The wheels velocities published by the self-implemented plugin and by the JointStates overlap, as expected

6.2.2 Low-level controller

The low-level controller is an Input-Output linearization system. Among the wide range of controllers illustrated during the course, it has been selected for different reasons.

Why a tracking controller and not a regulation?
The artificial potential planning carries out a trajectory (velocities + positions), even if it is an online computed trajectory, processed at each sample step. So also a regulation could have been employed, using as inputs only the planner's positional outputs. However, a tracking controller better suits the whole planning output than a regulation.

Among all the trackers, why an Input-Output linearization controller?
On this one, a structural tracking trouble is inherent to the unicycle (and differential drive robot) because of the non-holonomic pure rolling constraint.
It is not possible to track the entire configuration space $\boldsymbol{q} = [x \quad y \quad \vartheta]^T$, assigning non-persistent trajectories. Therefore, there is a choice to be made. Is it better tracking the whole $\boldsymbol{q}$ with persistent trajectories (through a nonlinear control or an approximate linearization controller) than controlling only x and y with non-persistent trajectories (through an Input-Output linearization control)?
During the preliminary solution sketch, it is noticed that the planned trajectory would not be user-defined but computed via an artificial potential criteria, making the resulting trajectory not precisely known beforehand and not editable. It cannot be ruled out that the trajectory could converge to zero during motion inversions, cusps, or when a global goal is reached. Therefore, being the trajectories possibly non-persistent, the implemented controller is an I-O linearization system.
This choice imposes not to control the whole $\boldsymbol{q}$ but only the x and y values, managing the ϑ yaw value numerically, when needed a fixed angle rotation.

During the control, the robot is considered as the equivalent unicycle. In the end, the v and ω computed values are mapped to the ω_l and ω_r differential robot's wheels velocities, through (1) and (2) reverse.

While the planner defines a trajectory for the C point (Chapter 1), the controller defines a B point at a b distance along the sagittal axis whose coordinates on the x-axis and y-axis are:

$$y_1 = x + bcos(\vartheta) \tag{36}$$
$$y_2 = y + bsin(\vartheta), \tag{37}$$

where x and y are the C point coordinates, and ϑ is the yaw value, all taken by the odometry; in the case at hand $b = 0.1$.

The low-level controller inputs velocity and position setpoints computed by the planner, whether from the *art_pot()* function or the *local_minima_manager()* function. As the position setpoint is the C point's local goal, it is mapped on the B point:

$$y_{1,g} = sp_x + bcos(\vartheta) \tag{38}$$
$$y_{2,g} = sp_y + bsin(\vartheta), \tag{39}$$

while the velocity setpoint remains unaltered.

Then two virtual inputs are defined as:

$$u_1 = sp_{\dot{x}} + k_1(y_{1,g} - y_1) \tag{40}$$
$$u_2 = sp_{\dot{y}} + k_2(y_{2,g} - y_2) \tag{41}$$

Where the k_1 and k_2 gains are both tuned to 0.1 through a trial and error technique.

Lastly, the heading and the steering velocities are computed as:

$$v = \cos(\vartheta)\, u_1 + \sin(\vartheta)\, u_2 \tag{42}$$

$$\omega = -\frac{\sin(\vartheta)}{b} u_1 + \frac{\cos(\vartheta)}{b} u_2. \tag{43}$$

These are then mapped to the wheels velocities of the differential robot by:

$$\omega_l = \frac{v}{\rho_w} - \frac{d_w \omega}{2\rho_w} \tag{44}$$

$$\omega_r = \frac{v}{\rho_w} + \frac{d_w \omega}{2\rho_w}, \tag{45}$$

and finally published to the */left_wheel_velocity_controller/command* and the */right_wheel_velocity_controller/command* topics, allowing the robot's motion.

The robot rotation trajectories had to be managed differently.
During the exploration task, the robot has to rotate 360 degrees to search for the Id marker.
How to command a rotation if the implemented controller does not allow defining a setpoint angle?
As before said, it is necessary a numerical management to define arbitrary rotations.

The *rotate()* function has been implemented in the planner [6.1.4 section], to allow this kind of motion. When this blocking function works, it sets to true a *rotate* trigger. While the trigger is on, the controller stops the standard functioning, directly defining v and ω: assigning them respectively a null value and a 0.5 rad/s value, a turning motion is designed. When the goal angle is reached, the *rotate()* function pulls down the trigger, allowing coming back to the controller standard routine.
The rotation task management is the only section of the code in which the controller does not work on the setpoints provided by the planner.

7 Mapping

"After the end of the task, the map of the inspected locations must be generated."

To generate the inspected location map, the *gmapping* package has been employed. This is a particle filter in which each particle carries information about the map of the environment. Using this package it can be created the *2d occupancy grid map* of the environment fusing laser data and pose data retrieved by the odometry. It could be employed also to solve a SLAM problem, but it's not the case.
This package relates each scan data to the odometry transform frame. So it requires the *tf* between robot's base frame (*robot_footprint*) and the odometry frame (*odom*) and provides a *tf* between the map and the odometry frames. All the required data are provided in the *gmapping.launch* file. When the navigation is completed the *grid map* could be saved through the *map_server* package.
Once created the map environment, it could be used to make smarter the planning system with environmental information. The following is the generated map of the environment [Fig.16]:

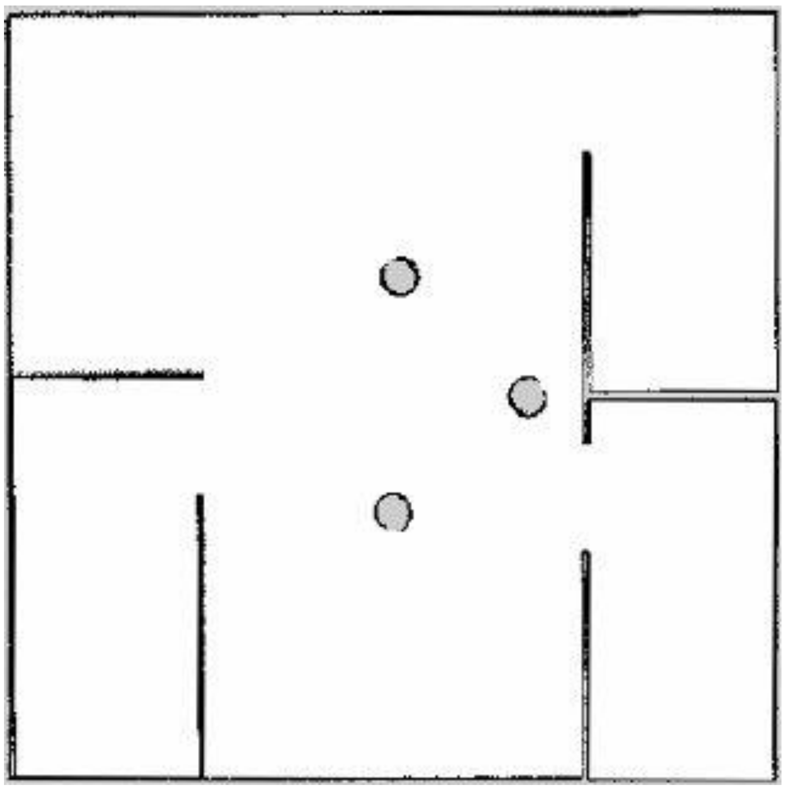


Figure 16: Environment generated map

8 Simulation example

The following simulation has been performed assuming the worst navigation case. As the marker id 8 is placed in room 3, the id to search is set to 8. In this way, the robot will explore the whole environment. The whole simulation lasts 316 seconds. However, the simulation time could vary, depending on local minima traps fallings and other factors.

To plot the following graphs a *rosbag* file has been defined, where all the topics' messages of interest are stored. Then, this *rosbag* file has been opened in the *Matlab* suite and all its content has been collected in different arrays, used to execute the *plot* command.

This is the path accomplished by the robot in the implemented environment estimated by the */odom* and the */gazebo/model_states* topics [Fig.17]:

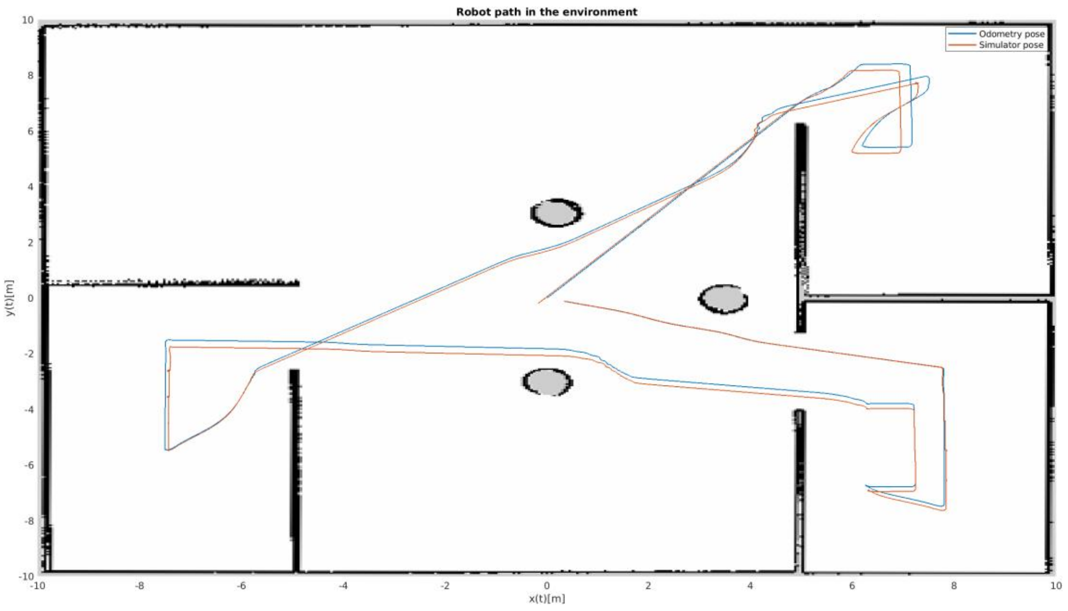


Figure 17: The robot path according to the /odom and the /gazebo/model_states topics

As noticeable from the graph the robot well avoids all the obstacles, whether they be cylinders or walls.

It is also evident a certain displacement among the two pose estimation systems. To value the position error between the implemented odometry and the pose retrieved from the simulator it has been plotted the following graph [Fig.18].

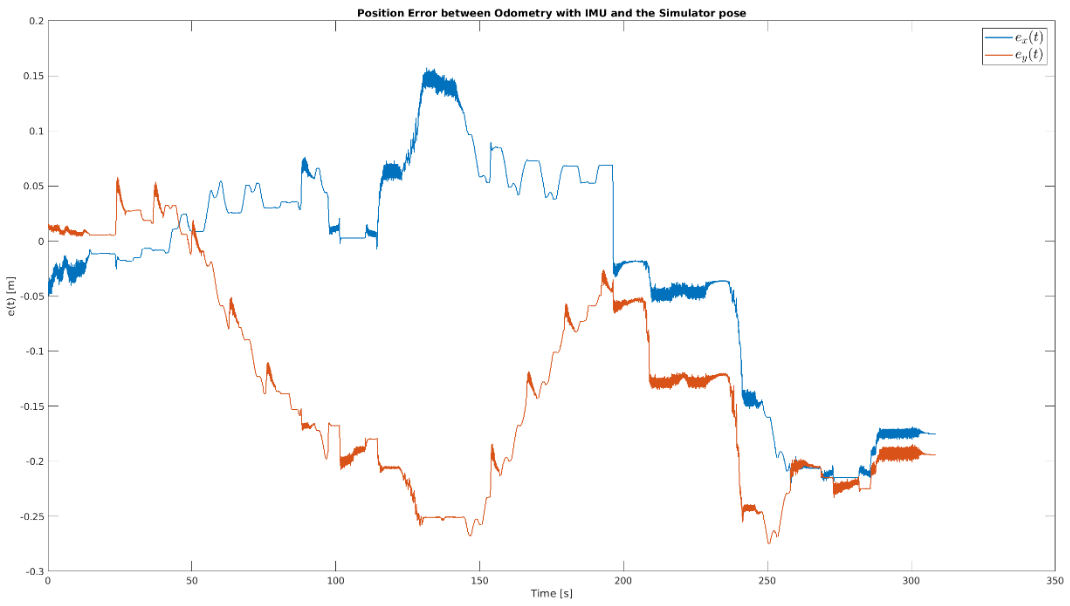


Figure 18: Error on the x and y coordinates dynamics

It is possible to evince that since these errors average values are below the robot's length (0.4 meters), they do not at all undermine the whole navigation accomplishment.

To keep down these errors it is essential to have excellent esteem of the yaw angle and the IMU sensor allows it, as the following [Fig.19] proves:

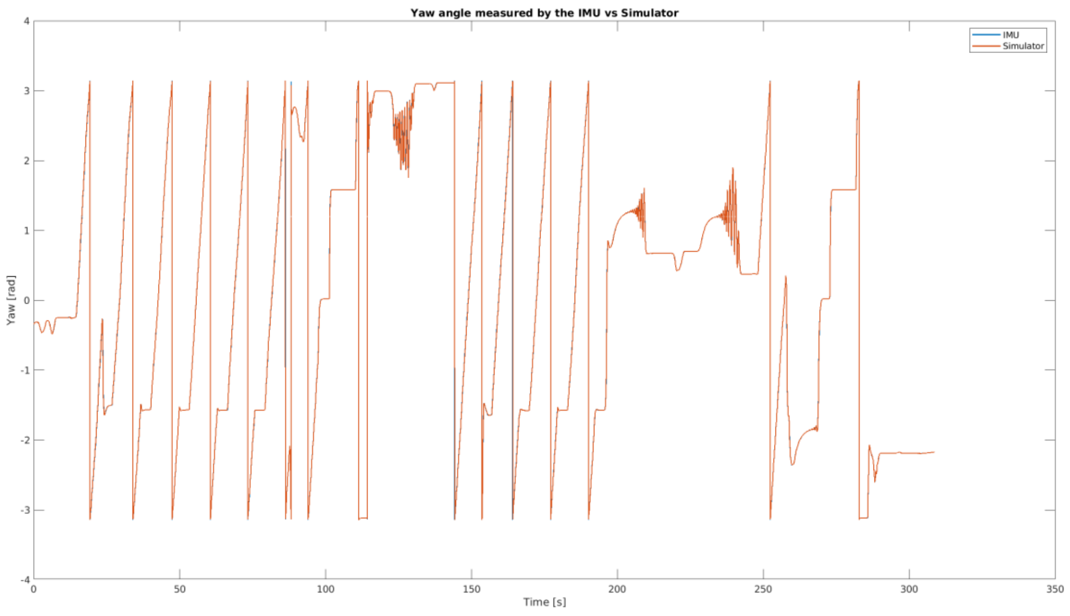


Figure 19: The yaw value from the IMU sensor and the Simulator's one. The two estimated angles overlap perfectly.

In the next section, the controller's behaviour will be displayed through different plots. The former one represents the tracking error [Fig.20]:

$$e = q_{local\ goal} - q_{robot} \tag{46}$$

where both the values are mapped on the B point. The latter one displays the wheels velocities commanded from the low-level controller [Fig.21].

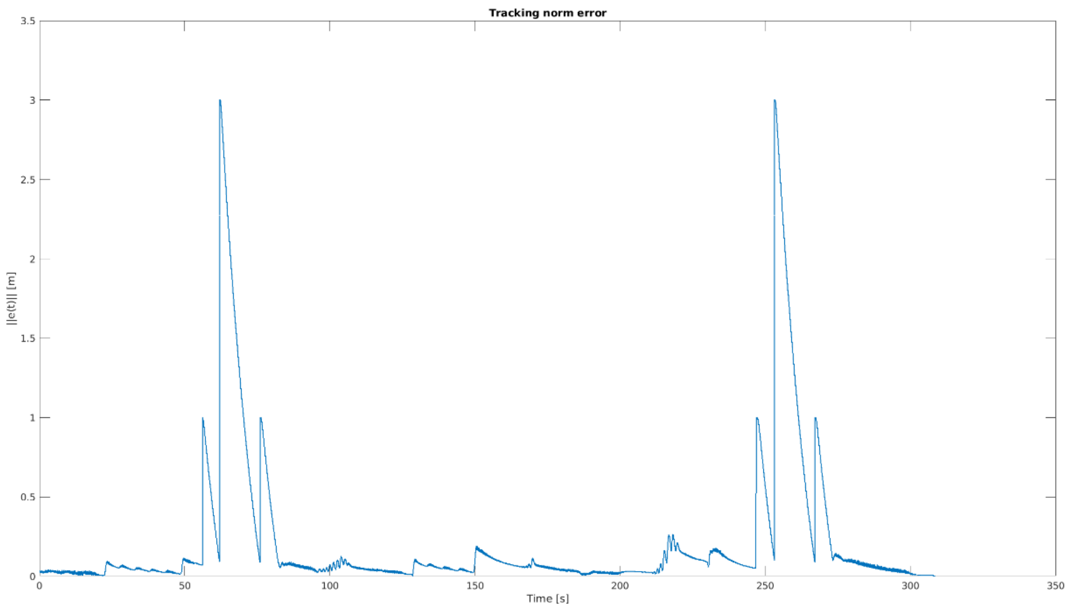


Figure 20: Controller tracking error dynamics

There is evinced that the controller asymptotically stabilizes the error dynamics. The peaks are due to:

- Global goal changing;
- Local minima management;
- Rotation management.

The tracking norm error is always below 3 meters but the transitional drift error is widely recovered at regimen.

The following are the wheels velocities [Fig.21]:

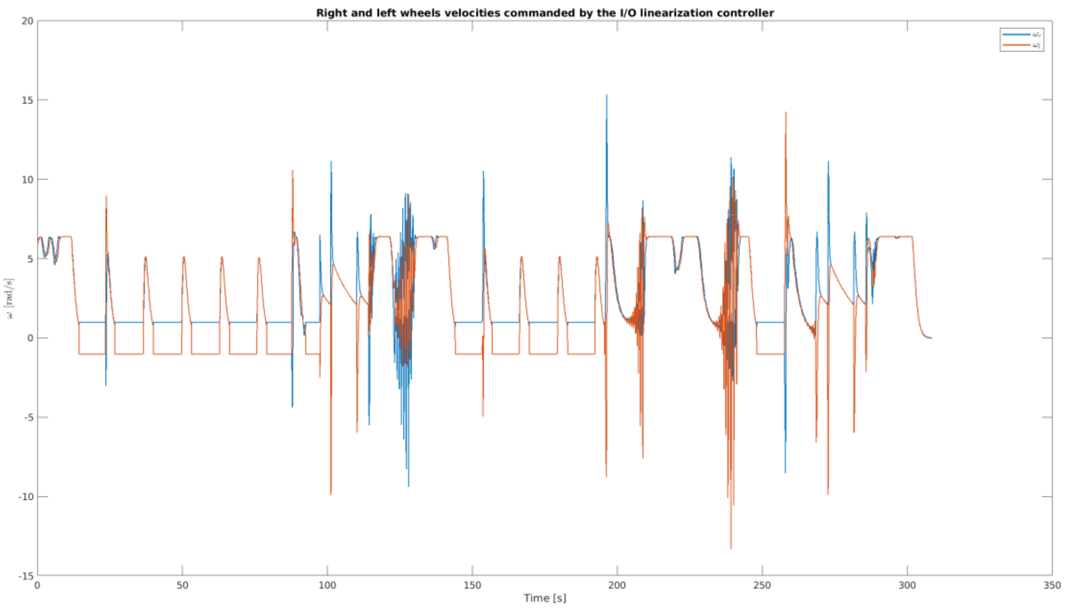


Figure 21: Left and right wheels angular velocities

There are different issues to pick about [Fig.21].

The discontinuities are caused by rotations and local minima management. When the robot has to turn the controller commands the wheels through a fixed steering velocity (0.5 rad/s) and a null heading velocity. While when the robot falls in a local minimum trap, the manager drives the robot with fixed velocity setpoints. In this way, there are introduced discontinuities in the velocity profile. It could be useful a filter to interpolate the displacement causing the discontinuity.

The oscillations are the other, in addition to local minima traps, drawback of the artificial potential planning. In presence of obstacles or narrow passages, the artificial potential planner delivers oscillations to the robot, since the repulsive potential modifies the total potential as the robot gets closer to an obstacle.

In the simulation at hand, oscillations happen when the robot gets close to the obstacle with [0 3] coordinates once came out of room 1, when it is getting out of the room 2, and when arriving to the room 3.

To solve this kind of fault literature provides different solutions. These are mainly refinements to the classical artificial potentials approach, adding some environmental information or particular filters as in *Artificial Potential Field with Discrete Map Transformation for Feasible Indoor Path Planning* (M. Zulfaqar Azmi, T.Ito).

All things considered, the controller and the planner well work synergically and allow the robot to succeed in the navigation.

Conclusions and possible future developments

The autonomous navigation capability is one of the modern research main topics of study. There have been produced so many high-level libraries to solve the navigation problem. However, the employed approach from scratch allows a deeper understanding of the planning and control algorithms workings. The writer hopes that the results achieved could be satisfying, whether to wish that the present report would be clear in each of its sections.

The implemented work tickles the imagination about possible future developments and improvings.

At first, a compensation of the uncertainty about wheelbase odometry error could be performed to make due with only passive odometry.

Then, a velocity filter could be implemented to avoid the wheels velocities discontinuities.

In addition, a development in the planning technique could be implemented according to the last literature studies, to manage the local minima problem or the oscillations inherent to the APF planner.

Finally, a smarter planning system could be employed fusing APF and other offline planning techniques, once the environment map has been retrieved to better avoid also moving obstacles.

Thanks for reading.

Paolino De Risi

References

1. *Robotics Modelling, Planning and Control* (B. Siciliano, L. Sciavicco, L. Villani, G. Oriolo)

2. *IEE Transactions on Robotics and Automation, Vol 12, No 5, October 1996*

3. *Artificial Potential Field with Discrete Map Transformation for Feasible Indoor Path Planning* (M. Zulfaqar Azmi, T.Ito)