POLITECNICO DI BARI

DIPARTIMENTO DI INGEGNERIA ELETTRICA E DELL'INFORMAZIONE

Corso di Laurea Magistrale in Ingegneria dell'Automazione - Robotics

Project in

ROBOTICS – MOBILE AND FIELD ROBOTICS

AUTOPARK

Students:

Luca MURANI

Mariangela STRAGAPEDE

Angela Maria TARTARELLI

Michele VENDOLA

Prof:

Ing. Luca DE CICCO

Ing. Carlo CROCE

A.A. 2020-2021

Table of contents

Abstract

The aim of this project is to design and develop a car-like robot with an autonomous parking system.

Initially the basic world is designed, where the parking grids are placed. Subsequently the code used to obtain the random spawn of the cars in the free parking slots is deployed.

Moreover a car-like robot is designed with an Ackermann steering controller using a LaserScan to detect an eventual free spot.

The car starts a parallel parking maneuver when the free spot is detect, this maneuver includes a trajectory controlled by an hybrid controller.

World Design

The gazebo world designed for this project can be viewed by launching the *autopark_car.launch* file in the *autopark* package. In this .launch file there is the custom_world.world file, which contains the basic world (see Figure 1).

Next in the file are called the *store_model_poses_node* (pkg="pose") and *object_spawner_node* (pkg="object_spawner") that will be used to spawn the cars in the basic world.

Finally, the *controller_spawner* (pkg="controller_manager") and *cmdvel2gazebo* (pkg="autopark") nodes are called, which are dedicated to the correct operation of the car-like robot controller (Ferrari).

Custom_world.world

The base world has been designed so that there are 28 total parking spots on the 4 sides of the square, so 7 parking slots per side. The *custom_world.world* is an xml file in which are defined the positions of the various sdf models (created in the *model* folder) that will be used to build the world.

Initially this file has been written in a *world.xmacro* language with the addition of xacro relative to the measurements and positions of the parking slots, in order to lighten and parameterize the writing of the code. Finally the file was converted to a *.world* file.

The first model mentioned in the file is the *parcheggio* model that is recalled 28 times in the various configurations that will be used to subsequently determine the parking grid, each parking (white stripes included) has a dimension of $5m\ x\ 2.5m$.

Subsequently in the file the *asfalto* model, of dimension $58m\ x\ 9m$, comes recalled 4 times and, such models, are positioned in such way to obtain the squared shape of the parking spot (see Figure 1).

Finally, the *pratino* model of size $100m\ x\ 100m$ is inserted.

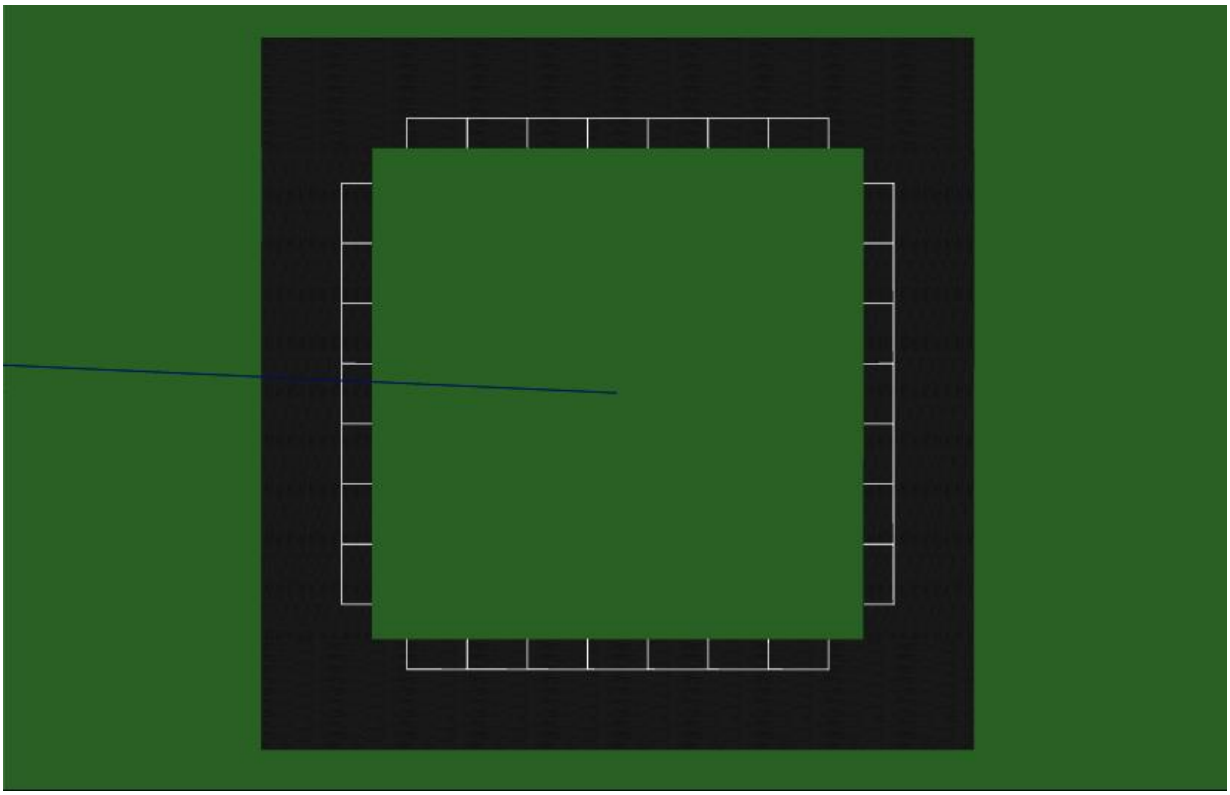

Figure 1 Basic World

Store_model_pose.py

The Python file *store_model_poses.py* is used to create the *parcheggi.yaml* file (pkg=object_spawner).

Initially the *GazeboModel* class from the *get_model_pose.py* is imported into the file.

The file retrieved through *gazebo_msgs.msg* creates a list containing the pose of the models loaded on Gazebo.

In the *store_model_poses.py* file, once the class with the various poses is loaded, a list is created containing all the names and poses of the parking spots in the *.world* file and the pose of the Ferrari.

Then the access to the list is randomized in order to make random the number and the spawn of Prius in the basic world.

Note that it is possible to spawn models of vehicles other than Prius, but this was not performed in the current project due to the computational load resulting from the overly detailed models and the limited power of the virtual machines used.

Finally, as pointed out earlier, the created list is written to the *.yaml* file so that it can be accessed later.

Object_spawner.py

The python file *object_spawner.py* accesses the previously created parking.yaml file, reading the model name and the respective pose. Next, the script accesses the *sdf* and *urdf* models in order to properly spawn the vehicles in the world.

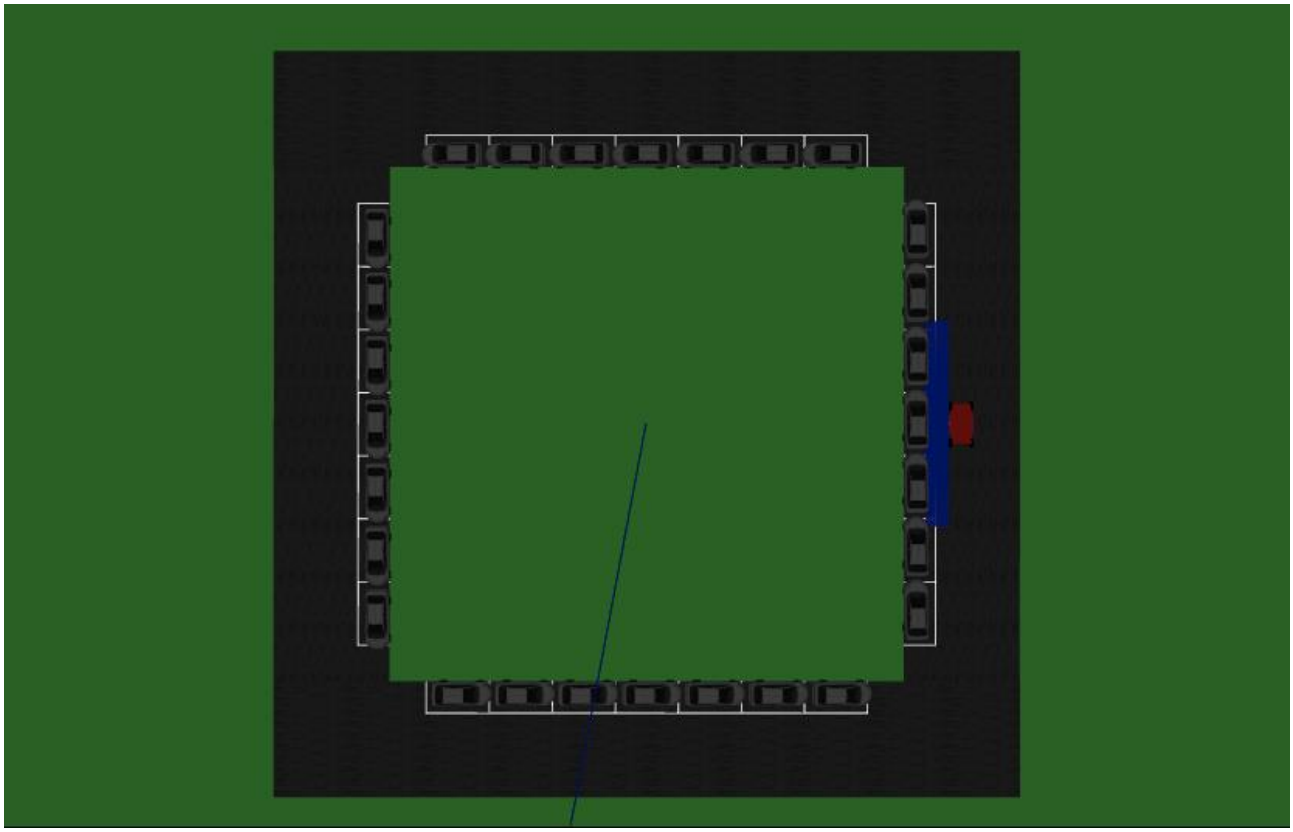

Figure 2 Final world

URDF model Car-Like robot

The construction of the physical model of the car has been implemented by writing xml files and using xacro language. In particular, 4 xml files have been produced:

- wheel.xacro: file for wheel generation. The file also use a mesh for the visualization of a realistic wheel model
- chassis.xacro: file to generate the car chassis
- ferrari.gazebo: file where the dynamic parameters of the various joints used in the model are declared and the plugins used to allow the control of the joints.
- ferrari.xacro: file that generates the complete physical model of the car, in which the transmissions are also included.

Basically, The structure is composed by a chassis, a rear_axle, two solid blocks (which represent the steering mechanism of the front wheels) and four wheels. Therefore, different types of joints were necessary:

- fixed joint connecting the dummy solid "map" to the chassis
- fixed joint connecting the rear_axle to the chassis.
- steering joints revolute, connecting the front solid steering blocks to the chassis
- revolute wheels joints, connecting the two rear wheels to the rear_axle and the two front wheels to the solid steering blocks

The actuated joints are the rear_joints (rear wheel drive cars) and the steering_joints. In particular, it has been necessary to insert in the file 'ferrari.xacro' two transmissions of type 'hardware_interface': 'hardware_interface/VelocityJointInterface' (rear_joints) and 'hardware_interface/PositionJointInterface' (steering_joints).

A complete visualization of the physical model of the car is shown in Figure 3.

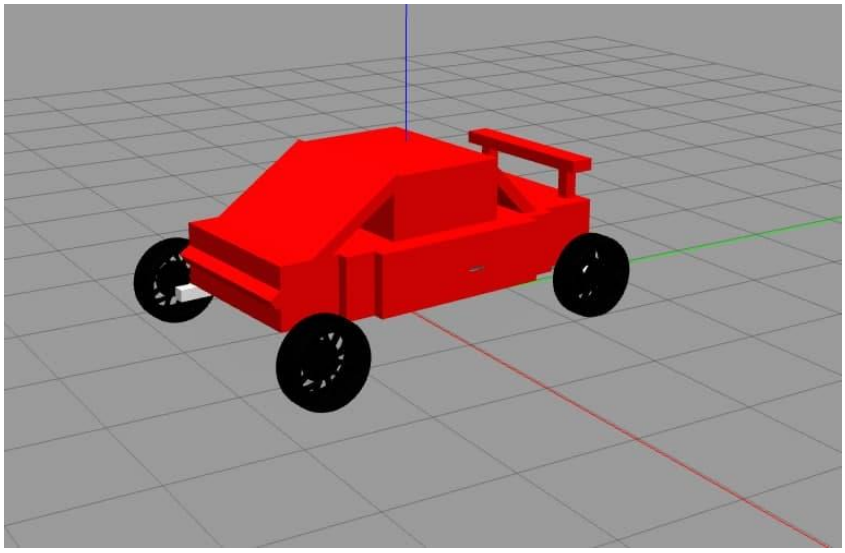

Figure 3 Car in Gazebo

In the file ferrari.gazebo are declared, specifically, the plugin "joint_state_publisher" that enables the controller to interact with the joints declared into <jointName> field and the plugin "gazebo_ros_control" that enables the initialization of a controller in Gazebo.

In addition, there is the file 'ferrari_control.yaml' in which the parameters of the low-level PID controllers that provide control of the rear_left_wheel_joint, rear_right_wheel_joint, front_left_steering_joint and front_right_steering_joint are declared.

The file 'ferrari.xacro' was finally converted into an urdf file (without xacro type syntax) throught the shell in order to enable the spawn of the urdf model in the Gazebo environment.

The randomic spawn in Gazebo environment is done, as for the sdf models, thanks to the "pose" and "object_spawner" packages. In particular, in the file 'store_model_poses.py' is generated, through the function 'def get_square_pose(self)', the random position that the car must assume in the world at spawn time. Depending on the side of the square-shaped road where the random point of coordinates (x,y) falls, a θ is assigned for the orientation of the vehicle, taking into account that the vehicle will travel clockwise on the road.

Controller

The strong assumption made was to reduce the Ackermann steering model to a unicycle by considering instantaneous steering. Therefore, starting from the linear velocity and angular velocity characteristic of the unicycle, the velocities of the rear wheels and the steering angles of the front wheels were derived. The kinematic relations governing the motion of the automobile are illustrated below.

The simplified kinematic model is expressed by the following system of equations:

$$\begin{cases} \dot{x} = v * \cos\theta \\ \dot{y} = v * \sin\theta \\ \omega = \dfrac{\mathrm{v} * \tan\Psi}{l} \end{cases}$$

Starting from Figure 4, through simple geometric observations and considering the kinematic model expressed above, it was possible to derive the following relationships:

$$\Psi_m = arctan2(l\omega, v)$$

$$r_m = \frac{l}{\tan(\Psi_m)}$$

setting:

$$r_1 = r_m - \frac{d}{2} \qquad r_2 = r_m + \frac{d}{2}$$

During steering, i.e., when the angular velocity is non-zero, the linear velocities for the left and right wheels result, respectively:

$$v_L = v\frac{r_1}{r_m} \qquad v_R = v\frac{r_2}{r_m}$$

While the steering angles result:

$$\Psi_L = arctan2\ (l, r_1) \qquad \Psi_R = arctan2\ (l, r_2)$$

In case the angular velocity is null then the speeds of the two wheels are equal and equal to v.

The code that implements the firmware foresees a maximum steering angle equal to 1.2 rad. So, in case one of the two steering angles is bigger than the limit angle, the steering cannot be correctly executed because the angle exceeds the limits of the operating space. Therefore, the car will steer with the maximum allowed steering angle.

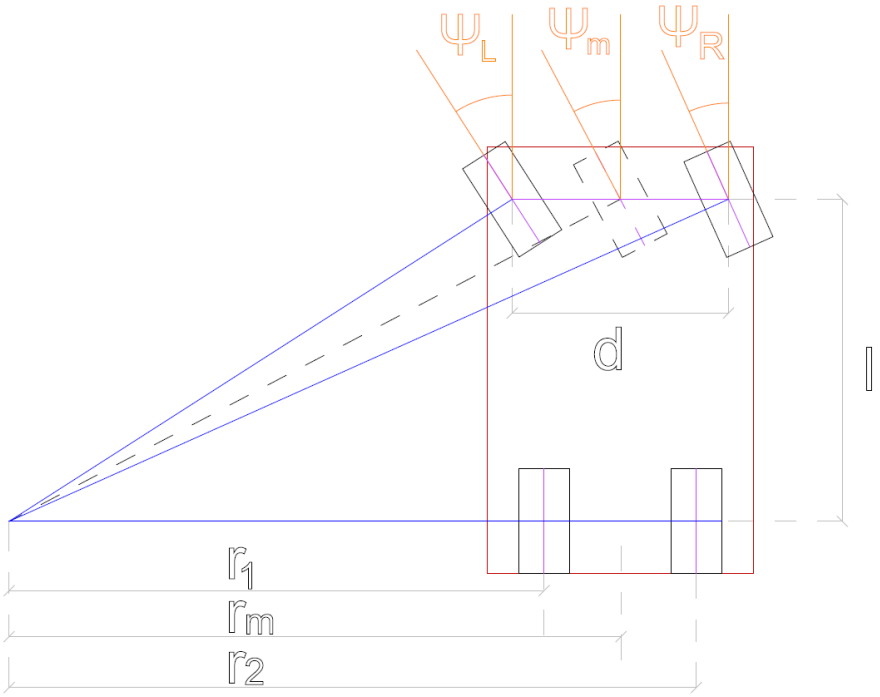

Figure 4 Car-like model

The controller has been implemented in the file "cmdvel2gazebo.py" in which a node has been created that subscribes to the topic 'cmd_vel' and allows the steering of the car by imposing linear and angular velocities.

The node publishes on the topics 'ferrari/front_left_steering_position_controller/command' , 'ferrari/front_right_steering_position_controller/command', 'ferrari/joint1_velocity_controller/command', 'ferrari/joint2_velocity_controller/command' which affect the implemented joints.

LaserScan

In order to be able to perform sensing of cars randomly spawned in parking spots, it was decided to use a laser sensor (*hokuyo laser*). This sensor has been placed on the left side of the car so that it is always facing the parking spots.

The laser has been set to have a reading range of $8m$ and a coverage of about $180°$. These settings allow the sensing of two or more cars at the same time in order to calculate the distance.

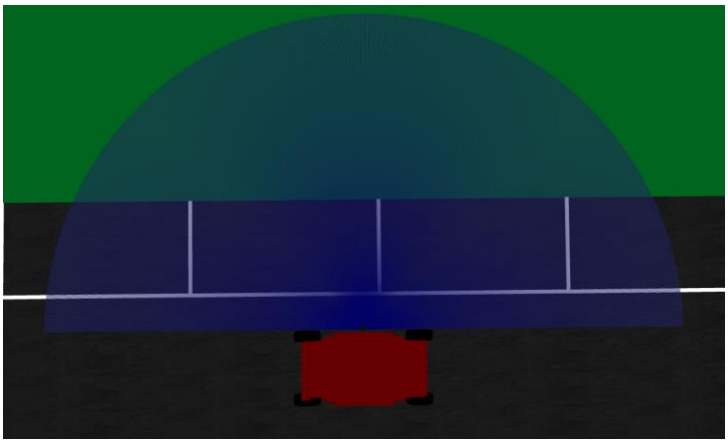

Figure 5 LaserScan

In order to implement it, the plugin "*libgazebo_ros_laser.so*" has been added in the *ferrari.xacro* file with the creation of the topic "*/scan*" on which the different information related to the laser and the readings of the distances from the vehicles are published.

The type of message used is "*sensor_msgs/LaserScan.msg*" " and, among the different data in it, we used the ranges field that contains the different distances read by the sensor.

LaserScan Behavior

As described above, the laser allows sensing in a radius of $8m$ and over a total angle of $180°$.

It was decided to adopt as a reference the counterclockwise rotation of the angular positions read by the sensor: the first reading on the right corresponds to $0°$, while the last on the left to $180°$.

By setting the laser sample number to 720 in the urdf file, it was possible to determine the step (angular distance) between the individual readings that compose the semicircle, that is equal to $0.25°$.

For each laser reading, through the use of the indices and step, the respective angles and distances to individual obstacles (if there are any) were determined.

Given this information, it is possible to determine the free angular distance and then, using trigonometry, it was possible to calculate the horizontal distance useful for parking.

Moreover, note that the msg ranges detect an infinite value (*inf*) in the case where there are no obstacles in correspondence of a given reading step.

Following are the different possible cases resulting from the random spawn of cars in parking spots.

Parking discrimination (dist_obj.py)

In general, in the cases analyzed below, the possible variation of the car orientation from the desired one was also considered.

Through the use of different arrays, the start and end distances of each parked car and their angles with respect to the zero laser reading were saved.

Note that in the figures below, for visual simplicity, the orientation of the car corresponds exactly to the orientation of the parking lot.

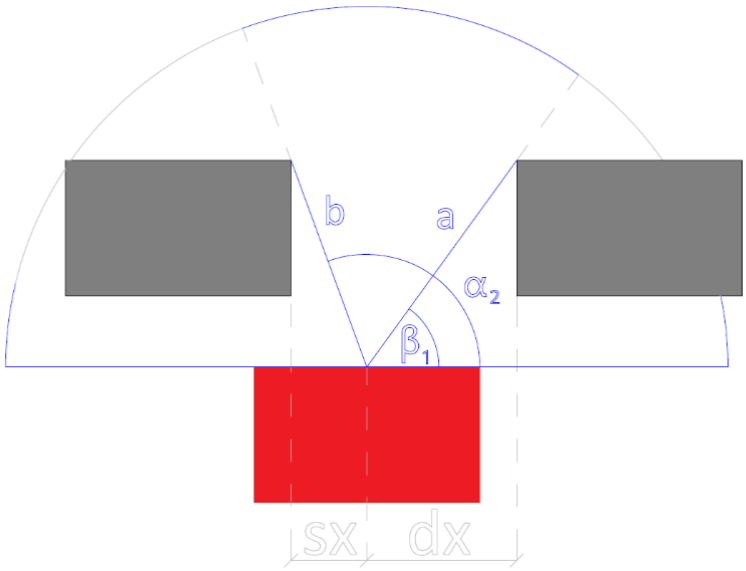

Figure 6 Parking between two cars (park_aff=1)

The logic used to calculate the free spot size is described below.

Having known the orientation of the car (θ_{odom}) and the desired orientation from the current side (parking orientation θ_park), the mismatch $\Delta\theta = \theta_{park} - \theta_{odom}$ can be determined so that the calculation can be generalized.

$$dx = a * \cos(\beta_1 - \Delta\theta)$$

$$sx = b * \cos(\pi - \alpha_2 + \Delta\theta)$$

Where *a* and *b* are the end and start distances of the cars of interest that delineate any free parking, while β_1 and α_2 identify the angles of the segments *a* and *b* with respect to the zero laser reading.

Having determined *dx* and *sx*, it was possible to determine the effective size of the useful distance between the two parked cars as follows, discriminating any negative values due to angles greater than $\frac{\pi}{2}$:

$$Distance = ||dx| + |sx||$$

If this distance is equal or superior to $5m$, the slot is free and the *find_park* flag is set to 1.

Below there are the different cases in which there is only one parked car that delineates the free parking.

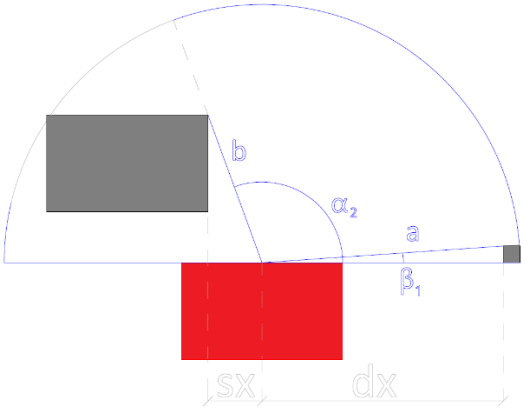

Figure 7 Parking with one car before (park_aff=0)

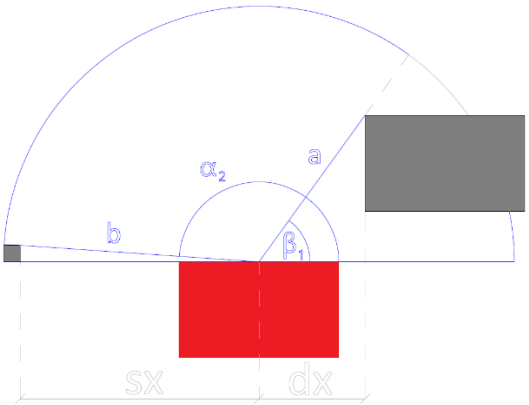

Figure 8 Parking with one car after (park_aff=2)

In order to be able to determine the useful size of the parking lot in the case where there is only one parked car, a "imaginary" car of small dimensions was inserted (box $8m$ away and with a width equal to a step of $0.25°$). In this way it was possible to use the same logic described above for parking between two cars and discriminate the presence of a parking spot of acceptable size.

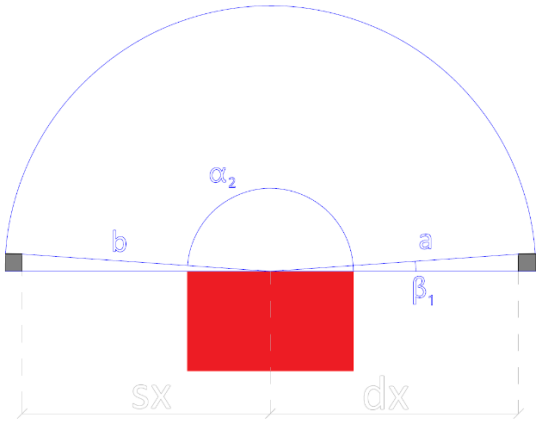

Figure 9 Parking without cars

In case the msg ranges is an array of only "*inf*", the script creates two "imaginary" cars on the laser reading extremes so that the logic described in the generic case can be applied again. Consequently the *find_park* flag will be set to 1 and the *park_aff* variable to 1 to detect the parking car spot.

The script under examination therefore returns a total of two variables:

- *find_park*: a flag that returns 1 in the case where there is a parking lot of acceptable size and 0 in the case where there are no free parking spots
- *park_aff*: a variable indicating the position of the free parking spot with respect to the current position of the car (0 if the free parking spot is after, 1 if it is in correspondence and 2 if it is before)

Parking extraction (goal_pos.py)

The script for the extraction of the parking spot is executed only if the *find_park* flag (output of the *dist_obj.py* script) is equal to 1.

This script (*goal_pos.py*) has two inputs: the variable that gives us the position of the free parking lot with respect to car (*park_aff*) and the dictionary (*parcheggi*) previously extracted from the parking yaml file. This dictionary contains the coordinates of the centers of all parking slots regardless of the random spawn of parked cars.

The extraction of the useful parking is based on the odometry of the car: when the free parking is detected by laser, the extraction of the center of the useful parking is carried out through two tolerances (on x and y) set differently according to the orientation of the car indicated by the parameter θ.

In general, we consider a tolerance of $4m$ in the direction of car advancement and a tolerance of $3m$ in the longitudinal direction. The setting of these tolerances has been optimized so that we can generalize the same script regardless of the car orientation (θ):

- for $\theta = 0$ or $\theta = \pi$ we chose a tolerance on x of $4m$ (*tolx*) and one on y of $3m$ (*toly*)
- for $\theta = \pi/2$ or $\theta = -\pi/2$ we chose a tolerance on x of $3m$ (*tolx*) and one on y of $4m$ (*toly*)

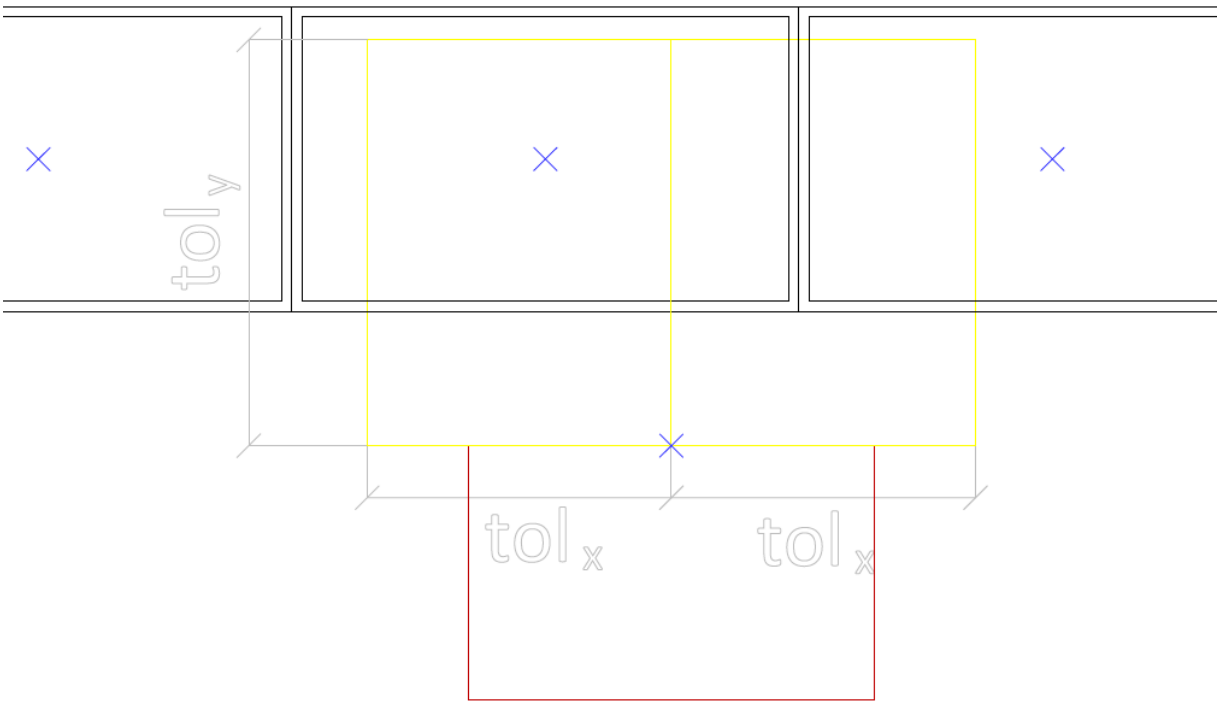

Figure 10 Tolerances for parking extraction

Consequently, scrolling through the dictionary of parking centers will create intervals on x and y centered in the odometry value of the car and extract the center of the parking spot contained within those intervals.

In the case where the center of the car is located in correspondence of the final line of a parking lot, the script could extract two parking spots that fall within the tolerances mentioned above. This "error" has been overcome with the introduction of the variable *park_aff* which allows us to discriminate the correct parking to extract.

Parallel Parking

After having extracted the center of the useful parking spot, the main script (*trajectory_tracking.py*) calls the *parallel_parking_trajectory* function in order to determine the starting point of the parking maneuver (point A in Figure 11) on which the car will position itself to perform it.

In order to move from the point in which the free slot is found to the point in which the parking maneuver begins, the *to_point* function is used, which makes the car move forward with a constant speed and a straight trajectory.

It should be noted that for the entire parking maneuver the center of the rear axle, which describes two arcs of circumference with different radii, was considered as the car's reference point.

Parking trajectory building

For the parking trajectory building, a single maneuver was planned in accordance with the maximum steering angles of the front wheels. These angles determine a physical limit of the maneuver as they could lead to collision with any parked cars.

It should be noted that the construction of this trajectory was done starting from the final point of parking (point *C* in Figure 11), determined by translating the center point of the parking spot extracted from *goal_pos.py*.

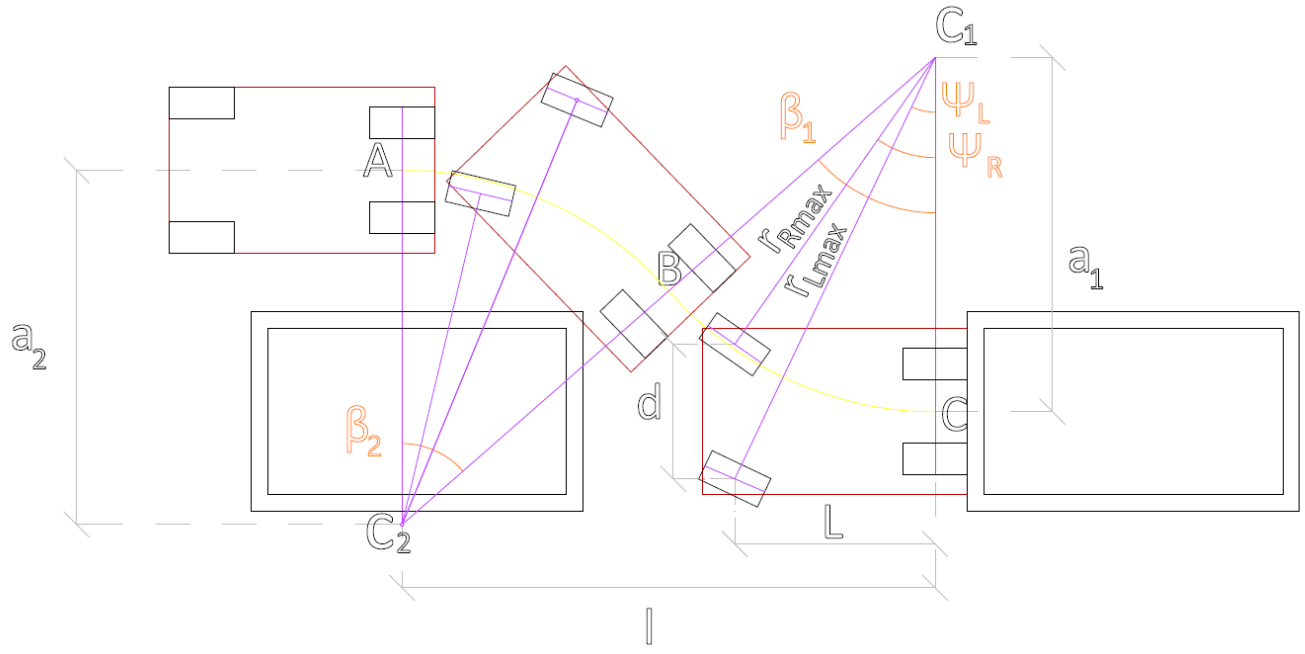

Figure 11 Parking maneuver

Known the constructive parameters and the limits of the car, through trigonometry, we calculated the radii and the salient angles of the trajectory, specifying that we assumed Ψ_R equal to the maximum steering angle.

$$r_{Rmax} = \frac{L}{\sin(\Psi_R)} \qquad r_{Lmax} = r_{Rmax} + d$$

$$\Psi_L = \operatorname{asin}\left(\frac{L}{r_{Lmax}}\right)$$

Then *r* was calculated by approximating the most external point of the car (edge of the car with possible impact).

$$r = r_{Lmax} + wheel\_radius$$

Calculation of radii of circumferences:

$$a_1 = r_{Rmax} * \cos(\Psi_R) + \frac{larghezza\ auto}{2} - \frac{battistrada}{2}$$

$$a_2 = \frac{l - a_1 * \sin(\beta_1)}{\sin(\beta_2)}$$

Using point *C* extracted from the previous script, points *B* and *A* were calculated:

$$B = [\ a_1 - a_1 * \cos(\beta_1) + Cx\ , Cy + a_1 * \sin(\beta_1)\ , 0\ ]$$

$$A = [\ Bx + a_2 - a_2 * \cos(\beta_2)\ \ , By + l - a_1 * \sin(\beta_1)\ , 0\ ]$$

Calculation of centers of circumferences:

$$c_2 = [Ax - a_2\ , Ay, Az]$$

$$c_1 = [Cx + a_1\ , Cy, Cz]$$

Known the centers of the circumferences and the radii, it was possible to determine the path in the operating space (curvilinear abscissa s_2 and s_1) by applying the parametric equation of the circumference (considering the appropriate rotations around the axes).

$$p_1 = c_2 + R_x(\pi) * \begin{bmatrix} a_2 * \cos(s_2/a_2) \\ a_2 * \sin(s_2/a_2) \\ 0 \end{bmatrix}$$

$$p_2 = c_1 + R_z(\pi - \beta_1) * \begin{bmatrix} a_1 * \cos(s_1/a_1) \\ a_1 * \sin(s_1/a_1) \\ 0 \end{bmatrix}$$

Where p_1 is the path joining points *C* and *B*, and p_2 is the path joining points *B* and *A*.

The velocity profile (*vel*) applied is trapezoidal with a maximum speed of $1.75\ m/s$ and the angular velocity was determined as follows:

$$\omega = \tan(\Psi) * \frac{vel}{L}$$

This definition is valid for both circumferences using the appropriate parameters.

Below is an example of trajectory generation ($\theta = -\frac{\pi}{2}$) with associated graphs of path, velocity, angular velocity, and orientation.

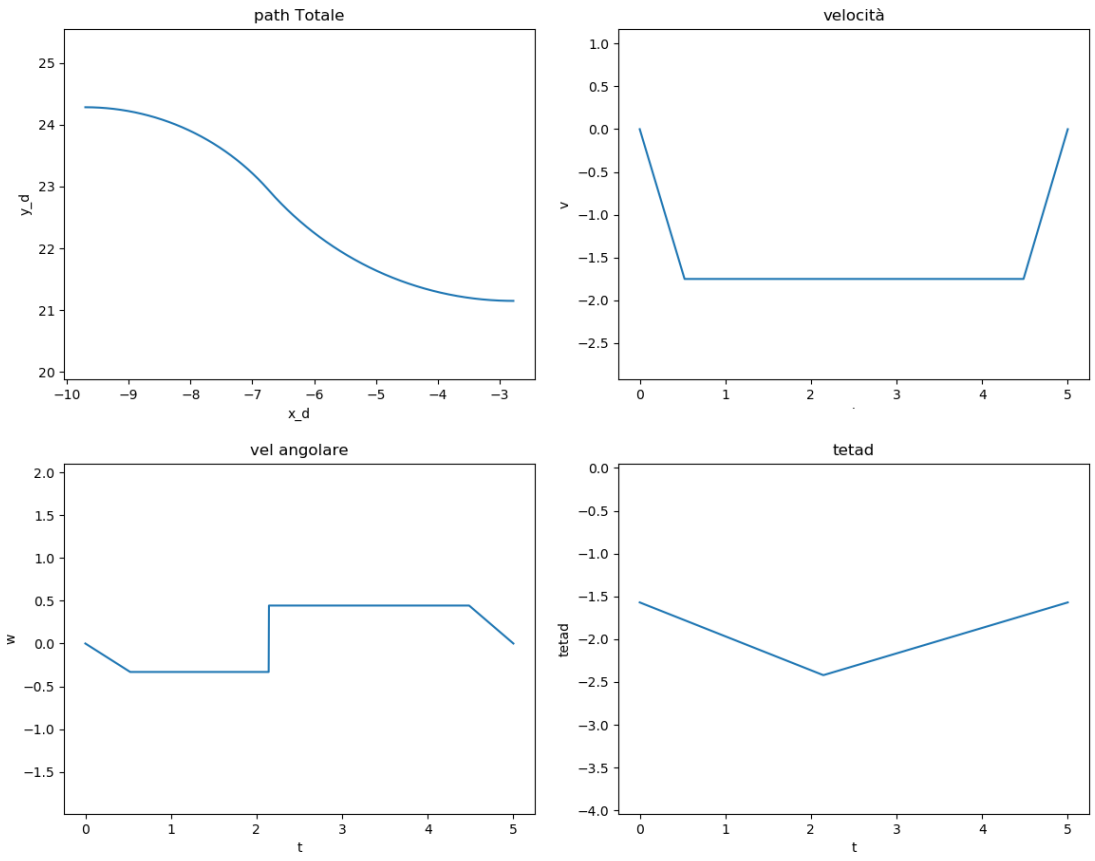

Figure 12 Trajectory generation

Hybrid Controller Synthesis

In order to drive the car and follow the desired path in the best way, it was decided to use a linear control.

The errors that allow the determination of the control inputs have been defined in the following way.

$$e_1 = (x_d - x_{odom}) * \cos(\theta_{odom}) + (y_d - y_{odom}) * \sin(\theta_{odom})$$

$$e_2 = -(x_d - x_{odom}) * \sin(\theta_{odom}) + (y_d - y_{odom}) * \cos(\theta_{odom})$$

$$e_3 = (\theta_d - \theta_{odom})$$

The controller used allows us to determine the control inputs through the following relationships:

$$u_1 = -k_1 * e_1$$

$$u_2 = -k_2 * e_2 - k_3 * e_3$$

Where:

$$k_1 = k_3 = 2 * \zeta * a$$

$$k_2 = \frac{a^2 - \omega_d^2}{v_d}$$

Performing a tuning of the parameters, the best results have been obtained using $\zeta = 0.9$ and $a = 1.45$.

Consequently the outputs of the controller are as follows:

$$v = v_d * \cos(e_3) - u_1$$

$$\omega = \omega_d - u_2$$

Note that since v_d is at the denominator in the calculation of k_2, for values close to zero it takes on very high values that negatively affect the control action producing undesirable effects on trajectory tracking. For this reason it was decided to disable the controller for some initial and final samples of the trajectory in such a way as not to diverge the determined control action.

Therefore, it can be said to have used a hybrid controller as the controller is used only between 3% and 87% of the total samples while the desired linear and angular velocities are used for the remainder.

Below are graphs for a simulation with a pure controller (left) and a hybrid controller (right).

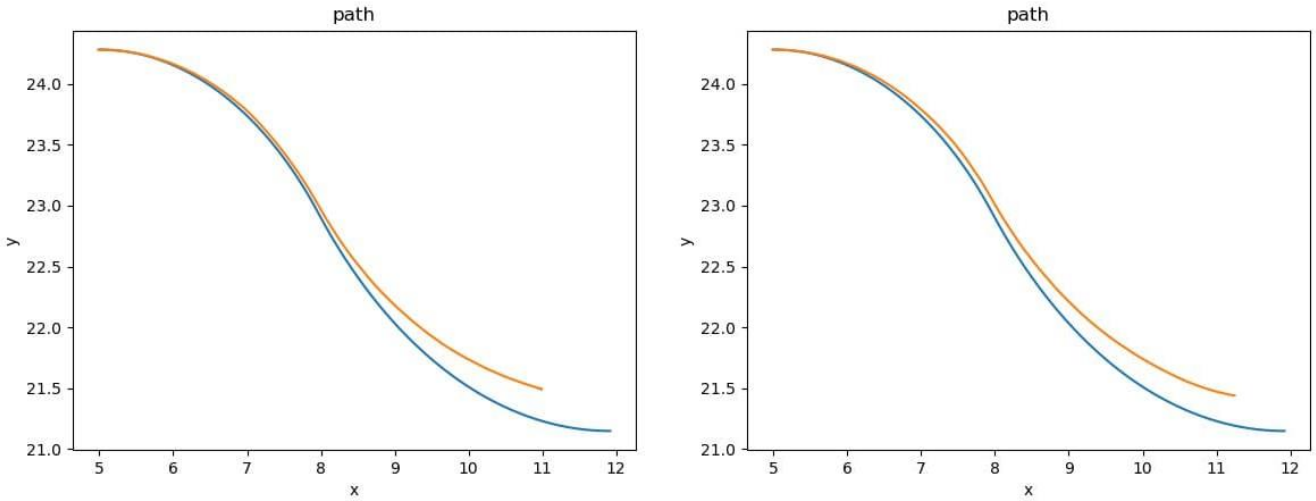

Figure 13 Path with pure – hybrid controller

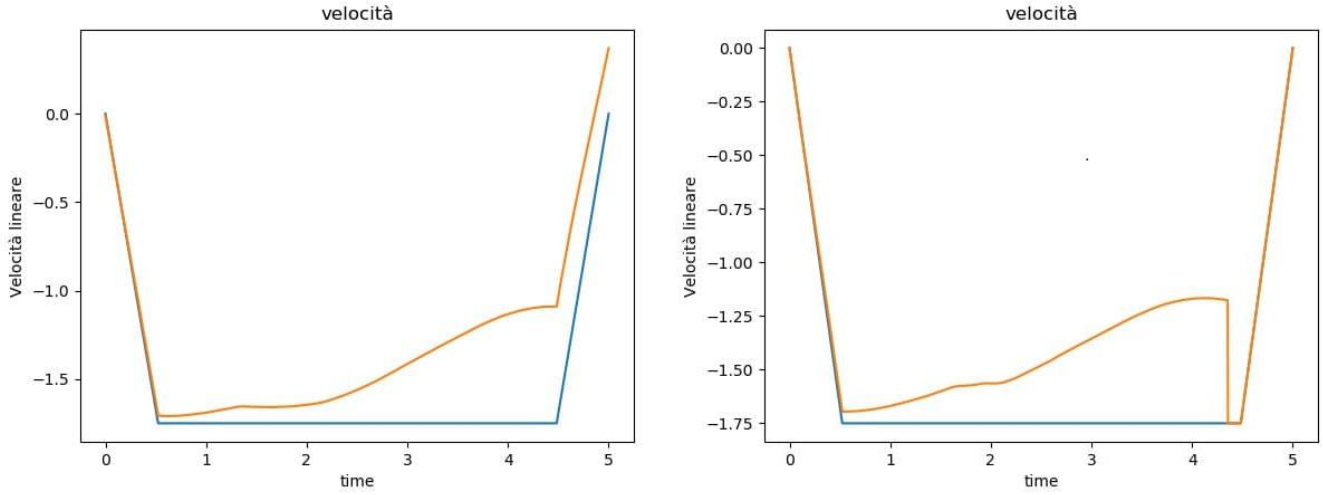

Figure 14 Velocity with pure – hybrid controller

As can be seen from Figure 14, in the simulation with pure controller, the speed reaches zero and reverses in sign. This results in a movement in the opposite direction to the one desired (parking) which causes an increase in errors and compromises the positioning of the car at the end of the maneuver.

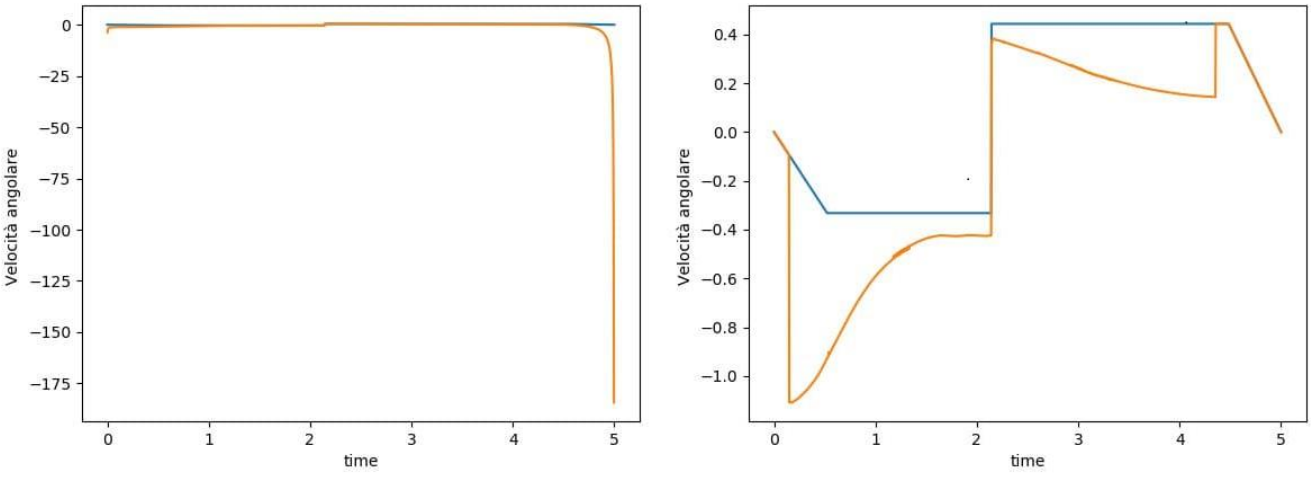

Figure 15 Angular velocity with pure – hybrid controller

As can be seen from Figure 15, in the simulation with a pure controller, the angular velocity presents a spike of considerable amplitude (greater than $175\ rad/s$) due to a desired velocity value close to zero and its sign inversion. As shown in the hybrid controller case, this occurrence can be avoided by disconnecting the controller at very small velocity values v_d.

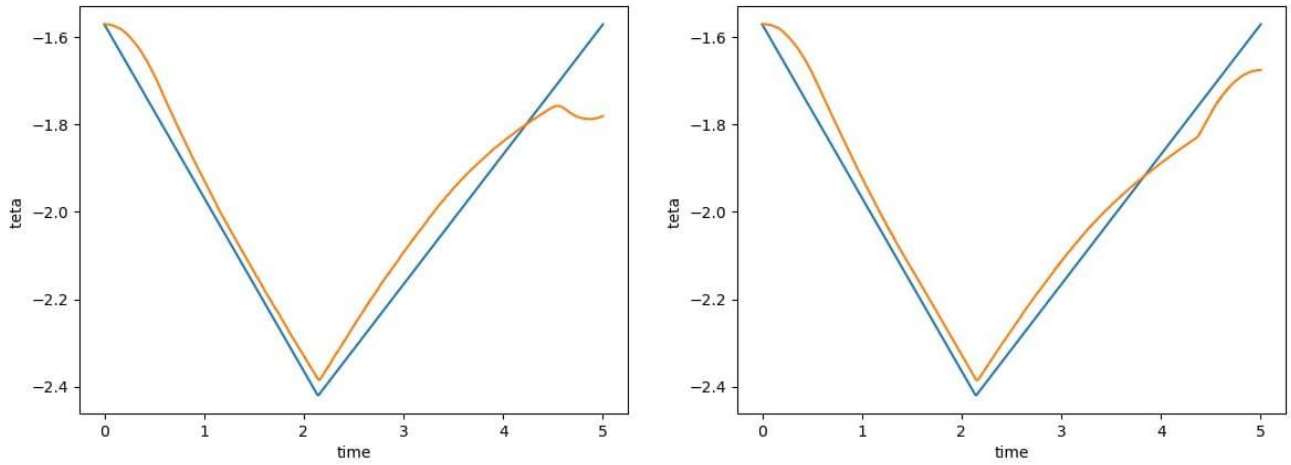

Figure 16 Theta with pure – hybrid controller

In conclusion, it can be seen that the final orientation of the car (Figure 16) is also very different in the pure controller case since in the final part of the trajectory the control action fails to track the desired trajectory. The use of linear and angular velocities in the final part leads to a significant improvement in trajectory tracking and thus to a more acceptable final positioning.

Simulation

The following simulation predicted the case of parking between two cars.

Initially the car moves in a straight line along the spawn side constantly detecting possible parking spots by means of the LaserScan. Once a free parking spot (equal to 5m) is found, the car stops and the initial point of the parking maneuver is determined (point A).

In the current simulation, parking spot was immediately detected as can be seen in Figure 17.

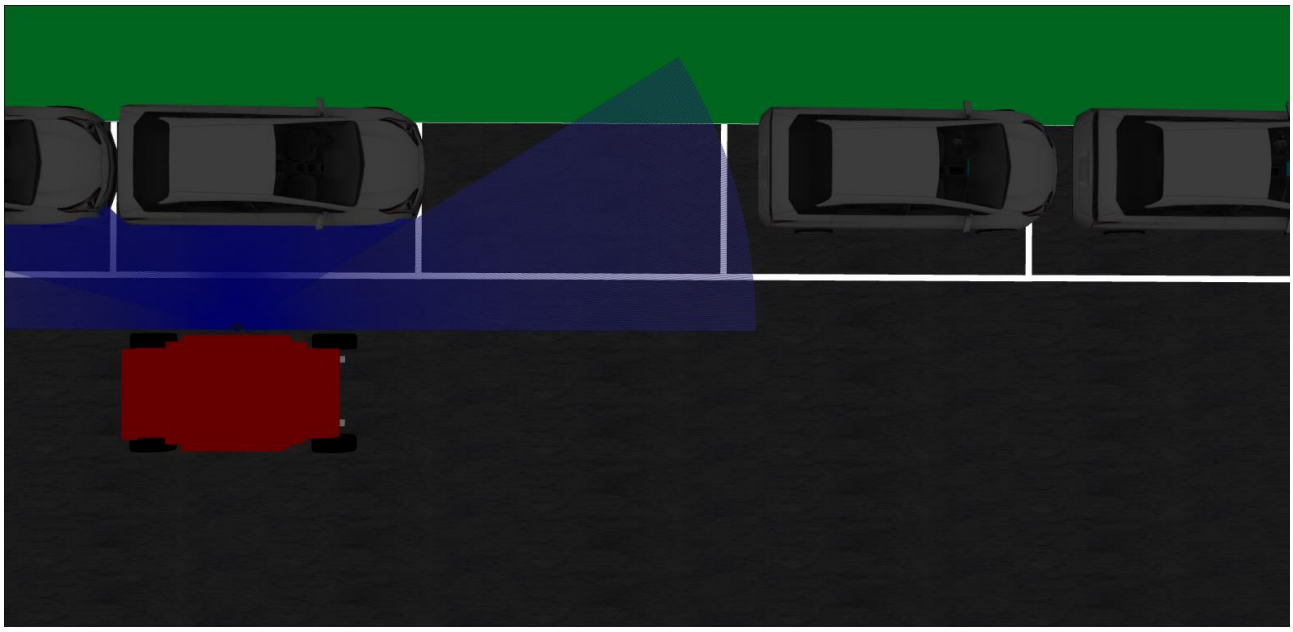

Figure 17 Free spot found

Then thanks to the *to_point* function the car moves to the previously determined point A (see Figure 18):

$$A = [0.099; 24.281]$$

Through odometry it was possible to notice a slight deviation from point A, in fact the point of interest on the rear axis of the car is located in coordinates:

$$A_{odom} = [0.145; 24.283]$$

This deviation will result in an error in the final parking position.

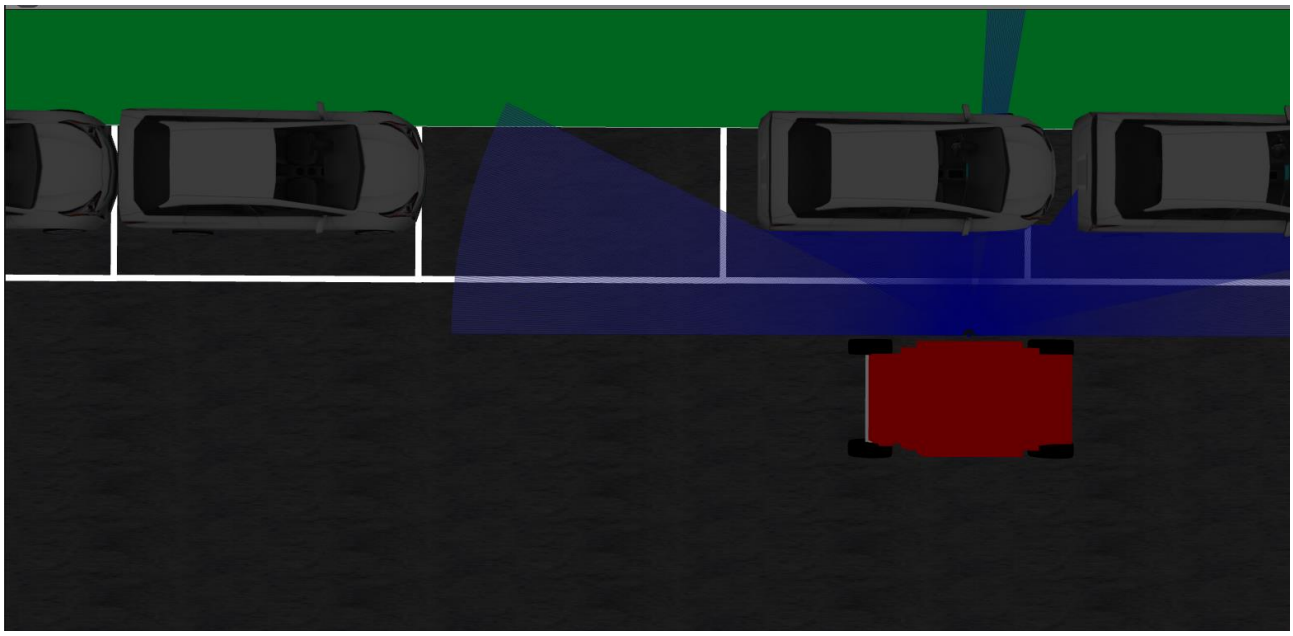

Figure 18 Parking maneuver

After reaching point A, the car starts the parking maneuver to bring the rear axle point to point C (see Figure 19).

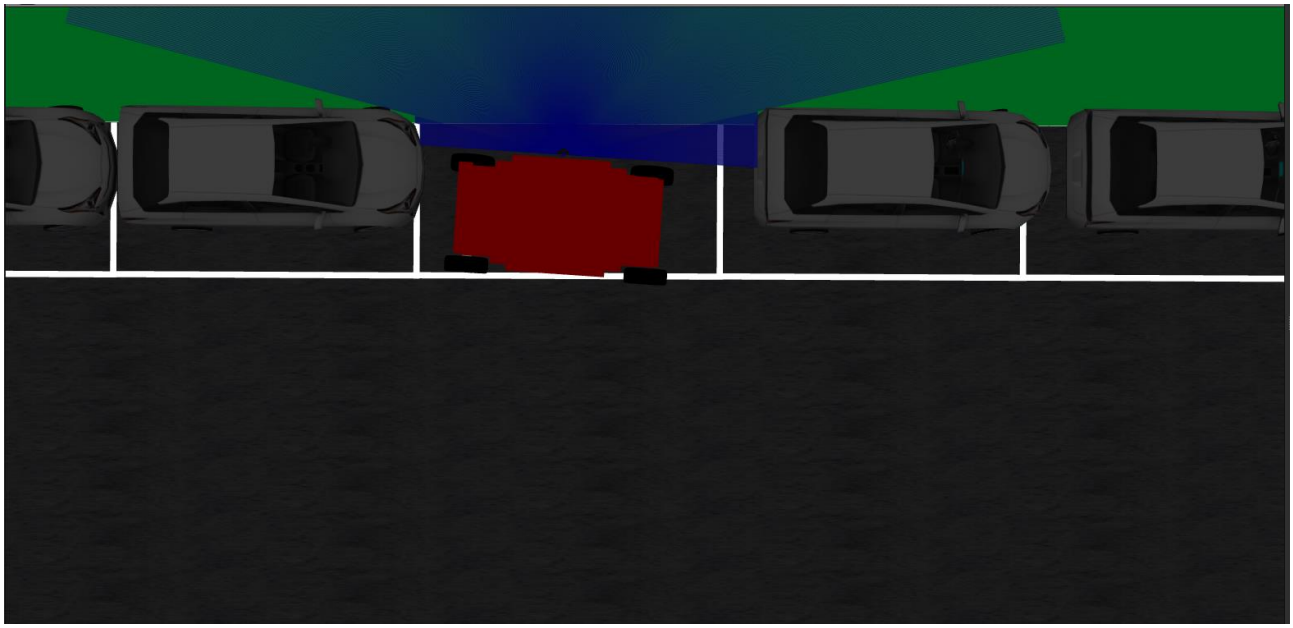

Figure 19 End of car parking

The above simulation is representative of all possible cases because, regardless of the number and positioning of parked cars, the machine always performs the same path.

Results Analysis

The final position of the machine turns out to be slightly different from the desired position, as can be easily seen from the errors above. This deviation is mainly due to two errors:

1. deviation of the rear point of the machine from the calculated point A
2. hybrid controller that fails to perfectly track the desired trajectory

Altogether the errors in the simulation carried out lead to a deviation from the desired values of:

- on x of $0.254m$
- on y of $0.654m$
- on θ of $0.077rad \cong 4.4°$

This difference can be appreciated in Figure 19.

The parking maneuver is never precise. Besides in the case in which the simulation had to be repeated with the same initial conditions and therefore the same trajectory, it turns out evident that the final position of parking (as it is evince from the errors) turns out to be always different.

Conclusions and future developments

In conclusion, in the world created in Gazebo a model of car-like robot (Ferrari) has been spawned to perform a parallel parking through sensing by LaserScan useful to discriminate free spots. The final goal has been achieved, however the project can be optimized in several ways for possible future developments.

In the first instance, it is possible to consider the dynamic model of the Ferrari in order to obtain more accurate and realistic analytical results and, subsequently, to obtain a better trajectory tracking.

In addition, it is possible to optimize the hybrid controller used for parallel parking by making it a pure controller that can guarantee exact trajectory tracking.

The initial project also included the implementation of a square trajectory in order to perform sensing on all possible sides of the square. Initially, we tried to apply I/O linearization. However, due to the length of the sides of the square and the large mass of the machine, the controller caused an exponential increase in linear velocities and therefore the derailment of the machine.

At the beginning we tried with adaptive k parameters and then disconnecting the controller in the curve sections. Both solutions did not give good results.

However, since the side was very long, the propagation of an angular error, even a small one at first, did not allow the correct tracking of the trajectory.

Given the considerable difficulties encountered in the pursuit of the square path, it was decided to abandon the resolution of the task and make the car stop at the end of the side where it is spawned.

The most critical issue is the difficulty of the controller to follow the velocity profiles due to the excessive mass of the car. Treating the problem with a kinematic model makes the control much more difficult. Therefore, in the future it is recommended to implement the control on a model that takes into account the dynamics of the vehicle.