SCUOLA POLITECNICA E DELLE SCIENZE DI BASE

Technical Project FSR/RL

Modellistica e controllo di un robot differenziale

Prof.
Jonhatan Cacace
Fabio Ruggiero

Candidato
Simone Contento
P38/24

Anno Accademico 2020/2021

Table of Contents

Abstract....3
Robot model....4
- Kinematic Model....4
- Turtlebot3 Waffle....5
- Robot description....6
- Low level control and odometry plugin....8
- Robot simulation....10

Scenario....11
Control....13
- Controller implementation....14
- Positioning routine implementation....19

Planning....21
- RRT implementation....23
- A* implementation....25
- Trajectory planning....28

Mission....29
- Phase 1 – mapping....29
- Phase 2 – Logistic task....30
- Logistic task implementation....31

Simulation and results....34
Final considerations and future works....37
References....38

Abstract

The following elaborate describe the step took for the implementation of a control system for a wheeled mobile robot used in a logistic environment.

This problem is faced with a bottom-up approach, starting from the low level with the control of the mobile wheeled robot and the planning of an optimal trajectory, to arrive at the high level mission, dealing with motion in environment with obstacle, mapping, localization and vision algorithms. In the next chapters this bottom up scheme will be followed to describe step by step the implementation choice and the results of each level.

First of all a brief introduction to mobile wheeled robot will bring us to the adoption of a specific mobile robot for our mission. The imported model will be the one used for all the simulations and tests.

Next we will provide a controller to our robot, to make it move from a starting position to goal following a desired hard coded trajectory. After that we will create a planner package, providing the controller the desired path to follow. The planner will be modified in the next steps to provide a trajectory that avoids obstacles. It will be important in these steps to design accurately the communication between each node.

At this point we should be able to move in the free space without particular problems. The following step is to provide a description of the environment and of its static obstacles. The models of the desired scene will be designed to be simulated together with the mobile robot.

Finally we will implement the high level mission, consisting of two phases. The first phase has the goal of mapping the environment and save the map. For this purpose we will use the SLAM technique together with a teleoperation package. In the second phase we will use the saved map and localization to solve a motion control problem with obstacles avoidance and make the robot move in the scene.

The last step will be to use the camera and the computer vision package to detect the AR Marker placed in the scene and retrieve information that will be useful for the mission. In particular in the first phase the recognized AR Marker will provide a localization information to set the rooms position, while in the second phase the ID red in the AR Marker will tell the mobile robot where to move.

All the mission will be implemented in ROS Noetic, with the help of Rviz for visualization and Gazebo for simulation. The whole algorithm will be implemented without using off-the-shelf library/command to solve it. The results are discussed in the chapter Simulation and results and illustrated with some video in the Github page of the project[7].

Robot model

The robot that will be used for the mission is the **differential-drive** robot, that is a particular type of mobile wheeled robot, very common in commercial application and useful for educational purpose.

This is because it’s kinematically equivalent to a unicycle robot, but will of course overcame its structural stability problem.

Kinematic Model

Lets derive the unicycle kinematic model and adapt it to the differential drive case. A unicycle is a vehicle with a single orientable wheel. Its configuration is completely described by the state $q=[x\ y\ \theta]^T$, where $(x\ y)$ are the Cartesian co-ordinates of the contact point of the wheel with the ground (or equivalently, of the wheel centre) and θ is the orientation of the wheel with respect to the x axis.[2]

The pure rolling constraint in the Pfaffian form $\begin{bmatrix}\sin(\theta) & -\cos(\theta) & 0\end{bmatrix}\begin{bmatrix}\dot{x}\\ \dot{y}\\ \dot{\theta}\end{bmatrix}=0$, entailing that the velocity is zero in the direction orthogonal to the sagittal axis, suggest to retrieve the kinematic model from the columns vectors of the basis of the null space of the matrix associated with the Pfaffian constraint, whose linear combination represents all the admissible generalized velocities.

So, indicating with *linear velocity* v the modulus (with sign) of the contact point velocity vector and with *steering velocity* ω the wheel angular speed around the vertical axis, we obtain the kinematic model:

$$\begin{bmatrix}\dot{x}\\ \dot{y}\\ \dot{\theta}\end{bmatrix}=\begin{bmatrix}\cos(\theta)\\ \sin(\theta)\\ 0\end{bmatrix}v+\begin{bmatrix}0\\ 0\\ 1\end{bmatrix}\omega \qquad (1)$$

In a **differential-drive** vehicle there are two fixed wheels with a common axis of rotation separately controlled, so that different values of angular velocity may be arbitrarily imposed.

The same kinematic model of the unicycle can be applied considering that in this case $(x\ y)$ will be the Cartesian coordinates of the midpoint of the segment joining the two wheel centres, and θ the common orientation of the fixed wheels (hence, of the vehicle body).

By varying the angular speed of the right and left wheel (ω_R, ω_L), we can vary the driving and steering velocities (v, ω) through these equations:

$$v=\frac{r(\omega_R+\omega_L)}{2}, \omega=\frac{r(\omega_R-\omega_L)}{d} \qquad (2)$$

where r is the wheel radius and d is the distance between the two wheels centers.

Turtlebot3 Waffle

The final model chosen for our mission is the **Turtlebot3** [1], because it's very detailed and tested model and it can be considered as a sort of "benchmark" for the differential drive simulations in ROS. Moreover it's open source and can be used and modified at will for our implementation.

The turtlebot3 project comes along with a variety of packages for mapping, motion control and many other functionalities to be used both for the physical robot and for the simulated one. For our purpose we will test the turtlebot3 only in simulation, and we have to write from scratch the packages for the desired functionalities, so we will delete all the folders from the project except the one with the URDF model, providing the robot description.

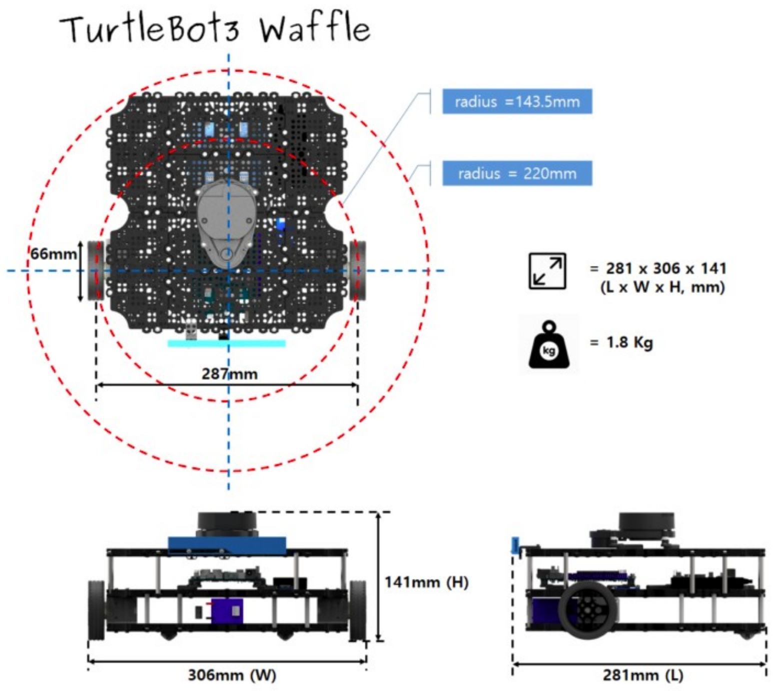
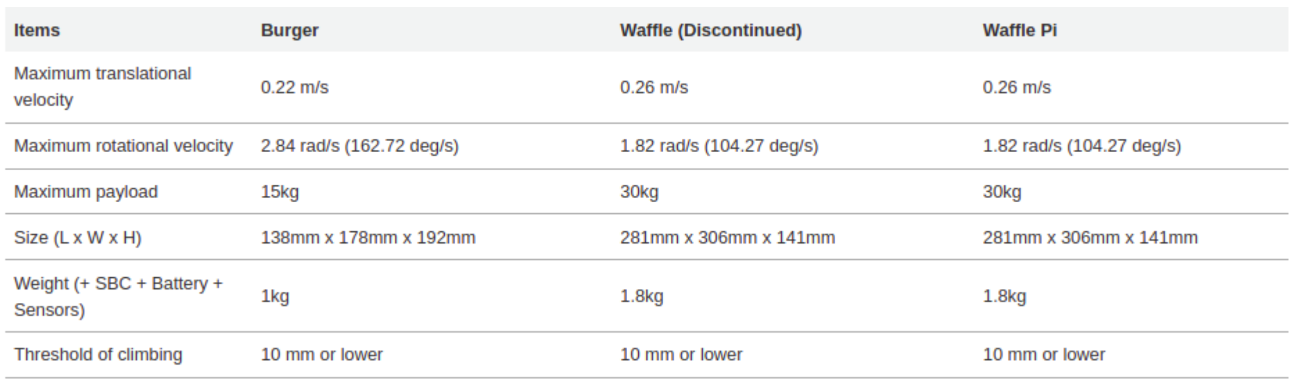


Figure 1: Turtlebot3 Waffle

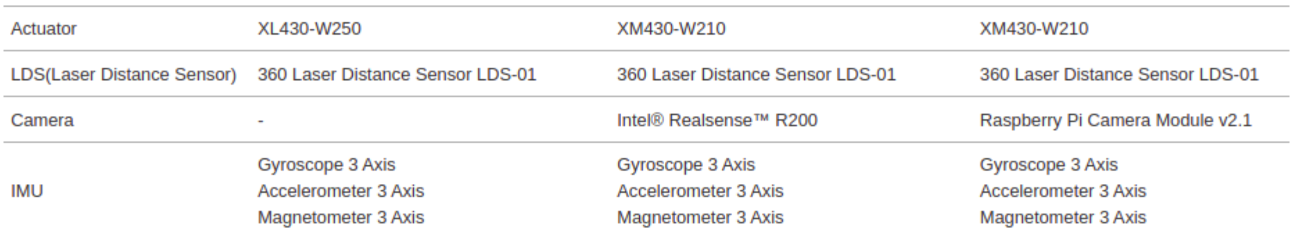
The turtlebot3 project comes in two different configuration, and we will use the **Waffle** configuration, more compact and already equipped with a 360 Laser, Imu and a RGB-Camera. Below is shown the table with the technical data:



Items	Burger	Waffle (Discontinued)	Waffle Pi
Maximum translational velocity	0.22 m/s	0.26 m/s	0.26 m/s
Maximum rotational velocity	2.84 rad/s (162.72 deg/s)	1.82 rad/s (104.27 deg/s)	1.82 rad/s (104.27 deg/s)
Maximum payload	15kg	30kg	30kg
Size (L x W x H)	138mm x 178mm x 192mm	281mm x 306mm x 141mm	281mm x 306mm x 141mm
Weight (+ SBC + Battery + Sensors)	1kg	1.8kg	1.8kg
Threshold of climbing	10 mm or lower	10 mm or lower	10 mm or lower

Figure 2: Turtlebot3 Technical table

And the implemented sensors:



Actuator	XL430-W250	XM430-W210	XM430-W210
LDS(Laser Distance Sensor)	360 Laser Distance Sensor LDS-01	360 Laser Distance Sensor LDS-01	360 Laser Distance Sensor LDS-01
Camera	-	Intel® Realsense™ R200	Raspberry Pi Camera Module v2.1
IMU	Gyroscope 3 Axis Accelerometer 3 Axis Magnetometer 3 Axis	Gyroscope 3 Axis Accelerometer 3 Axis Magnetometer 3 Axis	Gyroscope 3 Axis Accelerometer 3 Axis Magnetometer 3 Axis

Figure 3: Turtlebot3 Sensors

Of course, working only in simulation, the same sensor plugins could be added to both turtlebot3 model.

Robot description

As already said we will use the turtlebot3 Waffle model, that can be choose by setting the TURTLEBOT3_MODEL variable to waffle.

>>export TURTLEBOT3_MODEL=waffle

In order to avoid to do that every time we open a new terminal, it’s useful to set this variable in the .**bashrc** file.

The only package we are interested in is the ***turtlebot3/turtlebot3_description***, because it’s the one including the URDF describing the physical kinematic and dynamic parameters of the robot. As usual the URDF is derived from the **xacro** file, and so we will find different xacro files in which there are the macro definitions and the physical and visual implementation of all the links and joints of the robot.

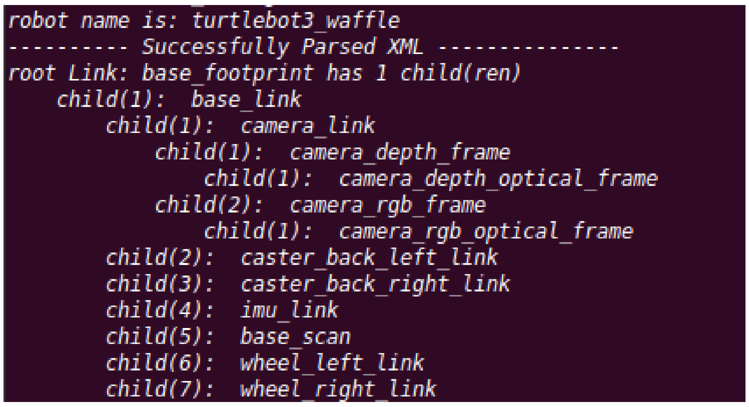
In these macro, in particular in the ***turtlebot3_waffle.gazebo.xacro*** files we can analyze the set up of the sensors plugins. Here we kept the default parameters that are properly tuned by the creators and the community, but it’s very useful to check the *frameName* and *topicName* because we will use them later.

We can check if the description is correct by generating the URDF file from the main xacro file

>>rosrun xacro xacro turtlebot3_waffle.urdf.xacro>generated.urdf

and using the following command:

>>check_urdf generated.urdf



```
robot name is: turtlebot3_waffle
---------- Successfully Parsed XML ---------------
root Link: base_footprint has 1 child(ren)
    child(1):  base_link
        child(1):  camera_link
            child(1):  camera_depth_frame
                child(1):  camera_depth_optical_frame
            child(2):  camera_rgb_frame
                child(1):  camera_rgb_optical_frame
        child(2):  caster_back_left_link
        child(3):  caster_back_right_link
        child(4):  imu_link
        child(5):  base_scan
        child(6):  wheel_left_link
        child(7):  wheel_right_link
```

Figure 4: check_urdf result

or generating the graph in pdf format with:

>>urdf_to_graphiz generated.urdf

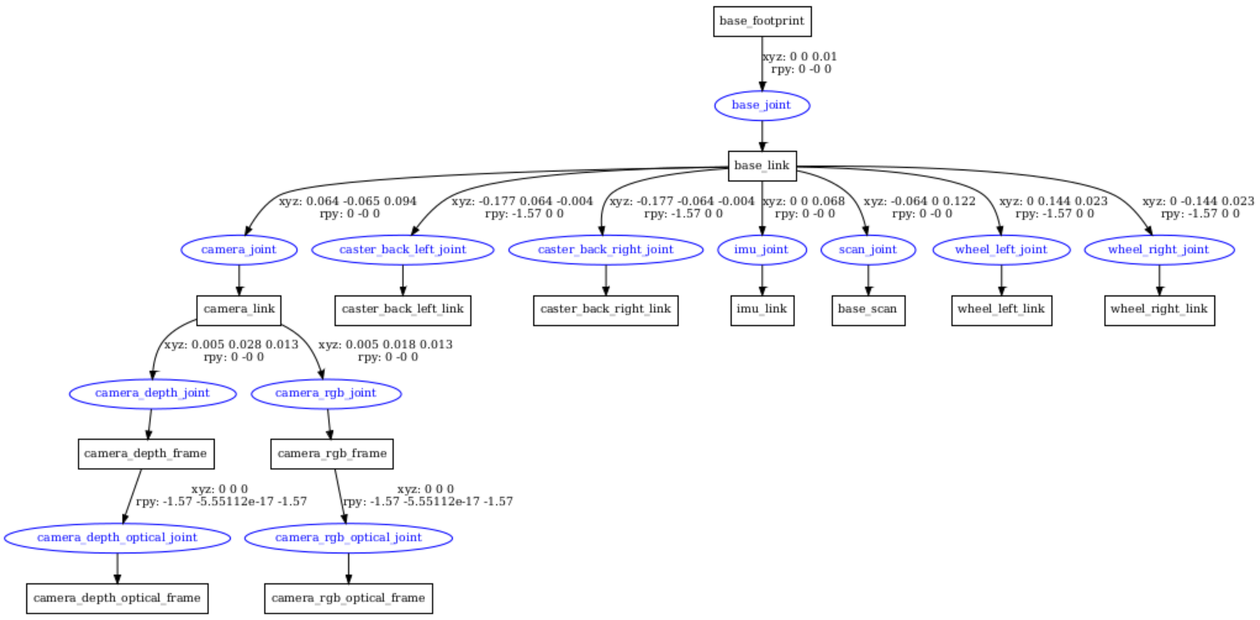
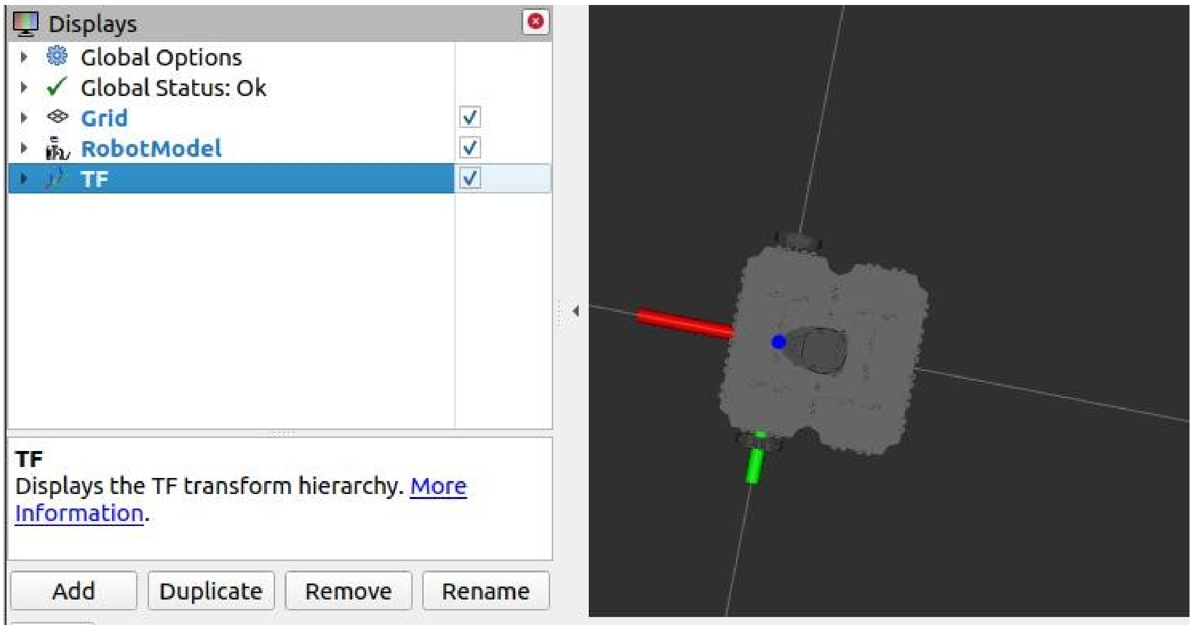


Figure 5: URDF graph PDF

Finally we can visualize the turtlebot3 model in Rviz creating a proper launch file and running it with:

>>roslaunch turtlebot3_description viz.launch



TF
Displays the TF transform hierarchy. More Information.

Figure 6: Turtlebot3 RViz visualization

Low level control and odometry plugin

Since it was required by the assignment, we have substituted the controller plugin (*libgazebo_ros_diff_drive*) that was included in the xacro file, with a new one designed from scratch (*libcontroller_plug*), with source code located in the ***turtlebot3_description*** *package* in the *src/plugin* folder.

Our plugin was written to be used as similarly as possible to the previous one, allowing to test all the next steps on both, and to do so it has to accomplish three task: publish odometry, publish odometry TF, receive heading and angular velocity, and command the single wheels.

The plugin is implemented as a **ModelPlugin** because it has to access to the physical properties of the robot model, is inserted in the URDF inside the <*robot*> element, and can pass information to Gazebo with the parameters specified in the <*gazebo*> tag.

The main elements are described below:

- **Load():** function called when the plugin is loaded the first time. Here all the variables (topic names, wheel radius, ecc...) are initialized to the value passed from the plugin parameters or to a default value.

- OnUpdate(): function called at each world update event. It will be used as a loop function to recall periodic functions. The actual frequency of each function will be conditioned to a *current_time* variable, so that every function will be called only if time T passed.

- cmd_cb: callback function called when a new command message is published by the high level controller on the command topic (/cmd_vel by default). *Receive a geometry_msgs/Twist* message with the heading and angular velocities and updates the single wheels velocity according to the kinematic model equation 2.

- **imu_cb:** callback function called when a new Imu measurement is published on /*imu topic.* When the parameter <useImu> is set to 1 uses the Imu measurement to adjust odometry computation.

- **_odom_pub:** publisher of the odometry on the odometry topic (*/odom* by default).

The odometry computation is a very crucial step for the assignment and for a good result of the task.

Considering v and ω constant in each sampling interval dt , and assuming to know the robot configuration q_k and input velocities (v_k, ω_k) at istant t_k , then the configuration $q_{(k+1)}$ at time $t_{(k+1)}$ can be reconstructed by integration of the kinematic model 1.

There are multiple formulation providing different level of approximation, but exploiting the chained form we can use the following (from [2]):

$$\begin{aligned} x_{(k+1)} &= x_k + \frac{v_k}{\omega_k}(\sin(\theta_{(k+1)}) - \sin(\theta_k)) \\ y_{(k+1)} &= y_k - \frac{v_k}{\omega_k}(\cos(\theta_{(k+1)}) - \cos(\theta_k)) \\ \theta_{(k+1)} &= \theta_k + \omega_k T_s \end{aligned} \tag{3}$$

It is necessary to handle the case $\omega_k = 0$ with a conditional instruction, that will compute the odometry with Euler and Rung-Kutta methods:

$$\begin{aligned} x_{(k+1)} &= x_k + v_k T_s \cos(\theta_k) \\ y_{(k+1)} &= y_k + v_k T_s \sin(\theta_k) \\ \theta_{(k+1)} &= \theta_k + \omega_k T_s \end{aligned} \tag{4}$$

which are exact and equal for line segments.

The values of v_k, ω_k can be reconstructed using the robot incremental encoders, and easily retrieved from the rotation of the left and right wheels thorugh the 2.

The forward integration of the kinematic model using the velocity com-mands reconstructed via the proprioceptive sensors is referred to as odometric localization or **dead reckoning.** As we will see better in the following, odometric localization is subject in practice to an error that grows over time (drift) and quickly becomes significant over sufficiently long paths. This error is the result of several causes, that include wheel slippage, inaccuracy in the calibration of kinematic parameters (e.g., the radius of the wheels), as well as the numerical error introduced by the integration method, if Euler or Runge–Kutta methods are used[2].

Robot simulation

Another crucial package is **turtlebot3_simulation/turtlebot3_gazebo**. In this directory we will find all the models and the worlds file that will be loaded in the simulated scene, together with the launch files to run the simulations. The turtlebot3 package comes along with already implemented launch files, models and worlds files to simply and quickly run some default simulations.

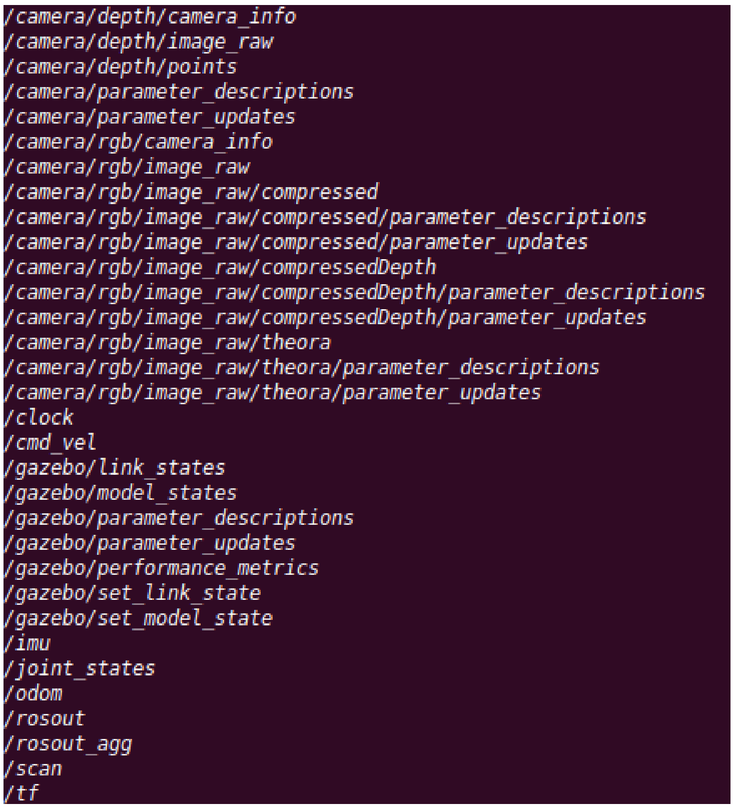
For our mission we're going to write personalized worlds file and to include different models, but, following the bottom up approach, we can start simulating the robot and developing the controller in the empty scene already provided. In this way we can simulate the turtlebot3 in an empty scene just using the command:

>>roslaunch turtlebot3_gazebo turtlebot3_empty_world.launch

Thanks to the mentioned plugins for the sensors we will see a set of messages published by the sensors on the topic specified in the configuration. Moreover our *libcontroller_plug* plugin will convert the Twist messages (*geometry_msgs/Twist*) published on the /*cmd_vel* topic *(*specified in *<commandTopic>)* into desired velocity for the wheel to make the robot move. The same plugin will also publish the calculated odometry data on the */odom* topic *(*specified in *<odometryTopic>)* in the structure of a *nav_msgs/Odometry* message.

We can always check the active topics with the command:

>>*rostopic list*



```
/camera/depth/camera_info
/camera/depth/image_raw
/camera/depth/points
/camera/parameter_descriptions
/camera/parameter_updates
/camera/rgb/camera_info
/camera/rgb/image_raw
/camera/rgb/image_raw/compressed
/camera/rgb/image_raw/compressed/parameter_descriptions
/camera/rgb/image_raw/compressed/parameter_updates
/camera/rgb/image_raw/compressedDepth
/camera/rgb/image_raw/compressedDepth/parameter_descriptions
/camera/rgb/image_raw/compressedDepth/parameter_updates
/camera/rgb/image_raw/theora
/camera/rgb/image_raw/theora/parameter_descriptions
/camera/rgb/image_raw/theora/parameter_updates
/clock
/cmd_vel
/gazebo/link_states
/gazebo/model_states
/gazebo/parameter_descriptions
/gazebo/parameter_updates
/gazebo/performance_metrics
/gazebo/set_link_state
/gazebo/set_model_state
/imu
/joint_states
/odom
/rosout
/rosout_agg
/scan
/tf
```

Figure 7: rostopic list result

Scenario

Also if we will start working with the empty world environment it’s useful to design the scene for the final mission in this early phase. The environment is composed by 3 rooms: a warehouse and two destination rooms.

To change the scene all we need is to modify the launch file to make it open our custom world file, in which it will be described the scene including all the models we need. The models can be easily located in the *models* folder because its path is added in the GAZEBO_MODEL_PATH variable in the package.xml file.

The models that we need to add are just the AR Markers that the mobile robot has to read. We can download the models of the Aruco Markers on internet and add the desired ones in the models folder. In our case we added the folders *aruco_visual_marker_x* with x representing the Id number of the marker and going from 0 to 7 (not all of them will be used). In each folder there are all the necessary files, like the *model.config* and *model.sdf* that describe the object and the visualization and the *materials* folder containing the textures.

The world file is created with the **Gazebo Building Editor**, that can be opened with the shortcut Ctrl+B. In this environment it’s easy to create the desired scene, composed by rooms, simply building the walls. When we have finished placing the walls it’s possible to add textures and colors to them and save the world file in the worlds folder. Then we can exit the building editor and insert with the Insert button the models of the AR Markers in the scene and place them in the desired position (on the walls) and with the desired orientation, that is important for the recognition step. Finally we can save again the complete world file and modify the launch file to make it open our custom world. Different world were designed, with different shapes of the wall and dimension of the rooms, and each one is opened with a different launch file.

The final version of the world is the *small_wh.world* that can be simulated with the *small_wh.launch*.

```
>>roslaunch turtlebot3_gazebo small_wh.launch
```

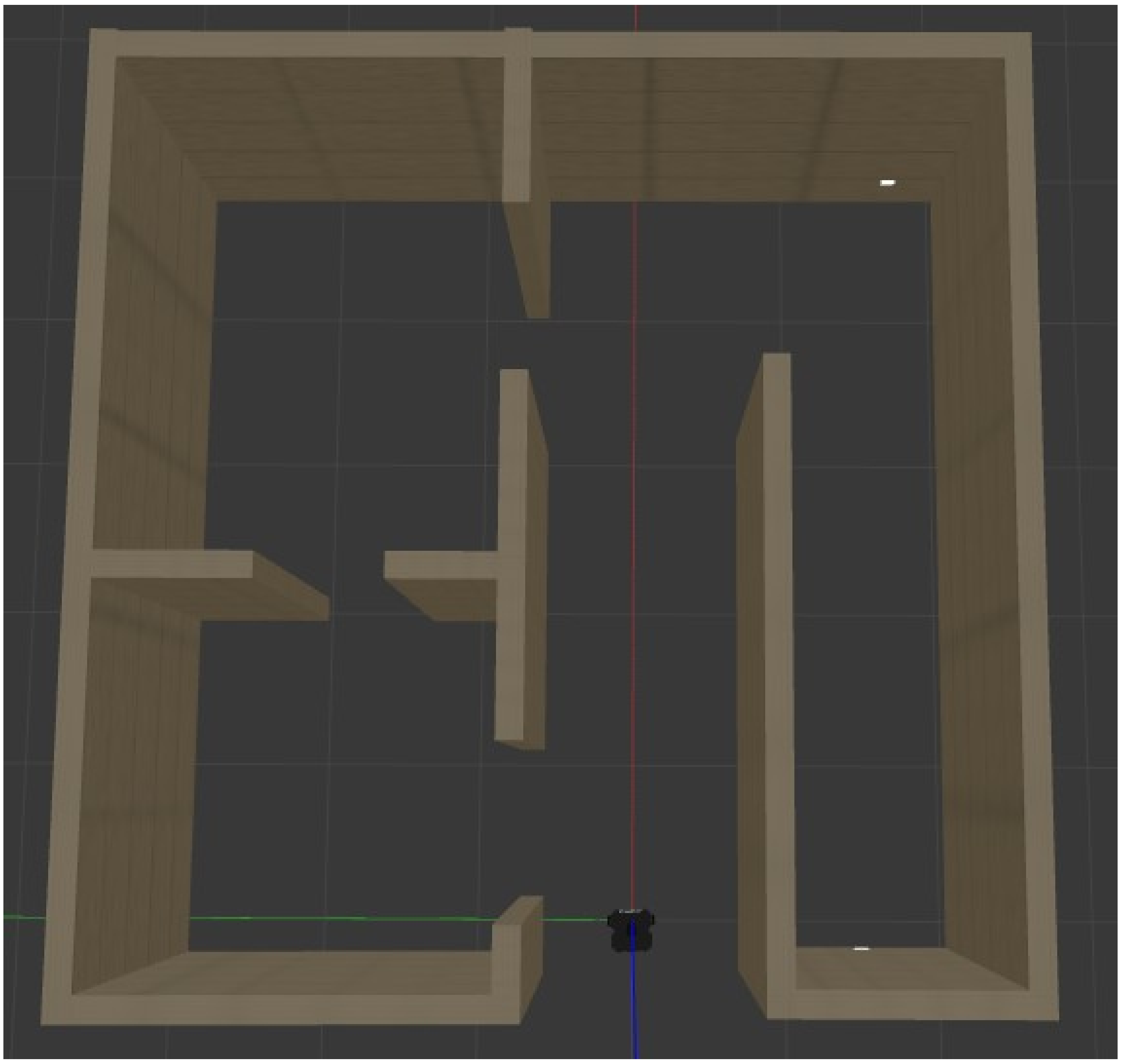


Figure 8:Final scene

Control

Now that we have our robot model and we are able to spawn it in a simulated scene, empty or not, we want to control it to make it move in the environment. As already specified, through the */cmd_vel* topic we can publish the Twist message to make the robot navigate. So the specified control input will be the heading velocity v and the steering velocity ω.

This is not a problem since the motion control problem for wheeled mobile robots is generally formulated with reference to the kinematic model, by assuming that the control inputs determine directly the generalized velocities $\dot{q}$. For example, in the case of the unicycle (and equivalently of the differential drive robot) this means that the inputs are the driving and steering velocity inputs v and ω. The angular velocity of each wheel ω_L and ω_R can be easily retrieved from the equations 2.

Looking at the final mission, in which the robot has to operate in a workspace containing obstacles, it is desirable to adopt a **trajectory tracking controller**, forcing the robot to track a desired trajectory planned in advance in such a way to keep the robot at a security distance from the obstacles and also to be admissible for the kinematic model.

For our mobile robot we choose to design a controller based on **input/output feedback linearization**. The basic idea is to use two auxiliary outputs representing the coordinates of a point B located along the sagittal axis of the vehicle at a distance $|b|$ from the contact point of the wheel with the ground, indicating the position of the robot.

$$\begin{aligned} &y_1 = x + b\cos(\theta) \\ &y_2 = y + b\sin(\theta) \\ &b \neq 0 \end{aligned}$$

The choice of the controller was guided by the possibility to track non persistent trajectory, that is not possible for approximated linearization and nonlinear controller. Moreover the main drawbacks of the input/output feedback linearization controller will be overcome by an additional ROS node providing a routine to adjust the final pose of the differential drive robot in front of the AR Marker. In this way it will not be a problem that the final orientation is not controlled and that the control scheme will drive the point B to the destination rather than the middle point of the axis of the unicycle. Finally, since the reference is assigned for the point B, the planned desired trajectory can exhibit isolated points with discontinuous geometric tangent (like in a broken line) without requiring the robot to stop and reorient itself at those points. As long as b is not null, B will follow the trajectory and the resulting trajectory of the robot will be smoother as b is increased and more accurate as b is decreased.

The control inputs can be computed from the relation between the derivatives of the auxiliary outputs and the robot velocities:

$$\begin{bmatrix} \dot{y}_1 \\ \dot{y}_2 \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -bsin(\theta) \\ \sin(\theta) & bcos(\theta) \end{bmatrix} \begin{bmatrix} v \\ \omega \end{bmatrix} = \boldsymbol{T}(\theta) \begin{bmatrix} v \\ \omega \end{bmatrix} \tag{5}$$

that can be inverted for $b \neq 0$

$$\begin{bmatrix} v \\ \omega \end{bmatrix} = \boldsymbol{T}^{-1}(\theta) \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = \begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\frac{\theta}{b}) & \cos(\frac{\theta}{b}) \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} \tag{6}$$

Obtaining the linearized closed loop relationship:

$$\begin{aligned} \dot{y}_1 &= u_1 \\ \dot{y}_2 &= u_2 \\ \dot{\theta} &= \frac{u_2 \cos(\theta) - u_1 \sin(\theta)}{b} \end{aligned} \tag{7}$$

At this point a simple linear controller is sufficient for u_1 and u_2

$$\begin{aligned} u_1 &= \dot{y_{1d}} + k_1 (y_{1d} - y_1) \\ u_2 &= \dot{y_{2d}} + k_2 (y_{2d} - y_2) \\ k_1 &> 0, k_2 > 0 \end{aligned} \tag{8}$$

This control guarantees exponential convergence to zero of the position tracking errors $y_{1d} - y_1, y_{2d} - y_2$ with decoupled dynamics on its components.

Controller implementation

The controller, together with the routine to adjust the final pose in front of the AR Marker, is implemented in the **mobile_navigation** package. In the includes folder there are the libraries defining the required classes while in the src folder there are the source codes that implement the methods. Let focus on the controller first.

The controller is implemented as a class named **NAVIGATION** that will manage all the needed data members and member functions. The main functionalities implemented are the publishers and subscribers to exchange information with the rest of the system and the control loop, that will iterate calculating the desired velocity on the basis of the received data.

The main elements are described below:

- **odometry_cb:** callback function called when a new odometry measurement is published on /*odom topic. Receive a nav_msgs/Odometry* message and updates the current position, orientation and velocities of the point B.

Note: In this first step the localization is based on the odometry computed from the measurements coming from the encoders of the wheels (dead reckoning). In the final configuration, as we will see later, the localization will be provided from AMCL, that will adjust the computed odometry on the basis of the laser measurements. So the odometry_cb will be linked to the /*amcl_pose topic, receiving a geometry_msgs/PoseWithCovarianceStamped* message.

- plan_cb: callback function called when a new path is published on /*path topic by the planner. Receive a nav_msgs/Path* message and fill a vector of desired position (_wp_list) and one of desired velocities (_wp_vel_list).

Note: In first instance, since we have no planner yet, the planner-controller chain will be replaced by publishing a path request on the topic /*fake_path to which the controller is subscribed (just a std_msgs/Int32* to call the function) and, in its callback (fake_path_cb), publish an hard coded path on the */path* topic.

- **_cmd_vel_pub :** publish the control output velocity to the */cmd_vel* topic to move the robot.

- _result_pub : publish the result of the planning and control phases to give a feedback on the correct execution of the task. If 0 is published the goal is reached, otherwise some error occurred in the control loop.

- ctrl_loop : it's the controller loop, that runs at 100 Hz. It waits for the first odometry measure and for the path to be received by the planner, then take the trajectory and gives it as input to the *navigation* function (described next). If the path cannot be generated publish a result of 1 (negative result), otherwise if it affords to finish in a desired time, it publishes 0 (positive result). It is started as a thread at the start of the node, to work in parallel to the other threads and callbacks.

- navigation : the navigation function take as input the desired trajectory and publishes on the /*cmd_vel* topic the desired velocity to move the robot. It implements the logic of the input/output feedback linearization controller and returns true when the position error is less then a threshold. When the control loop receive the true value returned by the navigation, provide it the new position and velocity reference.

The controller parameters such as the gains and the distance b are retrieved from the **controller_param.yaml** file located in the params folder so that they can be changed without recompiling all the source code.

We can test this first step by launching the turtlebot3 model in the empty world

>>*roslaunch turtlebot3_gazebo turtlebot3_empty_world.launch*

running the mobile_navigation node of the mobile_navigation packages

>>*rosrun mobile_navigation mobile_navigation*

and finally publishing an int data on the */fake_path* topic

>>*rostopic pub /fake_path std_msgs/Int32 "data: 0"*

At this point we should see the turtlebot3 follow the hard coded trajectory (square) described by the points given in the fake_path_cb function.

In Figure 9 we can see the desired trajectory in red and the actual trajectory of the robot in black, with

$$k_1 = 0.1$$
$$k_2 = 0.1$$
$$b = 0.3$$

In Figure 10 are plotted the x (black) and y (red) errors and the linear (black) and angular (blue) velocities.

We can compare the results of this simulation with the results obtained choosing $b = 0.1$ and see, as we expected, that lowering b we obtain that the trajectory is followed more strictly and the velocity peaks are higher, while with lower b we have a smoother trajectory for the robot.

Note: in this first phase the only information provided to navigate comes from the odometry computation (*dead reckoning*) so the real robot position could be affected by drift error caused by slippage an unmodelled dynamics.

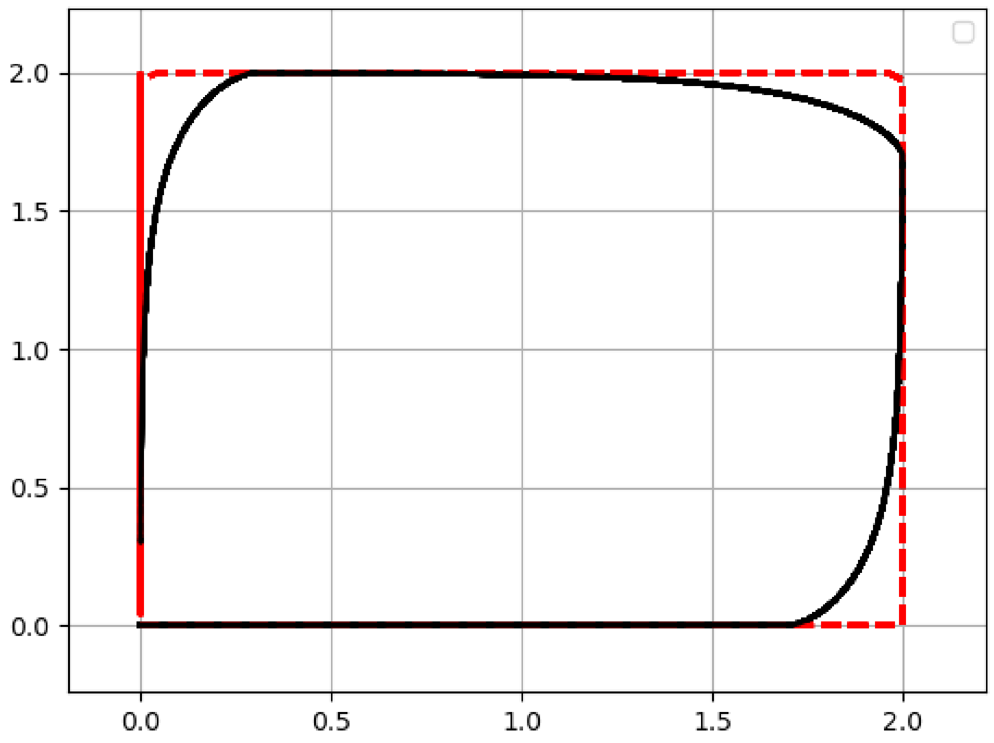


Figure 9: Square desired trajectory (red) and robot position (black) with b = 0.3

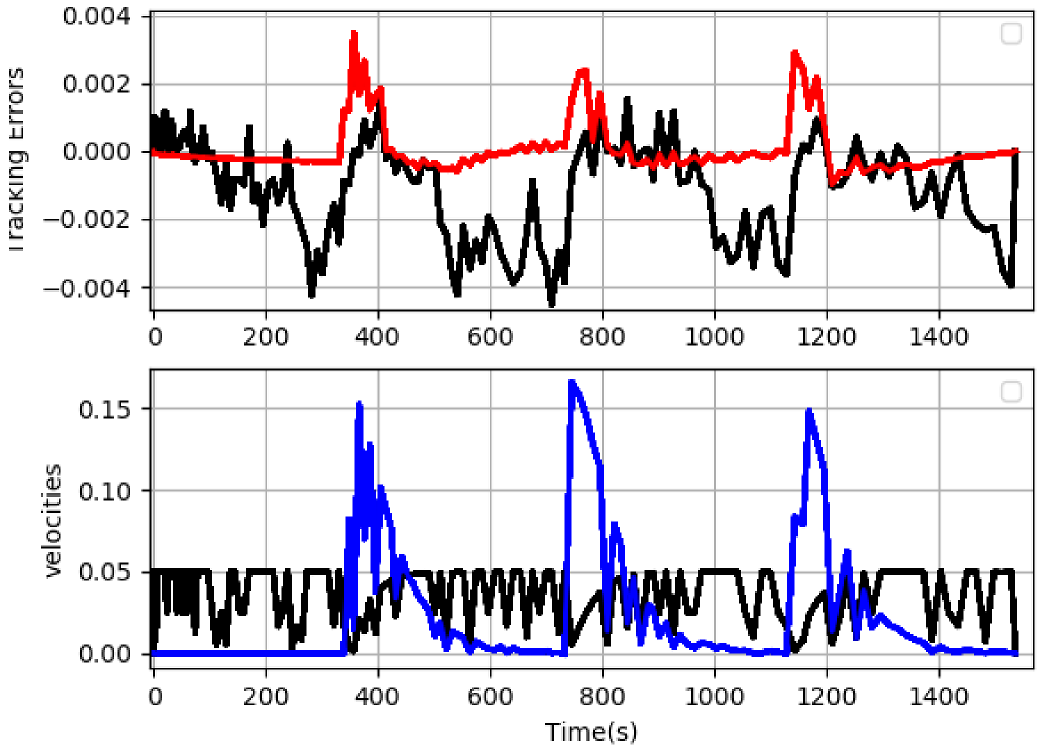


Figure 10: Tracking errors on x (black) and y (red). Linear (black) and angular (blue) velocities. b = 0.3

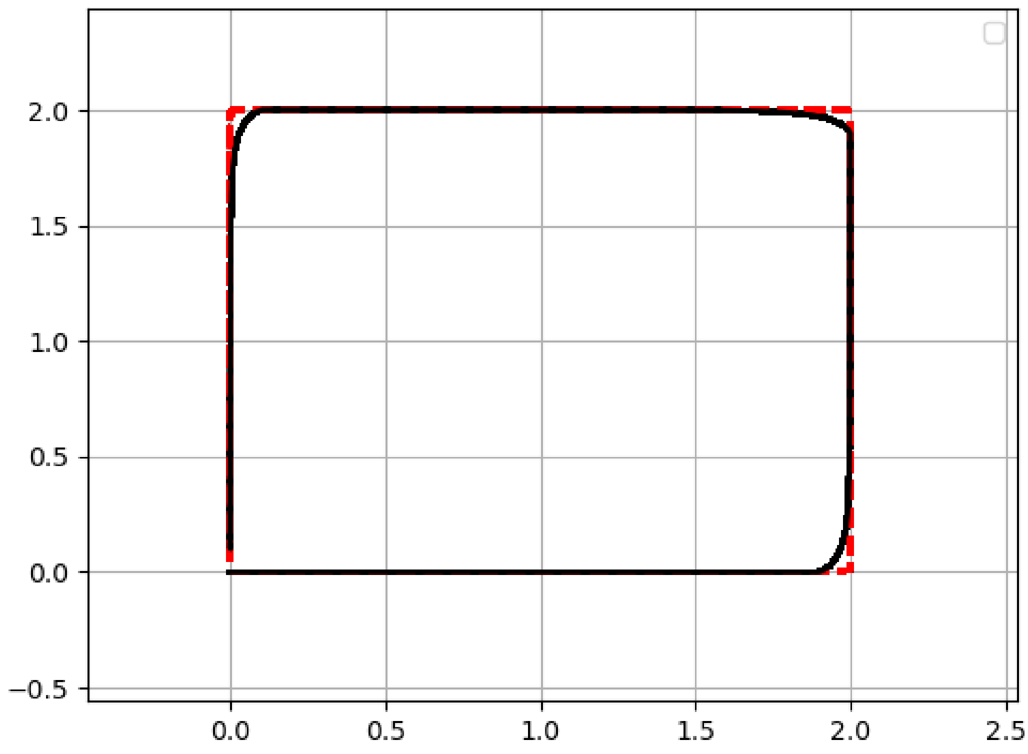


Figure 11: Figure 9: Square desired trajectory (red) and robot position (black) with b = 0.1

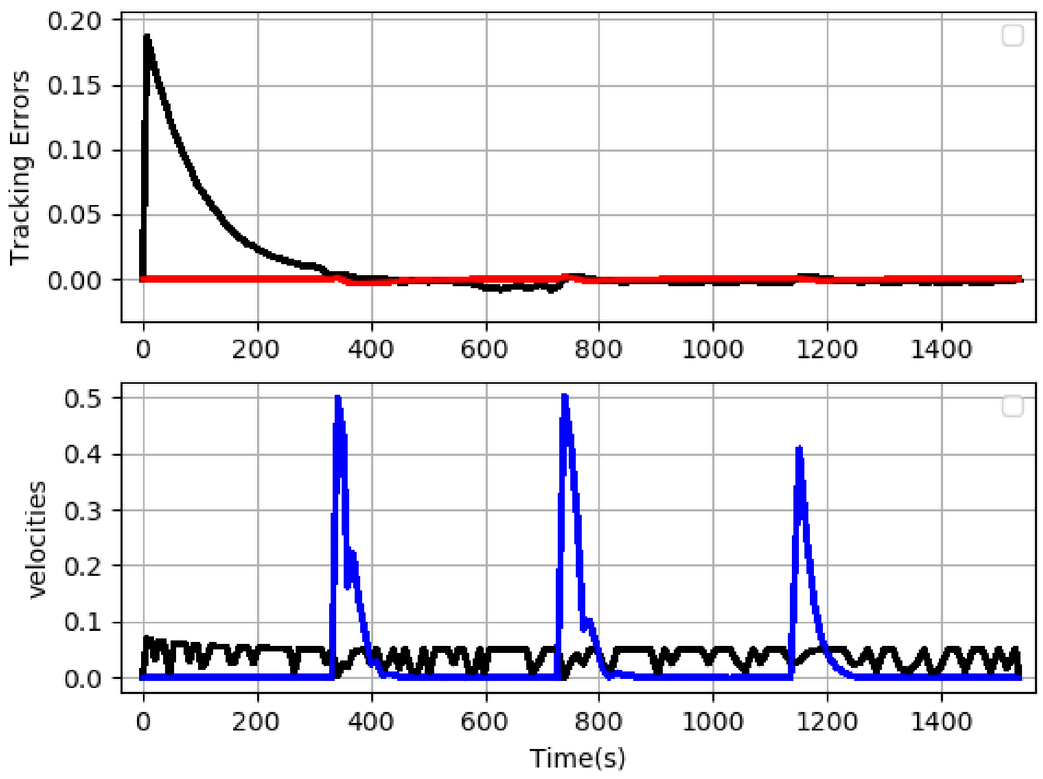


Figure 12: Tracking errors on x (black) and y (red). Linear (black) and angular (blue) velocities. b = 0.3

Positioning routine implementation

As already said, after the controller brought the turtlebot3 to the goal, it will start a positioning routine to adjust the position of the robot with respect to the AR Marker. This routine is needed to have a correct reading of the Id on the detected marker, reducing the lecture error due to distance and orientation angle, but it is also useful to overcome the lack of orientation control of the input/output feedback linearization controller.

The positioning routine is implemented as a service, that will be called by the main task after the controller gives a positive result for the goal reaching. The new service structure is defined in the *service.srv* file in the srv folder. It just consist of a *std_msgs/Int32* request that specifies the Id of the marker in front of which we want to place, and a *std_msgs/Int32* response whit a result value (as usual 0 for positive result 1 for negative).

The routine is structured as a class named **POS_ROUTINE** that will implement the service server and the needed publishers and subscribers. The main elements are described below:

- service_callback : it's called when a client send a request to the service server. When invoked will start the main loop of the positioning routine as the new thread *ctrl_loop_t* and will wait for the routine to finish while listening to the other callbacks of the topic on which is subscribed thanks to the *ros::spinOnce()* in the loop. When the routine finishes sends the response with the result value to the service client.

Note: the service client will be the main task (logistic_task described in the next chapters) in the final version, but as usual we want to test the code step by step in this bottom up approach to the mission. For this purpose a service_client package was implemented and it's located in the src folder. At the end of this section we will see how to use it.

- vision_cb : callback function called when a new message is published by Aruco on the */aruco_marker_publisher/markers* topic. *Receive a /aruco_msgs/MarkerArray* with the information about the recognized marker Id and relative pose and updates the current relative position and orientation used by the control loop.

- _cmd_vel_pub : publish the control output velocity to the */cmd_vel* topic to move the robot.

- ctrl_loop : consist of different subsequent step, running at 100 Hz like the controller. In first instance the robot will rotate until it receives the first Aruco lecture. Then it will start the positioning routine in front of the detected marker, accomplished in three step by the positioning(), adjusting() and rotating() functions.

The first step will make the robot point toward the center of the AR Marker by reducing under a certain threshold the error on the relative x position. Then in the second step, when the robot is pointing the marker, it will move forward or backward in order to not be too near or too far from the marker. This is accomplished controlling the relative z position value.

The last step will check if the relative orientation (pitch angle) is under the desired threshold, and in this case will stop the routine setting the *_finish* variable to true. Otherwise it will rotate the robot of a quantity inversely proportional to the distance from the center of the maker, so that when it's getting near it will not lose the marker from the field of view of the camera, and in the direction toward the marker normal. It will then proceed forward, rotating a little bit when the marker is exiting the field of view of the camera, until a certain distance is reached. At this point it will start again from the first step and repeat until the relative orientation is under the desired threshold.

To test this step we will need an empty scene with an Aruco marker positioned in it and we also need to configure and launch Aruco. This step will be described better in the next chapters, when the high level task will be discussed. For now we can just set the environment and launch Aruco.

>>roslaunch turtlebot3_gazebo turtlebot3_empty_world.launch

>>roslaunch ar_marker_test aruco.launch

Then we have to start the positioning routine service server and the service client

>>rosrun mobile_navigation positioning_routine

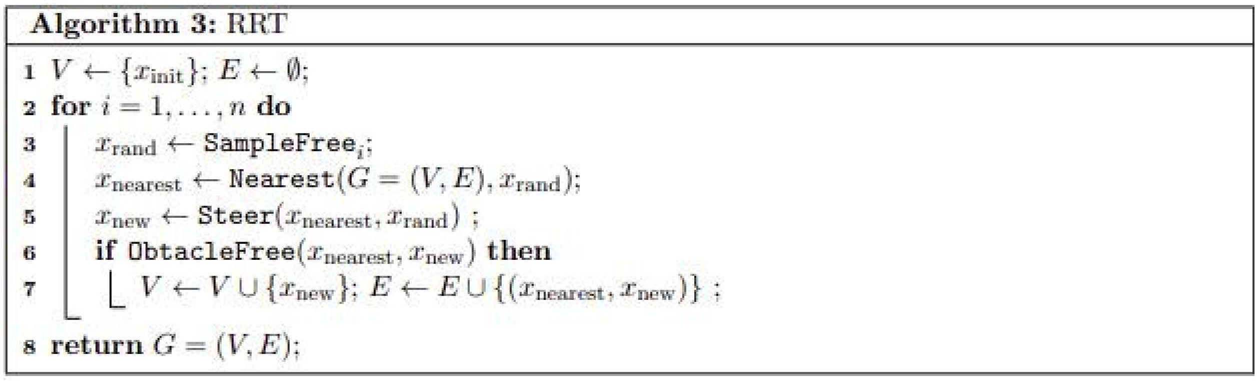
>>rosrun mobile_navigation service_client

At this point, entering an integer number different from 1 (exit condition) the service client will send a request to the service server and will wait for its response.

Planning

The problem of planning a trajectory for the robot can be broken down into finding a path and defining a required timing law on such path. So the next step is that of planning a desired path and publishing it on the */path* topic to send it to the controller. Looking at the final mission, that requires the robot to move in the presence of obstacles, it is necessary to plan motions that enable the robot to execute the assigned task without collisions. For this reason the adopted strategy is that of considering a motion planning strategy and to adopt a probabilistic approach. In particular we used the Rapidly-exploring Random Tree Method along with the A* graph search algorithm to find out the collision free path.

RRT is a single-query probabilistic method very fast in solving a particular instance of a motion planning problem, because of its tendency to explore only a relevant subset of C_{free}. It consist of incremental expansions of a tree T based on a simple randomized iterated procedure. A pseudocode of RRT from [3] is reported below.



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

First we add the initial pose to the tree, then the algorithm consist of iteration of four steps.

The first step is the generation of a random configuration x_{rand}. This configuration is not added to the tree but it only indicate the expansion direction of T.

Then the configuration $x_{nearest}$ that is nearest to x_{rand} is searched in T .

Next the steering function select a new candidate configuration x_{new} on the segment joining $x_{nearest}$ to x_{rand} at a distance δ from x_{near} .

Finally a collision check is performed on the segment joining $x_{nearest}$ and x_{new}, sampling the path and checking if all the sample are collision free. If it’s obstacle free, x_{new} is added to T together with the segment joining it to x_{near}.

For the graph search, we will consider that the arcs (segments) connecting the nodes (configurations) have a weight based on their length. In this way we can use A* algorithm to determine the minimum path. A* visits all the nodes in T and stores the current minimum paths from the starting configuration to the visited node.

The algorithm uses a cost function $f(N_i)$ which is an estimate of the cost of the minimum path that connects N_s to N_g passing through N_i , and is computed as

$$f(N_i) = g(N_i) + h(N_i) \tag{9}$$

where g(N_i) is the cost of the path from N_s to N_i as stored in the current tree T , and h(N_i) is a heuristic estimate of the cost of the minimum path between N_i and N_g . While the value of g(N_i) is uniquely determined by the search, any choice of h(·) such that not overestimate the cost of the minimum path from N_i to N_g is admissible.

In the following, a pseudocode description of A* is given from [2].

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

Both RRT and A* are implemented in a ROS node in the planner package. In this package we can find, in the include folder, the library defining the necessary class and structures, and in the src folder the source code of the planner.

RRT implementation

The planner is implemented as usual with a class named **RRT** defined in the library in the include folder. Inside it, it will also be useful to define some structured data to contain the nodes with all the needed attributes for the RRT and A* implementation: x and y position (double), a flag signing if a node it's the root node (bool), parent node index (int) , cost functions (double), and adjacency list indices (vec<int>).

In the source code in the src folder there is the implementation of the functionalities, consisting in subscribing to the */odom*, */map* and */goals* topics, publishing on ***/path*** (where the controller is subscribing) and on some topic for visualization on Rviz, in particular ***/tree_lines*** will show the path with green lines. The main elements of the code are described below:

- **odom_cb:** callback function called when a new odometry measurement is published on /*odom topic. Receive a nav_msgs/Odometry* message and updates the current position and orientation for the starting pose.

- **map_callback:** callback function that receive the map via the */map topic. Receive a nav_msgs/OccupancyGrid* message. An occupancy grid is a vector whose elements correspond to a square cell of the map with a specified resolution. The values of the Occupancy Grid can be 0 (white) if the position is free, -1 (gray) if the position is unexplored and 100 (black) if the position is occupied by an obstacle.

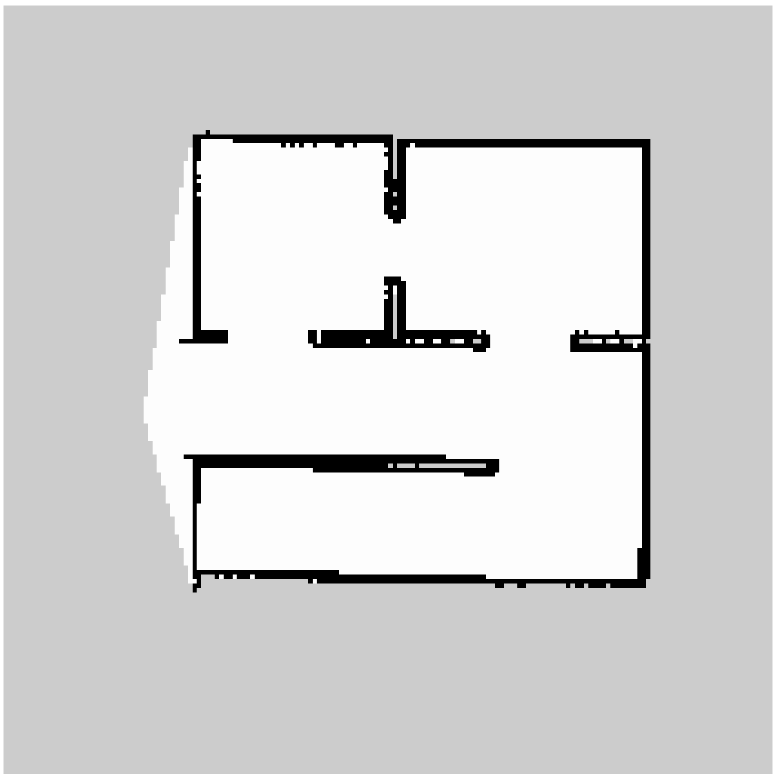


Figure 13: Occupancy grid map

Note: the map occupancy grid will be provided from the phase 1 mapping described in the next sections.

- **goal_callback:** callback function called when a new goal is published on /*goals topic. Receive a geometry_msgs/Pose* corresponding to the goal configuration. When called, this callback starts the planning algorithm. First it will wait until it has received the map and a first odometry lecture. Then it will start the RRT loop, repeated until the found configuration is sufficiently near to the goal configuration or until the MAX_ITERATION number is reached. As illustrated in the pseudocode description in the previous section, the steps to be done in the loop are implemented with subsequent functions.

- **sample:** This function returns a sampled point. The random sample is taken using the **std::mt19937** class, a very efficient pseudo-random number generator that uses the Mersenne twister algorithm. It is more efficient for real-world like random numbers and has a longer period compared to rand(). The random configuration is taken in the region limited by *x_limit_top*, *x_limit_bot* , *y_limit_left* , *y_limit_right* .

- **nearest:** Takes as inputs the tree and the previous sampled point and returns the index of the nearest node of the tree.

- **steer:** Expand the tree in the sampled point direction for a specified STEER_LENGTH. Takes a inputs the previously found nearest node and sampled point and returns the new node.

- **check_collision :** Takes as input the previously found nearest node and new node and returns a boolean indicating if the path between them is collision free. This path is sampled with a step size of 0.001. For each occupied cell in the occupancy grid, a collision is detected if the configuration is in the square of sides given by INFLATION_ RADIUS and centered in an occupied cell. In this way the planner will consider the inflated map in the figure below.

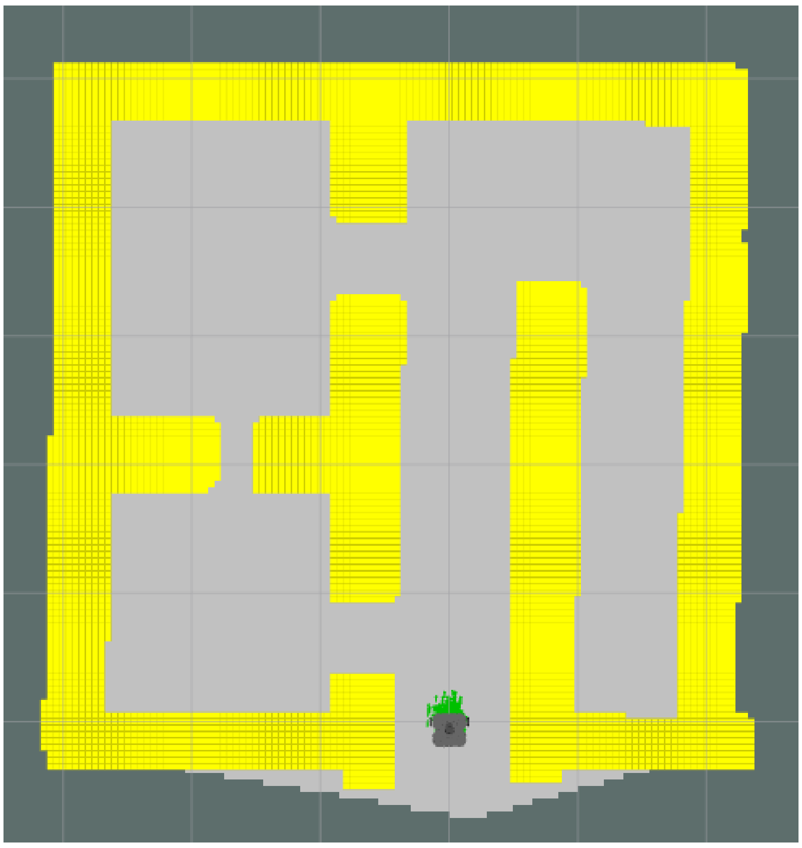


Figure 14: Inflated map

- is_goal : This function takes as input the last added node and the goal position and returns a boolean value indicating if they're close enough, based on TERMINATE_LENGTH, to terminate the algorithm.

A* implementation

The last function to be analyzed in the source file is the **find_path_A_star** function, that implements the A* graph search algorithm. This function is called when the previous described *is_goal* function returns true, so that the last added node is sufficiently near to the goal. At this point, the *find_path_A_star* function takes as input the tree (vec<Node>) and the latest added node (Node) and returns the best path (vector<Node>).

Assuming that all the nodes are initially unvisited, the algorithm starts from q_s, identified in the tree by the *q_s_index* , set as q_{best} , and visits all the nodes to which q_s is connected.

The indices of these nodes are inserted in a list *open_list_indices*, containing all the indices of the nodes visited of tree, and from which the tree can be extended. At each iteration the cost $f(q_i)$ of each node with index in *open_list_indices* is calculated if they have been unvisited and the current q_{best} is inserted as their parent. If a node has been already visited and if the cost $g(q_i)$ of the new path is lower, the cost and its parent are updated. Then the index of node in the list *open_list_indices* with the lowest $f(q_i)$ is set as q_{best} and it is extract from the list. In the next iteration the algorithm extract the new q_{best} node index, and visit all its adjacent nodes inserting them in *open_list_indices*. In this way the best path is always saved during the execution of the algorithm.

The algorithm terminate when $q_{best} = q_g$ or *open_list_indices* is empty. In this last case there exists no path from the start to the goal configuration. Otherwise the best path can be reconstructed by backtracking from q_s to q_g .

In the figures below we can see the results of different simulations, that points out the effect of changing some parameter. Using different values of INFLATION_ RADIUS can create a narrow passage between room 1 (lower left angle) and room 2 (higher left angle). In this way we can see that using the an high value of STEER_LENHT, like 0.7, will make the robot choose a path (Figure 15), while lowering STEER LENGHT to 0.2 will make it possible for the planner to explore narrow passages(Figure 16).

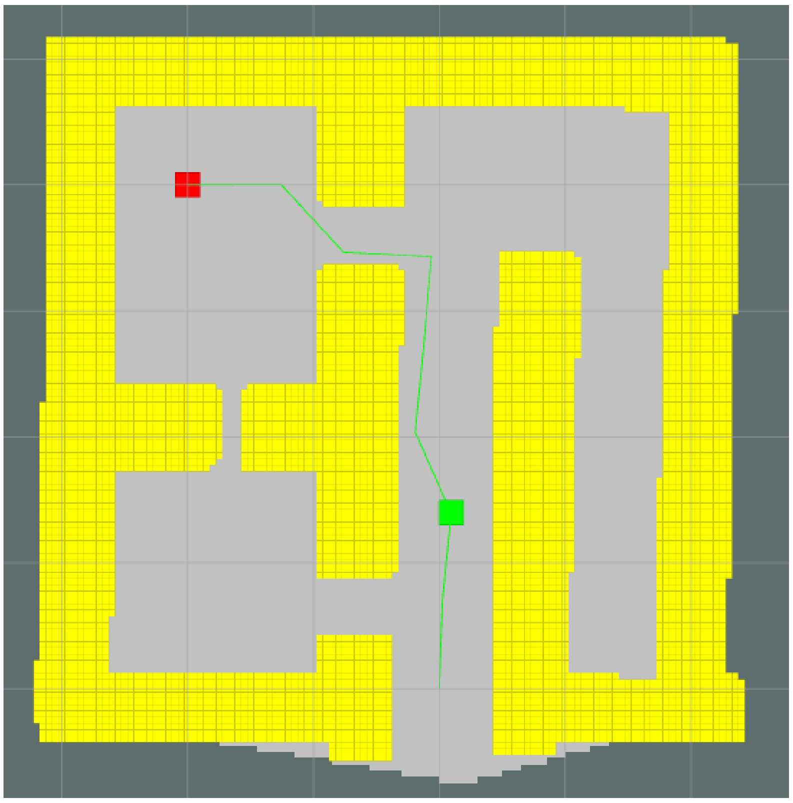


Figure 15: path generation with STEER_LENGHT = 0.7

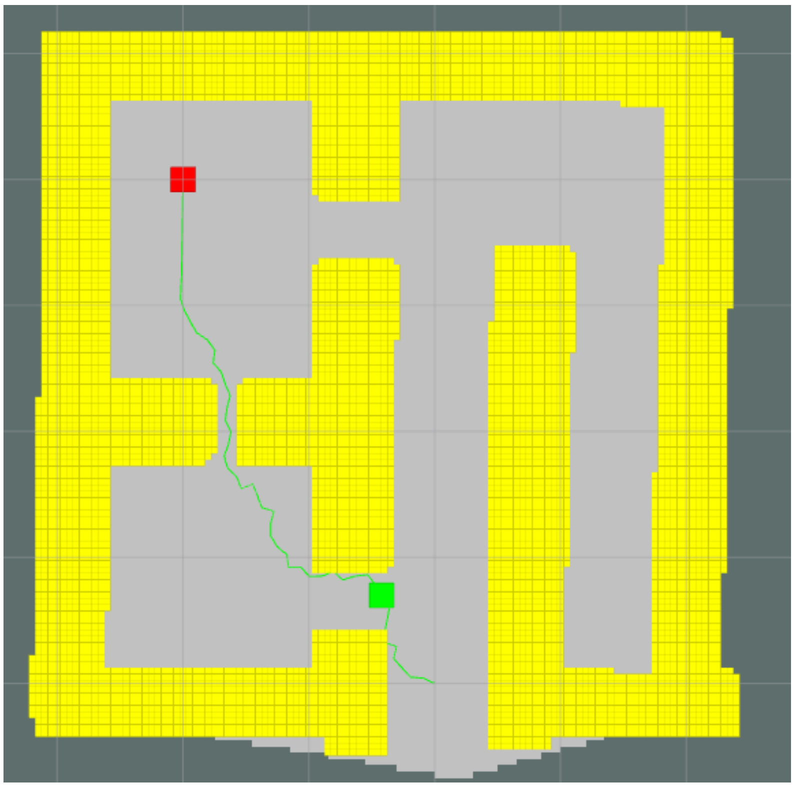


Figure 16: path generation with STEER_LENGHT = 0.1

Trajectory planning

Once we have found all the nodes of the path, going from the starting position to the desired goal, we have to define a time law on that path.

We can consider the linear segment connecting each node j of the tree to the previous one (j-1), that will have parametric representation (from [2]):

$$\boldsymbol{p}(s)=\boldsymbol{p}_j+\frac{s}{\|\boldsymbol{p}_j-\boldsymbol{p}_{(j-1)}\|}(\boldsymbol{p}_j-\boldsymbol{p}_{(j-1)}) \tag{10}$$

Differentiating 10 with respect to s gives the velocity and acceleration of p :

$$\frac{d\boldsymbol{p}}{ds}=\frac{1}{\|\boldsymbol{p}_j-\boldsymbol{p}_{(j-1)}\|}(\boldsymbol{p}_j-\boldsymbol{p}_{(j-1)}) \tag{11}$$

$$\frac{d^2\boldsymbol{p}}{ds^2}=0 \tag{12}$$

With these equations we can iterate on each couple of points, corresponding to adjacent nodes of the tree, set the velocities for the desired trajectory to follow, and send it to the controller.

In this way, choosing the parameter s equal to t, and controlling the robot iterating on time, we can make the trajectory faster or slower by multiplying the t with a scaling factor, that will actually change the velocity.

Mission

After we have implemented a working controller and planner that allow us to move the robot in the map without colliding with obstacles, we will face the high level mission. We can distinguish two different phases:

- **Phase 1** is a preliminary step in which the goal is to create the map of the environment that, as we have already seen, will be used by the planner to generate a collision free path. In this phase the robot has to localize himself into the map that is creating, for this reason the data of the sensor will be used to implement SLAM technique for simultaneous localization and mapping. The robot will explore the environment driving teleoperated by the user with the use of a specific ROS node.

- **Phase 2** is the real logistic task. In this phase we will use the previously implemented controller and planner to autonomously drive the robot in the warehouse. Then, with the help of the already seen Aruco library for computer vision, the robot will autonomously place itself in front of an AR Marker, read its Id and drive to the requested room.

Each phase can be started with an apposite launch file placed in the launch folder of the *logistic_task* package.

Phase 1 – mapping

The mapping task can be started with the *phase1.launch* file in the logistic_task package. This launch file will call other launch file and run some ROS node. First of all it will launch the already seen *small_wh.launch* launch file, that will simulate our custom world and spawn our turtlebot3 model.

The next roslaunch called is the *aruco_launch,* located in the *ar_marker_test* package. We have already used this launch file for the positioning routine, but we have not seen it in deep. In this launch file it will be called the launch file of the **aruco_ros** package, that is a ROS wrappers of the Aruco Augmented Reality marker detector library. In particular we have just called the *aruco_marker_publisher* node, for the detection of multiple markers, and configured some parameters like the **marker_size** and the **camera_frame** to make it work for our specific application.

In the phase 1 the Aruco library is only needed for the execution of the **ar_listener** node. This node will work during the teleoperated exploration and the creation of the map by the robot, and will subscribe to the *aruco_marker_publisher/markers* topic, waiting for Aruco to detect an AR Marker. When Aruco will publish the marker Id and pose on this topic, the **topic_cb** function will use the **tf** package to transform the pose of the marker from the camera frame to the map frame. At this point the position of the marker, and so of the respective room linked by the Id, will be saved in the **parameters server** to be used for the next phase.

Another package that we will need is the one for teleoperating the turtlebot3 with our PC keyboard. To do that we just need to read data from the keyboard and publish a respective *geometry_msgs/Twist* message on */cmd_vel* topic. By the way it can be useful and quicker to use the *turtlebot3_teleop_key* launch file of the **turtlebot3_teleop** package provided with the turtlebot3 package.

The last launch file included is the one for the SLAM. We can perform 2D SLAM with **gmapping**, that brings OpenSLAM libraries in ROS. Remembering that the *slam_gmapping* node will need as input the **tf** information, the **laser scan** data and the **odometry** information, to adjust the odometry estimated pose with the sensorial measurement, all we have to do is to create a proper launch file and tune the parameters in the configuration file to have a good behaviour. In particular, starting with the turtlebot3_slam package and the *turtlebot3_slam.launch* file, we configured the slam method to be gmapping and chose the correct parameters for the base, odom and map frames. The other parameters, assigned in the *gmapping_params.yaml* file in the config folder specify other configurations that can be changed, if the results are not satisfying, according to the documentation suggestion [4].

At this point we're ready to run our launch file

```
>>roslaunch logistic_task phase1.launch
```

It will open a new terminal with the teleoperation controller waiting for user input and will also open Rviz, included in the *turtlebot3_slam.launch.*

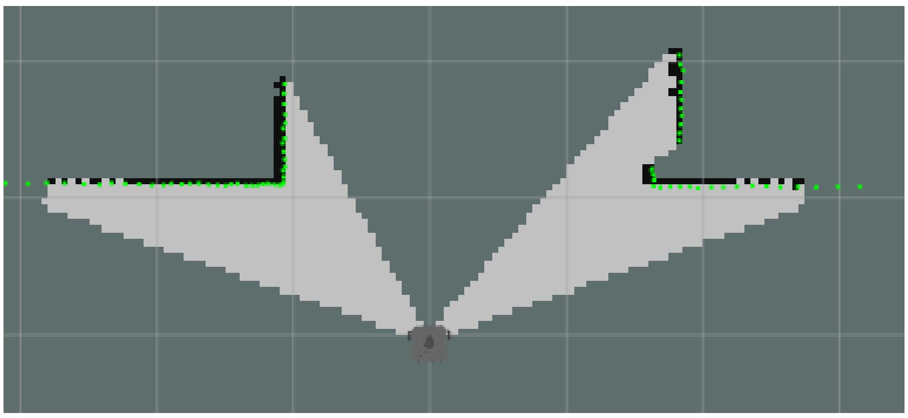


Figure 17: Mapping

Now we can control our turtlebot3 and make it explore the environment and create a complete map. Then we can use the **map_server** package map_saver command to save the created map in a map folder to make it available for the next phase. This comand will create both an image of the created occupancy grid map *small_wh.pgm* and a configuration file *small_wh.yaml* that specifies some parameters like resolution, origin ecc..

Phase 2 – Logistic task

After Phase 1 is completed and the map is saved, with the navigation system set up with planner and controller and the vision system to recognize AR Marker, we have all we need for starting the logistic task. The robot must transport an object from the warehouse to a destination room. Starting from a random position, the robot must move into the warehouse, search and read for the AR Marker: this contains the ID of the room where the transported object should be brought. Then, move the robot to the target room.

During the whole task, the robot must be able to localize itself into the map without using precise information provided by the simulator, and so odometry and sensor measurement will be used. Since we're in a static environment and we already have a map to which refer, we can solve the localization problem with **AMCL** (Augmented Monte Carlo Localization). AMCL is a probabilistic localization system for a robot moving in 2D, that uses a particle filter to track the pose of a robot in a known map. The AMCL node takes as input a **laser-based map**, **laser scan** data and **tf** and gives as output the **pose estimations**. As for the *gmapping* all we ned is to set and tune a list of parameters, writing an apposite launch file that will start the node and customize its configuration.

In particular, starting with the turtlebot3_navigation package and the *turtlebot3_navigation.launch* file, we removed the inclusion of *move_base* since we have our custom controller and planner, and configured the localization method to be a*mcl*, *i*n*cl*uding the am*cl.la*un*c*h. In this last launch file there are all the parameters to set up like the scan topic, odometry frame and base frame definitions. A very important parameter for the correct behaviour of the node is the **initial pose**. We can make sure that this value is good by visualizing the scan message in Rviz and making sure that it overlaps with the walls in the map. The other parameters are described in the documentation [5] and can be changed if the results are not satisfying.

For starting Phase 2 as usual we wrote a proper launch file named *phase2.launch* and placed in the launch folder of the **logistic_task** package. At this point we know what is the purpose of each file included: *small_wh.launch* of turtlebot3_gazebo package is for the gazebo simulation of the custom environment, *aruco.launch* of the ar_marker_test package is for the Aruco libraries and computer vision, *turtlebot3_navigation.launch* of turtlebot3_navigation package is for localization with AMCL. Then we have to start three nodes: *rrt_node* from planner package and *mobile_navigation* of the mobile_navigation package, that are the nodes implemented in the previous chapters, and finally the *logistic_task* node of the *logistic_task* package. Finally we can launch the positioning routine server, that will wait to be called by the client, implemented in the logistic task node.

This last node is the one implementing the logic of the mission and it's described below.

Logistic task implementation

In this section we will analyze the crucial part of the **logistic_task.cpp** source code, that implements the logic of the phase 2 of the mission.

As usual we will design a class **LOG_TASK** that will help us implementing all the needed functionalities. The logic of the logistic_task is that of a **state machine**, in which we can know in what sub-phase of the mission we are based on the current state. The core of the algorithm is the **task_loop** that is started as a thread when a new LOG_TASK object is created and the **run** member function is called. At each loop start the state is checked with a **switch case** statement and, based on its value, something will be done. The state is initialized at 10, a value that will bring the system in the "**manual**" mode, in which the console will guide the user to run the node inserting an input value. We will list below the state and their role.

- **STATE 10:** Starting state of the system. Call the **take_input** function that permits the user to specify in input the desired action. Insert 0 for starting the mission will set the **state to 0**. Other choice are made for debug and allow to drive the robot directly in room 1, room 2, or in a position specified clicking with the mouse on the map in Rviz (input respectively 1, 2 or 3). This choice will make the state jump to 5 after setting the position of the room as goal (in the case of input 1 and 2) or make the state jump to 9 (in the case of input 3).

- STATE 0: Sending the first goal, warehouse position, to the controller. The warehouse position is taken from the parameter server, where it has been set in the phase 1 as we have seen. Set **state to 1**.

- STATE 1: Waiting for result from the controller. The result will be received on the already seen */result* topic. Here if the result is 0 the sub-task of driving to the warehouse succeeded and we set **state to 8**, otherwise the sub-task failed and the state returns to 10 where we can manually choose an action and start from the beginning.

NOTE: state jumps to 8 because here it will call the positioning routine server that was added in a second moment.

- STATE 8: Searching the AR marker to read. Here we will send a request to the positioning routine service server indicating the Id of the marker to search and will be blocked waiting a response. If the sub-task succeeded set **state to 2.**

- STATE 2: Reading AR marker Id. Wait for the Aruco detected Id from the *aruco_marker_publisher/markers* topic and save it in the **current_id** std_msgs/Int32 variable. Set **state to 3**.

- STATE 3: Setting the new goal as room 1 or room 2 position based on the detected Id. If the value of the detected marker Id is 1 then retrieve from the parameter server the position of room 1 and drive to that position. Otherwise if the value of Id is 2 do the same for room 2. Set the **state to 4**. If the Id is different from this two values return to the starting **state 10**.

- STATE 4: Waiting for AR marker to leave the camera field of view. In this transition state we only receive the pose published by Aruco to know when the AR marker is out of view. This will allow the robot to rotate on itself of 180° and will stop when will detect the Id of the Aruco marker on the opposite wall. **State set to 5**.

- STATE 5: Waiting for result from the controller. The result will be received on the already seen */result* topic. Here if the result is 0 the sub-task of driving to the desired room succeeded, otherwise it failed. In both cases **state returns to 10** where we can manually choose an action and start from the beginning.

As we have seen, the logistic_task node has to communicate with the other node that we have implemented before, by publishing and subscribing to certain topics. Moreover it will call the positioning_routine service node, so it has to start a service client able to send a request. We will describe below these core elements:

- _client: is the service client needed to send the request to the positioning routine service server. The structure of the service is the same of the one described above for the server and it's in the srv folder.

- _goal_pub: publisher of the goals on the */goal* topic that is subscribed by the planner to plan the trajectory.

- topic_cb: callback function of the _topic_sub subscriber to the *aruco_marker_publisher/markers* topic, where the Aruco detection are published. Based on the current state this function will update the current Id (state 2) or wait for the marker to leave the camera field of view before changing the state (state 4).

- result_cb: callback function of the _controller_result subscriber to the */controller_result* topic, where the controller publishes its results. Based on the current state this function will know if the result is referred to the first goal (drive to warehouse state 1) or to the second (drive to room1/room2 state 5) and will update the state according to that.

- clk_p_cb: callback function of the _clk_p subscriber to the */clicked_point* topic, where we can publish a geometry_msgs/PointStamped msg clicking on the map with the Rviz tool “Publish Point”.

Simulation and results

As we stressed above, we have already simulated and tested all the single steps during the development. Now we will simulate the complete task and see the results of the whole system at work.

First we will consider to use the following value for the controller, chosen with a trial and error approach:

$$k_1=0.1$$
$$k_2=0.1$$
$$b=0.3$$

With this configuration we can see that the point B (on which the errors are computed) track very well the desired trajectory. There are some peaks corresponding to the phases in which the robot starts a new trajectory, and so at the beginning of the task following the trajectory for the warehouse , at half task done when the robot start to follow the trajectory for the desired room, and at the end when the robot approaches the final destination.

The velocities are always in the physical limit, specified in the parameter server, and in case they try to exceed their limits will be saturated and the trajectory will be uniformly scaled.

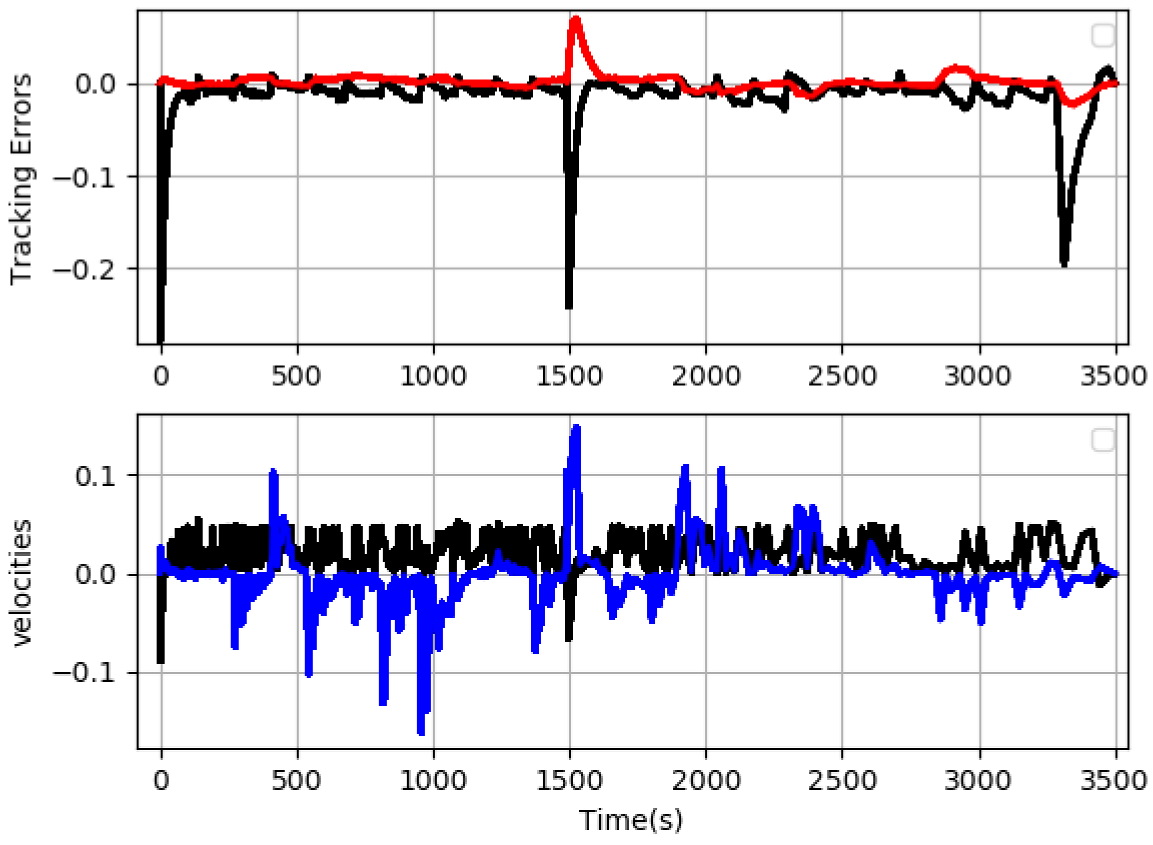


Figure 18: Tracking error on x (black) and y (red). Linear (black) and angular (blue) velocities.

Finally we can see that the choice of b leads to a smooth path for the robot, but it can be a problem in our case, with a structured environment with some narrow passage.

We can try to use a lower b to make the robot follow more strictly the trajectory, that is computed to be obstacle free, and so safe from impact.

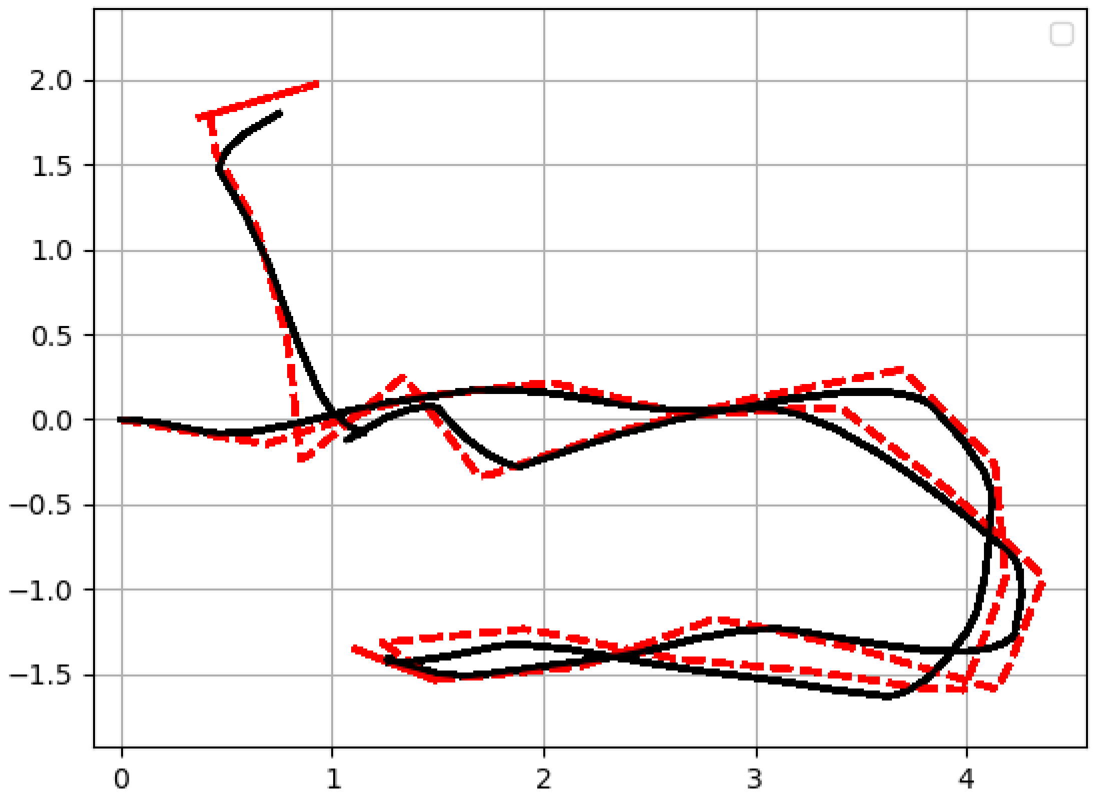


Figure 19: Desired trajectory (red) and robot position (black). With b = 0.3.

In this last case we will see that the path is less smooth, but follow more strictly the desired traectory.

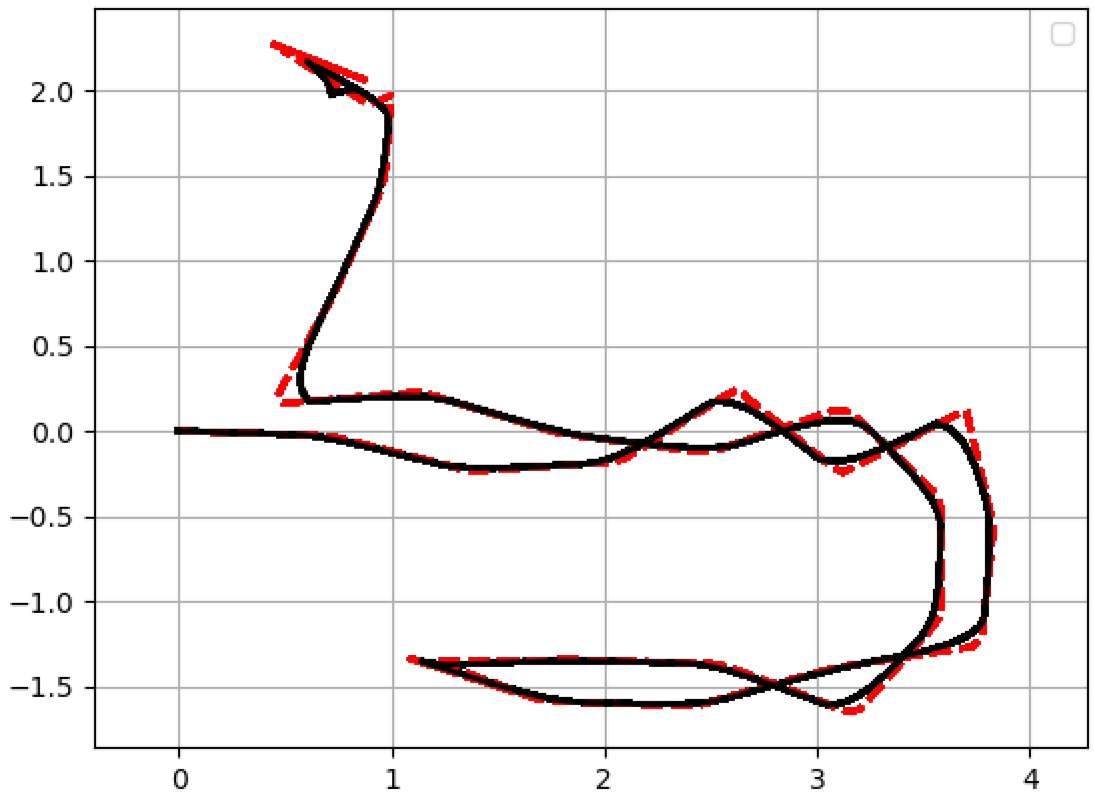


Figure 20: Desired trajectory (red) and robot position (black). With b = 0.15

Final considerations and future works

We can say that the system shows good results, and that is also adaptable to different scenario. Performances can be improved modifying some configuration parameter based on the running configuration, for example in the case of wide or narrow passages and of big or small environment.

Moreover, thanks to its modularity, it’s easy to provide new features to implement other desired behaviours. For example, two features already implemented but that were not very useful for our use case are the **IMU based odometry** and the **laser scan feedback** to recover from wall proximity. In particular this last is very important, but in a very small environment like the one designed, with some narrow passage as we have seen, it will be very common to be near to the wall and exceed the threshold. So, since the robot collided not so frequently, we decided to not use this features to give more continuity to the task, in order to have cleaner plots, but it can be useful in large spaces with a big value of the INFLATION_RADIUS.

Another future work could be that of implementing different type of controllers to compare the achieved results, and to plan a smoother path instead of one composed of linear segment.

References

[1] https://emanual.robotis.com/docs/en/platform/turtlebot3/overview/

[2] B. Siciliano et al. Robotics: Modelling, Planning and Control. Springer, 2009.

[3] arXiv:1105.1186

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

[5] http://wiki.ros.org/amcl

[6] http://gazebosim.org/tutorials?tut=ros_gzplugins

[7] https://github.com/s-contento/TP