Robotics Lab: Homework 4
Students: Anzalone Claudio, Maisto Paolo, Manzoni Antonio

Here is the link to my public repo on github:
https://github.com/PaoloMaisto/HW4_RL_Maisto_Paolo.git

We want to specify that all the participants of the group worked at each stage of the development of the project. In order to simplify the drafting of the report (as recommended by the professor) we have fairly divided the writing of the development of the various points.

Control a mobile robot to follow a trajectory

1. **Construct a gazebo world and spawn the mobile robot in a given pose**

   (a) Launch the Gazebo simulation and spawn the mobile robot in the world rl_racefield in the pose

   $$x = -3 \quad y = 5 \quad yaw = -90 \deg$$

   with respect to the map frame. The argument for the yaw in the call of spawn_model is Y.

Used commands to switch to the gmapping branch
  $ cd catkin_ws/src/rl_fra2mo_description/
  $ git checkout gmapping

Due the fact, we have two branches on github (main and gmapping) firstly we had to use the command git checkout gmapping in order to switch to the gmapping branch.
After that we moved within the package "*rl_fra2mo_description*" and in particular we have modified the position's values and we added the yaw argument within the file "*spawn_fra2mo_gazebo.launch*" which is located inside the launch folder. What is made it's shown in the following figures:

```
4    <!-- these are the arguments you can pass this launch file, for example paused:=true -->
5    <arg name="paused" default="false"/>
6    <arg name="use_sim_time" default="true"/>
7    <arg name="gui" default="true"/>
8    <arg name="headless" default="false"/>
9    <arg name="debug" default="false"/>
10   <!-- Pose modified x=-3, y=5, yaw=-90° to satisfy the 1.a point-->
11   <arg name="x_pos" default="-3.0"/>
12   <arg name="y_pos" default="5.0"/>
13   <arg name="z_pos" default="0.0"/>
14   <arg name="yaw_or" default="-1.57"/>
15   <env name="GAZEBO_MODEL_PATH" value="$(find rl_racefield)/models:$(optenv GAZEBO_MODEL_PATH)"/>
```

```
31   <!-- Run a python script to the send a service call to gazebo_ros to spawn a URDF robot -->
32   <!-- Add yaw argument called Y -->
33   <node name="urdf_spawner" pkg="gazebo_ros" type="spawn_model" respawn="false" output="screen"
34   args="-urdf -model fra2mo -x $(arg x_pos) -y $(arg y_pos) -z $(arg z_pos) -Y $(arg yaw_or) -param robot_description"/>
```

Used commands to launch the Gazebo simulation
  $ roslaunch fra2mo_2dnav fra2mo_nav_bringup.launch

As we can see from the the following picture the robot pose in the world rl_racefield has been implemented succesfully:
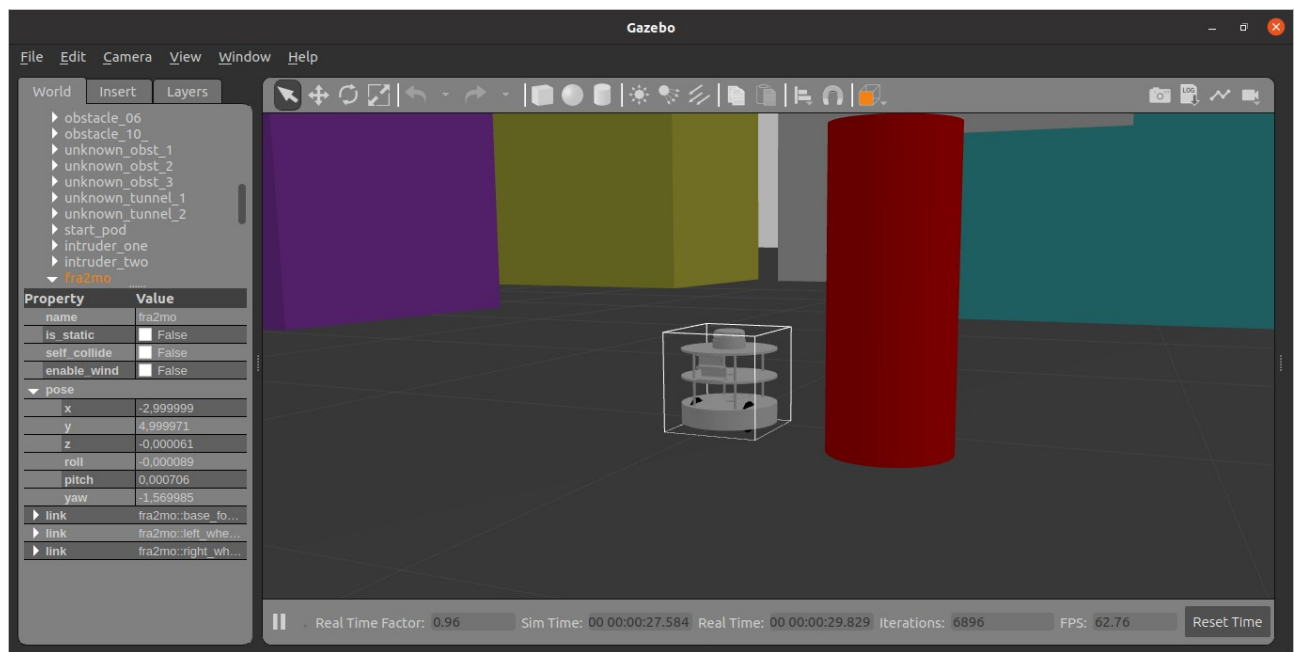
(b) Modify the world file of rl_racefield moving the obstacle 9 in position:

$$x = -17 \quad y = 9 \quad z = 0.1 \quad yaw = 3.14$$

In this section we have modified the file *"rl_race_field.world"* contained in the worlds folder of the rl_racefield package in such way to obtain the x, y, z positions and the yaw requested as in the following figure:

```
75        <include>
76          <name>obstacle_09</name>
77          <pose> -17 9 0.1 0 0 3.14159</pose>
78          <uri>model://obstacle_09</uri>
79        </include>
```

We can see in the following picture that the pose of the obstacle 9 in the world *rl_racefield* is the same of what we have developed in the code:
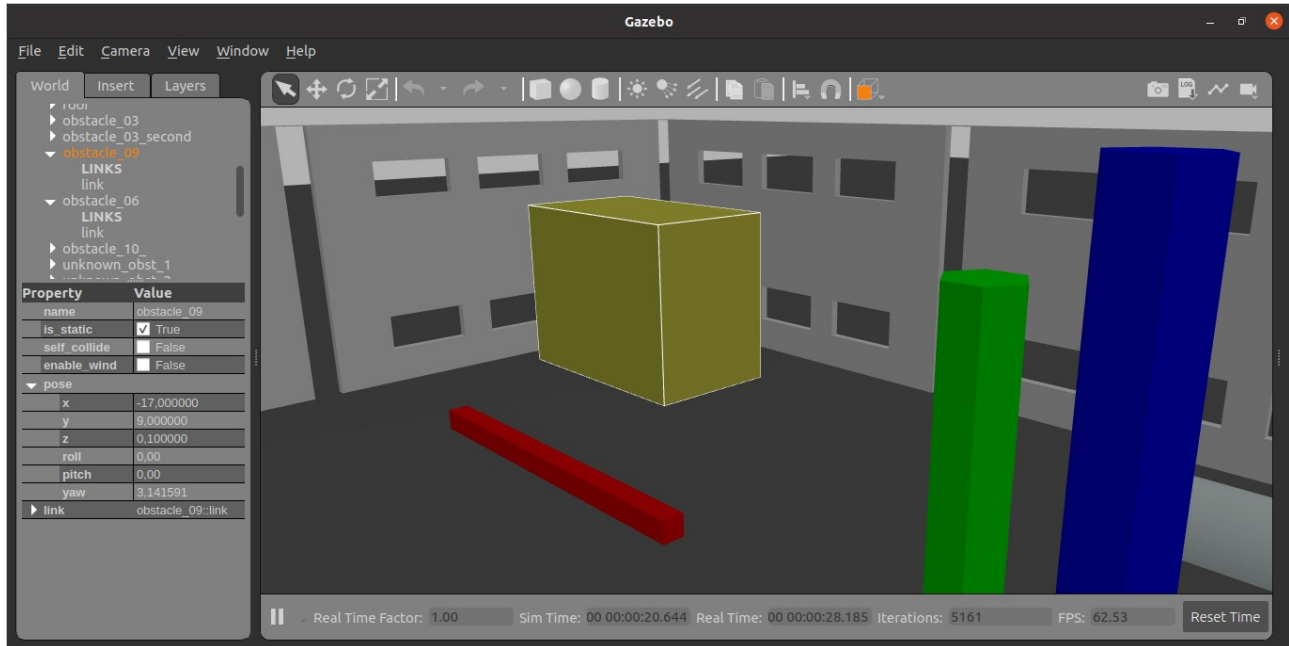
(c) Place the ArUco marker number 115 on obstacle 9 in an appropriate position, such that it is visible by the mobile robot's camera when it comes in the proximity of the object.

Within the folder models located inside the package *"rl_racefield"* we took the file *"marker_0.sdf"* and we have modified the names in an opportune way. Then we have renamed the folder in *"aruco115"* which contains a file sdf *"aruco115.sdf"*, the *"model.config"* file and the folder *"material"* with the relative *"scripts"* and *"textures"* folders. We have modified the folders renaming all the files with the prefix *"aruco115"* and then we added the image of the aruco marker 115 called *"aruco115.png"* that has been created through the link https://chev.me/arucogen/ given by the homework' trace. To insert this marker inside the world file we have modified the file *"rl_race_field.world"* , and in particular we have uncommented the rows of the code of the marker and we have modified the pose in such way to positioning the aruco marker on the obstacle 9, as it is shown in the following figures:
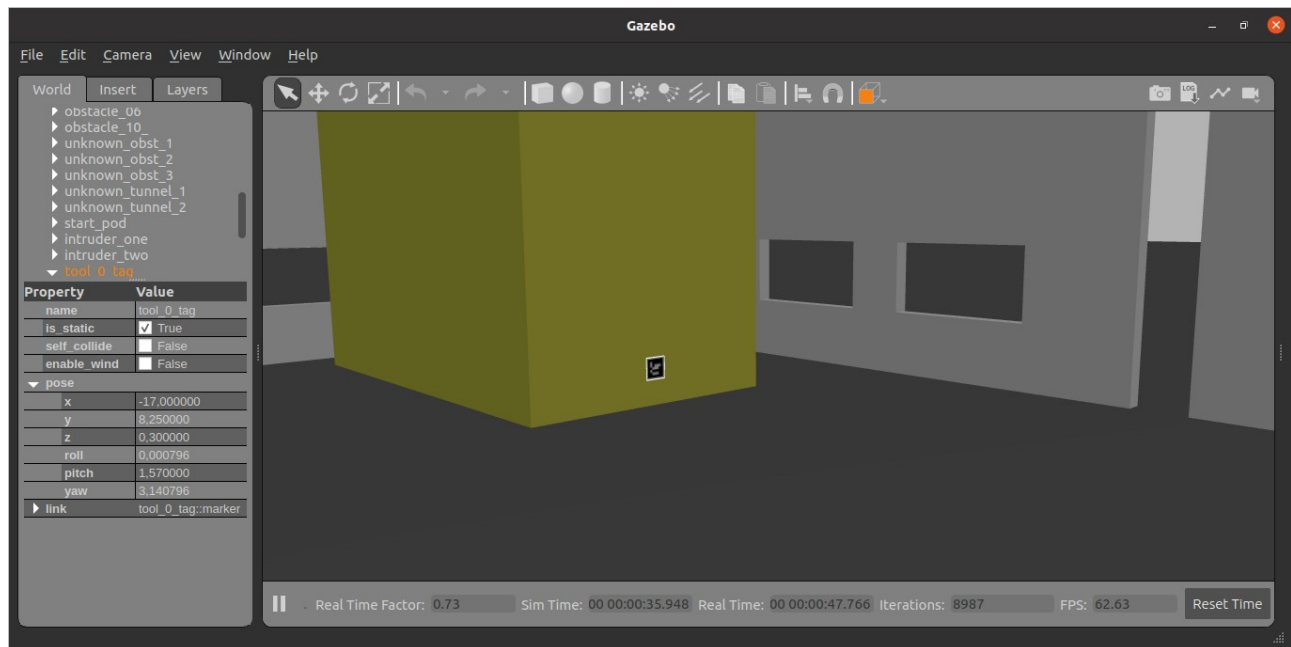
Used commands
```
$ cd catkin_ws
$ roslaunch fra2mo_2dnav fra2mo_nav_bringup.launch
```

2. **Place static tf acting as goals and get their pose to enable an autonomous navigation task**

   (a) Insert 4 static tf acting as goals in the following poses with respect to the map frame:
       - Goal_1: $x=-10$ $y=3$ $yaw=0$ deg
       - Goal_2: $x=-15$ $y=7$ $yaw=30$ deg
       - Goal_3: $x=-6$ $y=8$ $yaw=180$ deg
       - Goal_4: $x=-17,5$ $y=3$ $yaw=75$ deg

       Follow the example provided in the launch file rl_fra2mo_description/launch/spawn_fra2mo_gazebo.launch of the simulation.

In this point we had to modify the file *"spawn_fra2mo_gazebo.launch"* in the launch folder within the package *"rl_fra2mo_description"* in order to set the desired pose specified by the goals.

To fulfill the goals it has been used the quaternion convention to define the orientation.

The quaternion is defined by 4 parameters (the first is the scalar part of the quaternion defined by the cosine of the angle divided by 2, while the last three are specified by the sine of the angle divided by 2).

$$Q = \{\eta, \epsilon\} \qquad \eta = \cos\frac{\vartheta}{2}$$
$$\epsilon = \sin\frac{\vartheta}{2}\, r$$

In the following figure we'll show how it has been implemented in code:

```
46
47    <!--Static tf publisher for goal1-->
48    <!-- Define publisher tf nodes for four goals (2a point) -->
49    <node pkg="tf" type="static_transform_publisher" name="goal_1_pub" args="-10 3 0 0 0 1 map goal1 100" />
50    <node pkg="tf" type="static_transform_publisher" name="goal_2_pub" args="-15 7 0 0 0 0.25882 0.96593 map goal2 100" />  <!--This is for 2a point -->
51    <node pkg="tf" type="static_transform_publisher" name="goal_3_pub" args="-6 8 0 0 0 1 0 map goal3 100" />
52    <node pkg="tf" type="static_transform_publisher" name="goal_4_pub" args="-17.5 3 0 0 0 0.60876 0.79335 map goal4 100" />
53
```

Where the first 3 arguments are relative to the position and the remaining ones are relative to the quaternion.

(b) Following the example code in fra2mo_2dnav/src/tf_nav.cpp, implement tf listeners to get target poses and print them to the terminal as debug.

In this point we have implemented the listeners to get the targets poses and then we have printed them to the terminal.

Firstly, we had to go inside the file *"tf_nav.cpp"* inside the folder *"src"* of the package *"fra2mo_2dnav"* and define the poses of the goals as shown in the following picture:

```
 9        // Define pose of four goals (2b point)
10        _goal1_pos << 0.0, 0.0, 0.0;
11        _goal1_or << 0.0, 0.0, 0.0, 1.0;
12
13        _goal2_pos << 0.0, 0.0, 0.0;
14        _goal2_or << 0.0, 0.0, 0.0, 1.0;
15
16        _goal3_pos << 0.0, 0.0, 0.0;
17        _goal3_or << 0.0, 0.0, 0.0, 1.0;
18
19        _goal4_pos << 0.0, 0.0, 0.0;
20        _goal4_or << 0.0, 0.0, 0.0, 1.0;
21
```

Then in the same file we defined the listeners' functions and the debug print (due to space we show just one listener in the following photo):

```
 83   // Goal listener functions (2b point)
 84
 85   void TF_NAV::goal_listener_1() {
 86       ros::Rate r( 1 );
 87       tf::TransformListener listener;
 88       tf::StampedTransform transform;
 89
 90       while ( ros::ok() )
 91       {
 92           try
 93           {
 94               listener.waitForTransform( "map", "goal1", ros::Time( 0 ), ros::Duration( 10.0 ) );
 95               listener.lookupTransform( "map", "goal1", ros::Time( 0 ), transform );
 96           }
 97           catch( tf::TransformException &ex )
 98           {
 99               ROS_ERROR("%s", ex.what());
100               r.sleep();
101               continue;
102           }
103
104           _goal1_pos << transform.getOrigin().x(), transform.getOrigin().y(), transform.getOrigin().z();
105           _goal1_or << transform.getRotation().w(),  transform.getRotation().x(), transform.getRotation().y(), transform.getRotation().z();
106
107           // // Debug Print
108           // ROS_INFO("Goal_1 Position: %f %f %f", _goal1_pos[0], _goal1_pos[1], _goal1_pos[2]);
109           // ROS_INFO("Goal_1 Orientation: %f %f %f %f", _goal1_or[0], _goal1_or[1], _goal1_or[2], _goal1_or[3]);
110
111           r.sleep();
112       }
113   }
114
```

Used commands
  $ cd catkin_ws
  $ roslaunch fra2mo_2dnav fra2mo_nav_bringup.launch
  $ rosrun fra2mo_2dnav tf_nav

(c) Using move_base, send goals to the mobile platform in a given order. Go to the next one once the robot has arrived at the current goal. The order of the explored goals must be *Goal_3 → Goal_4 → Goal_2 → Goal_1*. Use the Action Client communication protocol to get the feedback from move_base. Record a bagfile of the executed robot trajectory and plot it as a result.

In this point we had to send the goals to the robot in a certain order, in particular the sequence to follow is 3-4-2-1. To do that we went inside the file *"tf_nav.cpp"* and modified the function send_goal.
This function allows us to select between two different choices: send goal from TF or send home position goal. The first option that has been implemented brings the mobile robot to the desired points specified by the goals while the second one brings the robot to the home configuration. In the following picture we show the function that send the robot to the goal 3:

```cpp
350        if ( cmd == 1) {
351            MoveBaseClient ac("move_base", true);
352
353            while(!ac.waitForServer(ros::Duration(5.0))){
354            ROS_INFO("Waiting for the move_base action server to come up");
355            }
356            goal.target_pose.header.frame_id = "map";
357            goal.target_pose.header.stamp = ros::Time::now();
358
359            // First goal of the sequence is Goal_3
360            goal.target_pose.pose.position.x = _goal3_pos[0];
361            goal.target_pose.pose.position.y = _goal3_pos[1];
362            goal.target_pose.pose.position.z = _goal3_pos[2];
363
364            goal.target_pose.pose.orientation.w = _goal3_or[0];
365            goal.target_pose.pose.orientation.x = _goal3_or[1];
366            goal.target_pose.pose.orientation.y = _goal3_or[2];
367            goal.target_pose.pose.orientation.z = _goal3_or[3];
368
369            ROS_INFO("Sending goal 3");
370            ac.sendGoal(goal);
371
372            ac.waitForResult();
373
374            if(ac.getState() == actionlib::SimpleClientGoalState::SUCCEEDED)
375            ROS_INFO("The mobile robot is in the pose of the Goal_3");
376
377            else
378                ROS_INFO("The base failed to move for some reason");
```

We have created a bag file called: *bag_2c.bag* and the results obtained on Matlab will be shown below. The first plot is the one relating to the quaternion and the second to the pose of the robot.
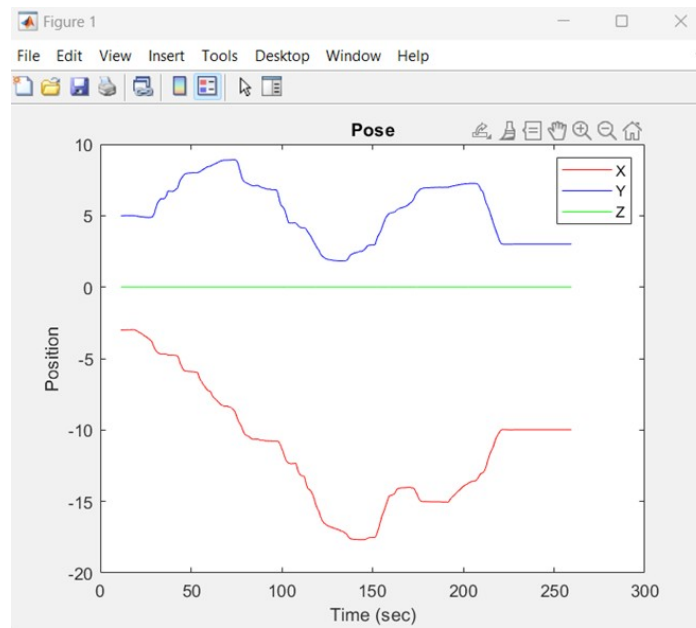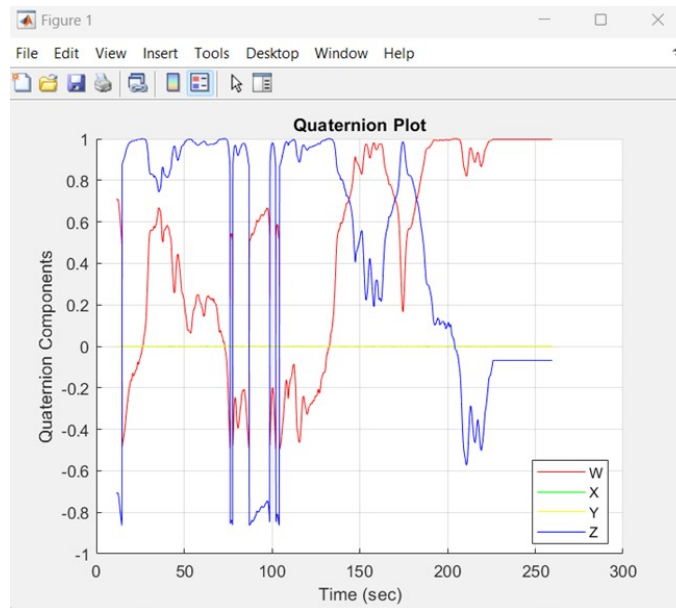
The Matlab file used to visualize the plots is shown below.

```matlab
untitled.m    +
1    close,clear,clc
2
3    bag=rosbag('bag_2c.bag');
4    topic = '/fra2mo/pose';
5    pose= readMessages(select(bag, 'Topic', topic));
6
7    timestamps = cellfun(@(msg) double(msg.Header.Stamp.Sec) + double(msg.Header.Stamp.Nsec)/1e9, pose);
8    x = cellfun(@(msg) msg.Pose.Position.X, pose);
9    y = cellfun(@(msg) msg.Pose.Position.Y, pose);
10   z = cellfun(@(msg) msg.Pose.Position.Z, pose);
11
12   figure;
13   plot(timestamps, x,'r',timestamps, y,'b', timestamps, z,'g');
14   legend('X', 'Y', 'Z');
15   xlabel('Time (sec)');
16   ylabel('Position');
17   title('Pose');
18
19   % Access the orientation (quaternion) data in the messages and plot
20   orientation = cellfun(@(msg) msg.Pose.Orientation, pose);
21
22   % Extract quaternion components
23   quatW = cellfun(@(msg) msg.Pose.Orientation.W, pose);
24   quatX = cellfun(@(msg) msg.Pose.Orientation.X, pose);
25   quatY = cellfun(@(msg) msg.Pose.Orientation.Y, pose);
26   quatZ = cellfun(@(msg) msg.Pose.Orientation.Z, pose);
27
28   % Data plot
29   figure;
30   grid on;
31
32   % We have set the color order
33   colorOrder = [1 0 0; 0 1 0; 1 1 0; 0 0 1]; % Red, Green, Yellow, Blue
34   set(gca, 'ColorOrder', colorOrder, 'NextPlot', 'replacechildren');
35
36   % We plotted the data with the specified colors
37   plot(timestamps, quatW, timestamps, quatX, timestamps, quatY, timestamps, quatZ);
38
39   % We added legend, labels and title
40   legend('W', 'X', 'Y', 'Z', 'Location', 'Best');
41   xlabel('Time (sec)');
42   ylabel('Quaternion Components');
43   title('Quaternion Plot');
44
```

3. **Map the environment tuning the navigation stack's parameters**

   a) Modify, add, remove, or change pose, the previous goals to get a complete map of the environment

In this point we have created new goals to get a complete view of the map. What was made is shown in the following picture:

```
47    <!--Static tf publisher for goal1-->
48    <!-- Define publisher tf nodes for four goals (2a point) -->
49    <node pkg="tf" type="static_transform_publisher" name="goal_1_pub" args="-10 3 0 0 0 1 map goal1 100" />
50    <!-- <node pkg="tf" type="static_transform_publisher" name="goal_2_pub" args="-15 7 0 0 0 0.25882 0.96593 map goal2 100" />  This is for 2a point -->
51    <node pkg="tf" type="static_transform_publisher" name="goal_2_pub" args="-17.305 9.507 0 0 0 0.25882 0.96593 map goal2 100" /> <!-- This is for 3a point -->
52    <node pkg="tf" type="static_transform_publisher" name="goal_3_pub" args="-6 8 0 0 0 1 0 map goal3 100" />
53    <node pkg="tf" type="static_transform_publisher" name="goal_4_pub" args="-17.5 3 0 0 0 0.6088 0.7933 map goal4 100" />
54
55    <!-- Add publisher tf nodes for four goals in order to map all the environment (3a point) -->
56    <node pkg="tf" type="static_transform_publisher" name="goal_5_pub" args="-6.206 0.5 0 0 0 1 0 map goal5 100" />
57    <node pkg="tf" type="static_transform_publisher" name="goal_6_pub" args="-6.1293 5.1205 0 0 0 1 0 map goal6 100" />
58    <node pkg="tf" type="static_transform_publisher" name="goal_7_pub" args="-4.115 9.367 0 0 0 1 0 map goal7 100" />
59    <node pkg="tf" type="static_transform_publisher" name="goal_8_pub" args="-0.709 3.19 0 0 0 1 0 map goal8 100" />
```

To have a better idea of what was made we show the gazebo map:


11
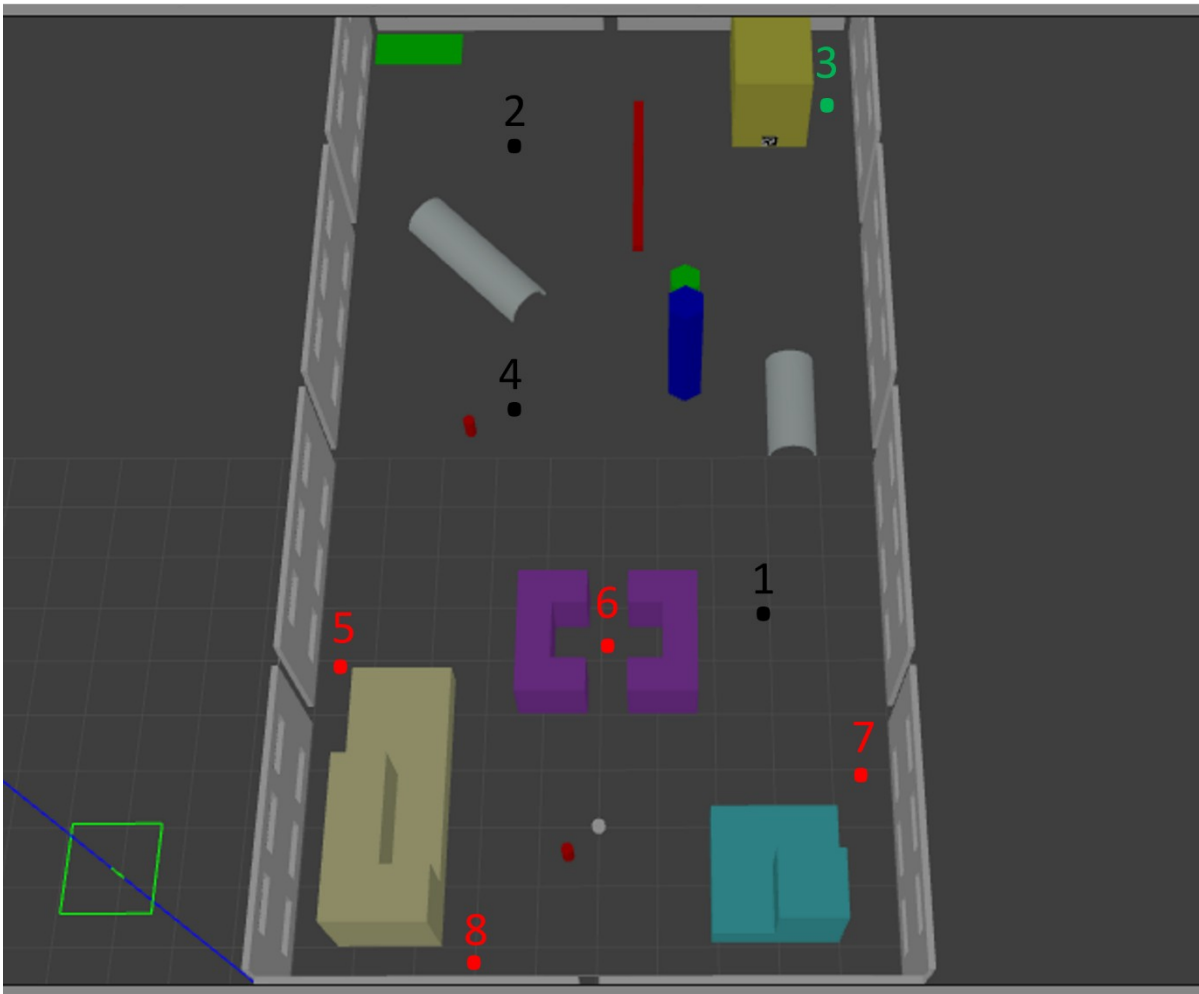
There is a note: the points in the photo above are not the goals but the sequence of motions. In particular, we added a goal through the two purple obstacle, one behind the azure obstacle, one behind the sand-colored one and the last one behind the obstacle 9 where it is located the aruco.

Note that we have also modified the code line relative to the goal 2 (point 3) in such way the robot could see better behind the obstacle 9. The last things to do were go inside the *"tf_nav.cpp"* and add orientation, position, send goals and listeners of the new goals, and finally we added all those stuffs inside the file *"tf_nav.h"*.

Used commands
  $ cd catkin_ws
  $ roslaunch fra2mo_2dnav fra2mo_nav_bringup.launch
  $ rosrun fra2mo_2dnav tf_nav
To prove that all the map is scanned we provide also the rviz view:
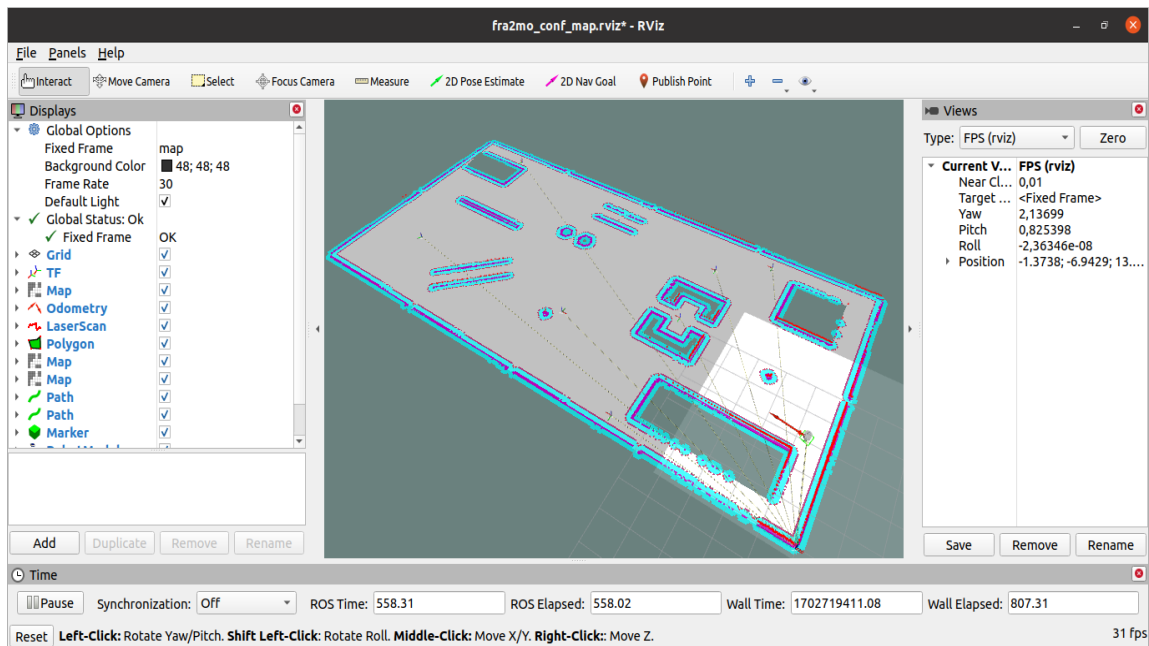
b) Change the parameters of the planner and move_base (try at least 4 different configurations) and comment on the results you get in terms of robot trajectories. The parameters that need to be changed are:
- In file teb_locl_planner_params.yaml: tune parameters related to the section about trajectory, robot, and obstacles.
- In file local_costmap_params.yaml and global_costmap_params.yaml: change dimensions' values and update costmaps' frequency.
- In file costmap_common_params.yaml: tune parameters related to the obstacle and raytrace ranges and footprint coherently as done in planner parameters.

In this point we had to change some parameters of the planner and the move_base to test other configurations, this can be done inside the folder config of the package *"fra2mo_2dnav"*.
Firstly we had modified the file *"teb_locl_planner_params.yaml"* , and in particular the minimum distance to the obstacle that we set to 0.2. With 0.2 the robot could reach all the points unless the one situated through the two purple obstacles, nevertheless it could scan what is inside.

```
40   # Obstacles
41
42   min_obstacle_dist: 0.1 # 0.2 # For 3b point we modified the minimum distance to obstacle at 0.2 value
43   include_costmap_obstacles: True
44   costmap_obstacles_behind_robot_dist: 1.0
45   obstacle_poses_affected: 20
46   costmap_converter_plugin: ""
47   costmap_converter_spin_thread: True
48   costmap_converter_rate: 5
49   inflation_dist: 0.1
50   include_dynamic_obstacles: false
```

Used commands
  $ cd catkin_ws
  $ roslaunch fra2mo_2dnav fra2mo_nav_bringup.launch
  $ rosrun fra2mo_2dnav tf_nav

In the following picture it has been shown the map visualization in rviz with this change:
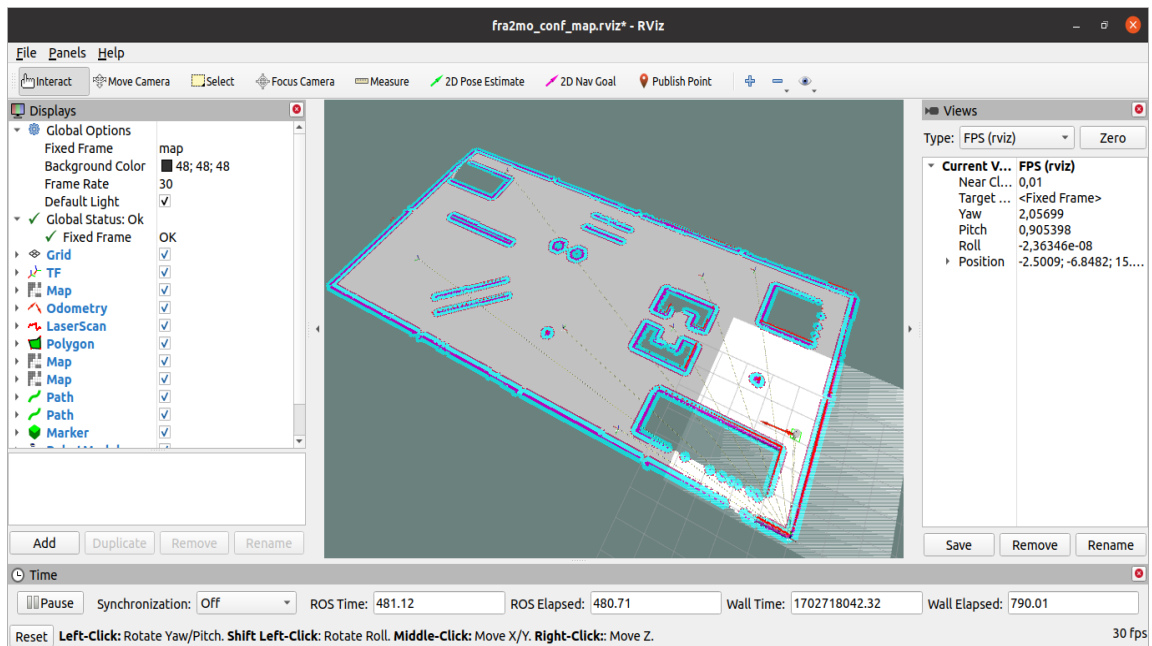
And then the same result but with the distance of 0.1 is shown in the first figure.

In theory, even if we cannot appreciate it with the naked eyes, the minimum distance of the robot to the obstacle has been increased.

We have also tried to set the minimum distance equal to 0.3, but in this case the robot could not generate the trajectories to reach the points.

The second point demanded to change dimensions' values and update costmaps' frequency of the files *"local_costmap_params.yaml"* and *"global_costmap_params.yaml"*. So, in first, we had modified the update frequency and the publish frequency of the local_costmap; in particular, we set the first one equal to 6.0 and the second one equal to 11.0.

```
1  local_costmap:
2    global_frame: map
3    robot_base_frame: base_footprint
4    update_frequency: 5.0 # 6.0 # For 3b point we modified the update_frequency at 6.0 value
5    publish_frequency: 10.0 # 11.0 # For 3b point we modified the publish_frequency at 11.0 value
6    static_map: false
```

The result of those changes is that the robot could not reach all the points of the map, in particular it skipped goal 4 and 8. The result is shown in following figure.
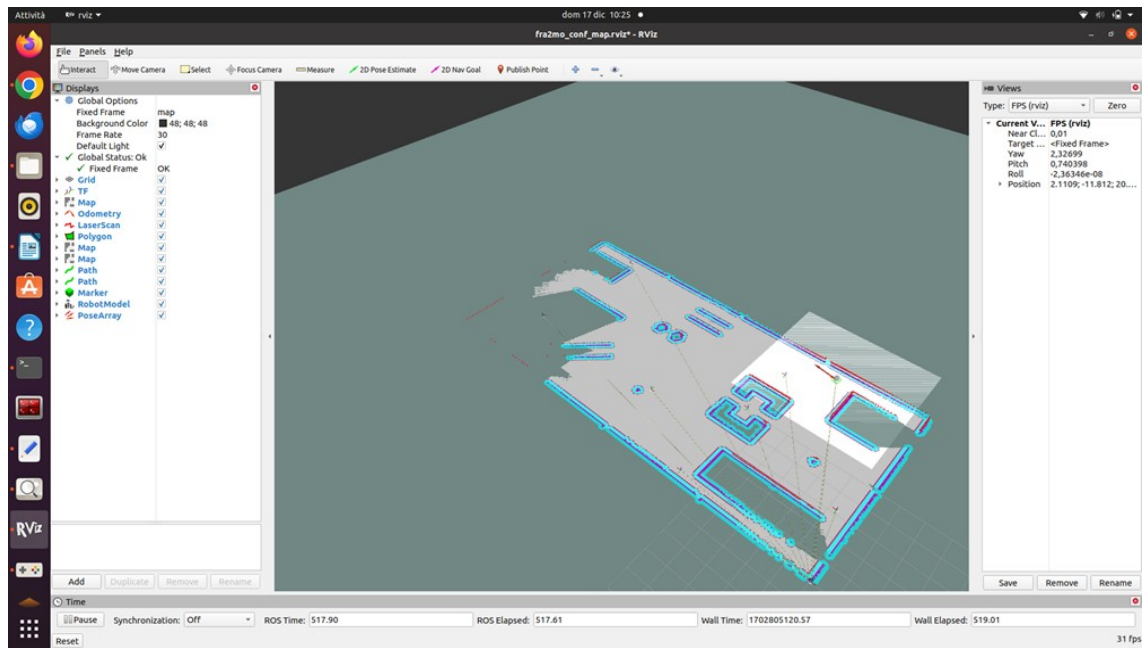
In second place we had modified the width (the number of cells along the x axis) and the height (the number of cells along the y axis) of the map representation setting both the parameters equal to 15.0.

```
1   local_costmap:
2     global_frame: map
3     robot_base_frame: base_footprint
4     update_frequency: 5.0 # 6.0 # For 3b point we modified the update_frequency at 6.0 value
5     publish_frequency: 10.0 # 11.0 # For 3b point we modified the publish_frequency at 11.0 value
6     static_map: false
7     rolling_window: true
8     width: 7.0 # 15.0 # For 3b point we modified the width at 15.0 value
9     height: 7.0 # 15.0 # For 3b point we modified the height at 15.0 value
10    resolution: 0.03
```

As expected, we can see a more detailed view of the map:
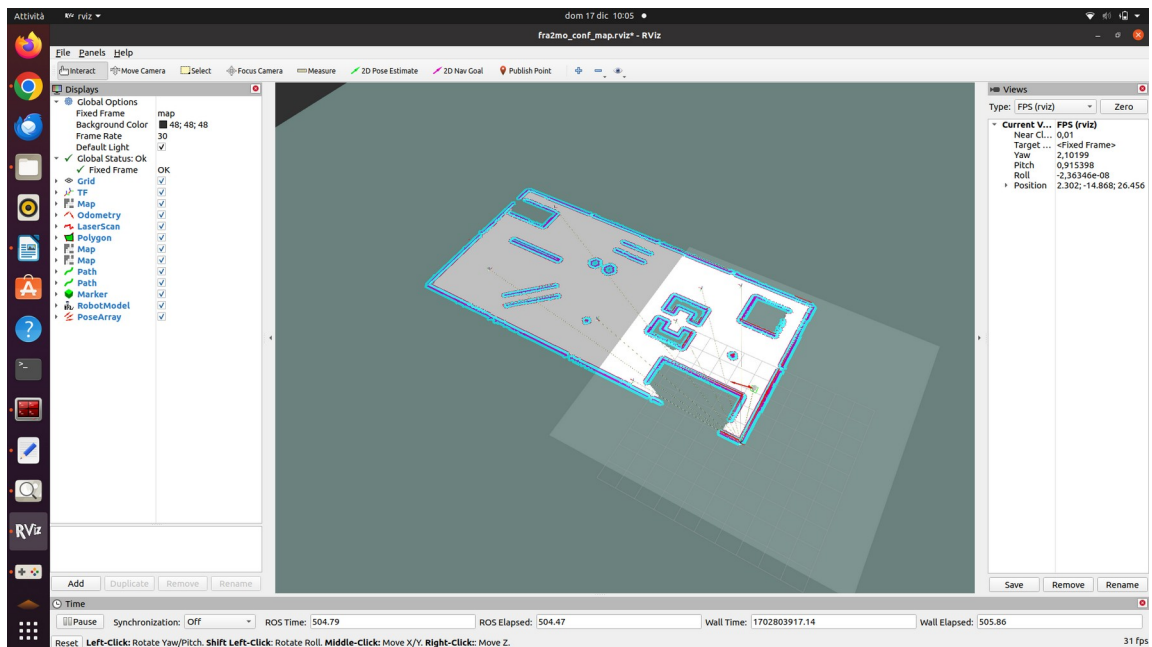
In global_costmap we have modified only the width and the height of the map.

```
1   global_costmap:
2     global_frame: map
3     robot_base_frame: base_footprint
4     update_frequency: 5.0
5     publish_frequency: 2.0
6     #always_send_full_costmap: false #default is false
7     rolling_window: false
8     resolution: 0.05
9     width: 30 # 60 # For 3b point we modified the width at 60 value
10    height: 30 # 60 # For 3b point we modified the height at 60 value
11    origin_x: -15
12    origin_y: -15
13
```

The results show us that the robot could not scan the entire map:
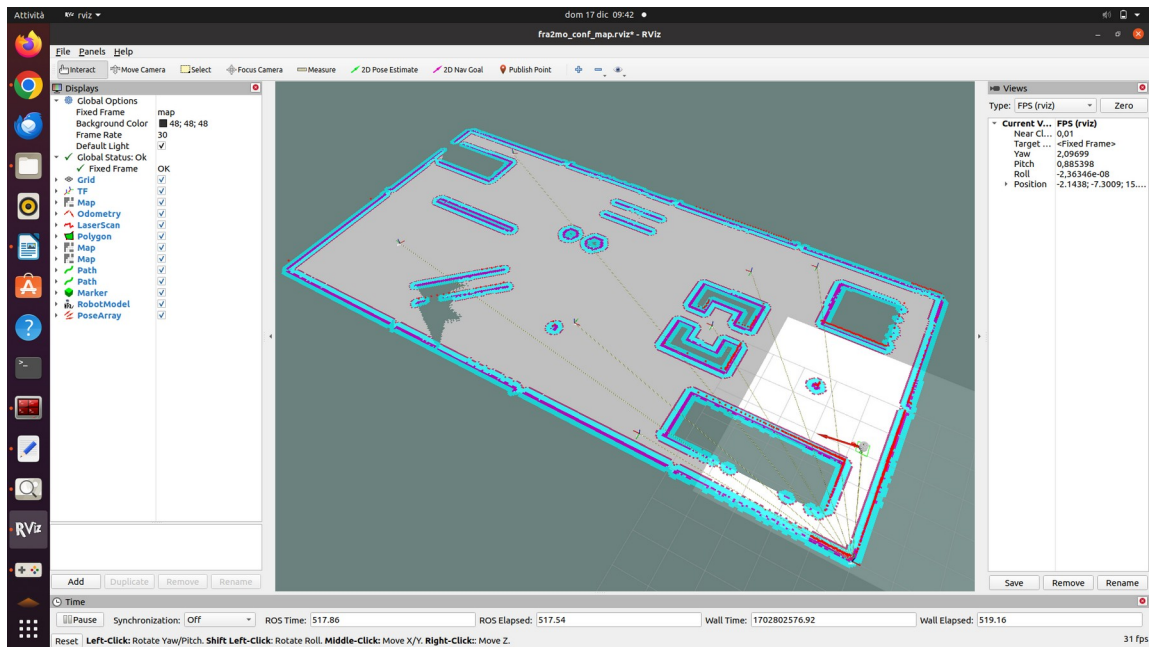
The last thing is to tune the parameters related to the obstacle and raytrace ranges and footprint coherently as done in planner parameters in the file *"costmap_common_params.yaml"*.
Increasing the value of the obstacle_range will change the maximum distance of the sensor scanning.

```
1  publish_voxel_map: false
2  transform_tolerance: 0.5
3  meter_scoring: true
4  obstacle_range: 7.0 # maximum range sensor reading that will result in an obstacle being put into the costmap # 8.0 # For 3b point we modified the obstacle_range at 8.0 value
5  raytrace_range: 8.0 # range to which we will raytrace freespace given a sensor reading # 9.0 # For 3b point we modified the raytarce_range at 9.0 value
6  footprint: [[0.15, -0.15], # 0.20, -0.20 # For 3b point we modified the first raw at 0.20, -0.20 value
7              [-0.15, -0.15], # -0.20, -0.20 # For 3b point we modified the first raw at -0.20, -0.20 value
8              [-0.15, 0.20], # -0.20, 0.25 # For 3b point we modified the first raw at -0.20, 0.25 value
9              [0.15, 0.15]] # 0.20, 0.20 # For 3b point we modified the first raw at 0.20, 0.20 value
```

In first we have set the obstacle range equal to 8.0 but the robot was not able to navigate in several points.
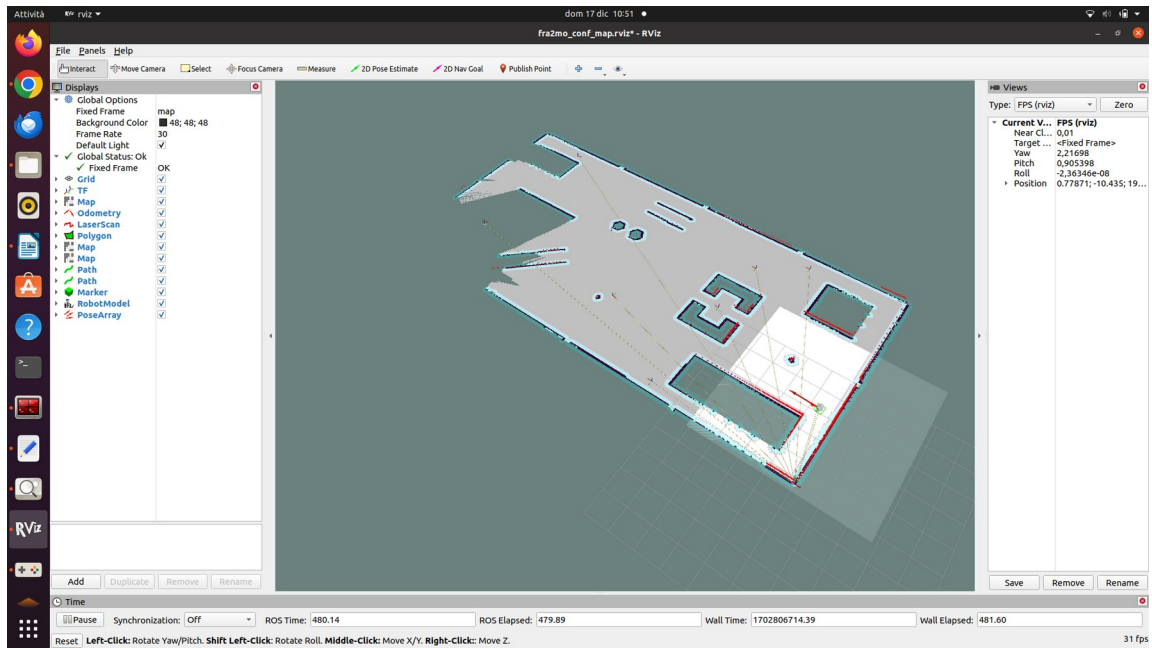
Then we have raised the raytrace range to 9.0 from 8.0. Increasing the raytrace range may let the robot to have a more extended perception of the space.
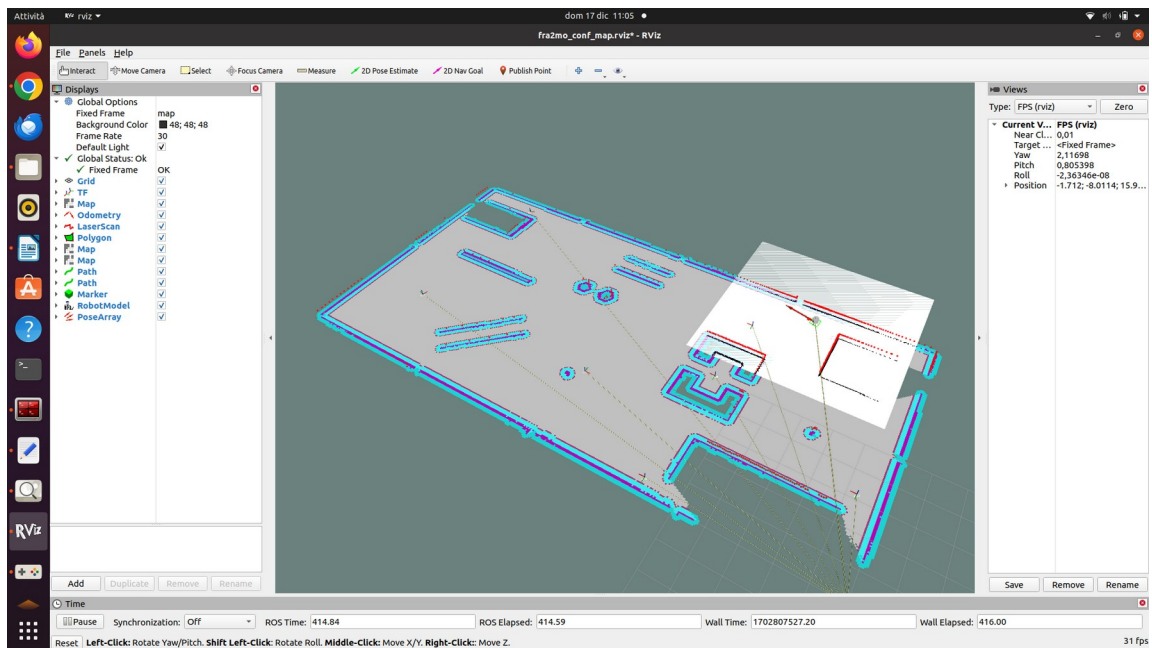
The robot in the simulation shown above was not able to reach the goal 5,6 and 8.
As last thing we have modified the footprint of the robot but as we can see from the following picture, he could satisfy only one goal that is the sixth.
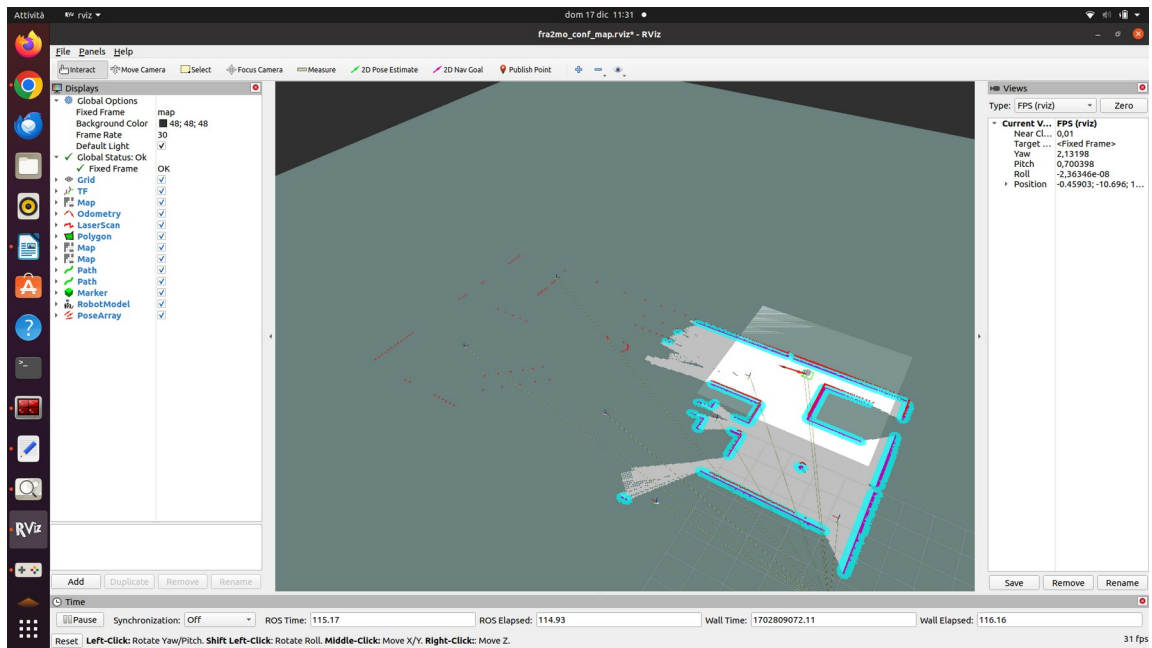
4. **Vision-based navigation of the mobile platform**

   a) Run ArUco ROS node using the robot camera: bring up the camera model and uncomment it in that fra2mo.xacro file of the mobile robot description rl_fra2mo_description. Remember to install the camera description pkg: sudo apt-get install ros-<DISTRO>-realsense2-description

To accomplish this point we went inside the file *"fra2mo.xacro"* in the folder *"urdf"* of the package *"rl_fra2mo_description"* and we have uncommented the code lines regarding the camera, as it is shown in the figure below:

```xml
1   <?xml version="1.0"?>
2
3   <robot name="fra2mo" xmlns:xacro="http://ros.org/wiki/xacro">
4     <xacro:include filename="$(find rl_fra2mo_description)/urdf/fra2mo_base_macro.xacro" />
5     <xacro:include filename="$(find rl_fra2mo_description)/urdf/lidar_gazebo_macro.xacro" />
6     <xacro:include filename="$(find rl_fra2mo_description)/urdf/d435_gazebo_macro.xacro" />
7
8
9     <xacro:property name="LIDAR" value="True"/>
10    <xacro:property name="DEPTH" value="True"/>
11
12    <!-- Diff Drive Robot Base  -->
13    <xacro:fra2mo_base />
14
15    <!-- LIDAR Sensor -->
16    <xacro:if value="${LIDAR}" >
17      <xacro:lidar_gazebo_sensor parent="lidar_link" />
18    </xacro:if>
19    <!-- Uncomment if you want to add also a D435 camera -->
20    <!-- RGBD Sensor -->
21    <xacro:if value="${DEPTH}" >
22      <xacro:d435_gazebo_sensor parent="d435_link" />
23    </xacro:if>
24
25  </robot>
```

b) Implement a 2D navigation task following this logic
- Send the robot in the proximity of obstacle 9.
- Make the robot look for the ArUco marker. Once detected, retrieve its pose with respect to the map frame.
- Set the following pose (relative to the ArUco marker pose) as next goal for the robot

$$x = x_m + 1, \quad y = y_m,$$

where x_m, y_m are the marker coordinates.

To implement the first point of this task, it has been added to the file *"tf_nav.cpp"* another choice (command "3") when the node is running.

To bring the robot in front of the obstacle 9 we first have moved the robot to an intermediate point *"goal_4b_1"* and then to the final goal *"goal_4b"*. Inside the files *"tf_nav.h"* and *"tf_nav.cpp"* we have defined the poses of the goals and the listeners and inside the file *"spawn_fra2mo_gazebo. launch"*, those have been published, as shown in figure.

```
61  <!-- Add publisher tf node for goals in order to send the mobile robot in front of the Aruco marker (4b point) -->
62  <node pkg="tf" type="static_transform_publisher" name="goal_4b_pub" args="-15 7.8 0 0 0 1 0 map goal4b 100" />
63  <node pkg="tf" type="static_transform_publisher" name="goal_4b_1_pub" args="-7 8 0 0 0 1 0 map goal4b_1 100" />
```

The second point demanded to make the robot look for the ArUco marker, and once detected, the robot should retrieve its pose with respect to the map frame. To do that, we've added a vector *"aruco_pose"* that contains aruco position and orientation to the *"tf_nav.cpp"* file in *"fra2mo_2dnav"* package.

Then, we've added a *"aruco_pose_sub"* subscriber to the *"/aruco_single/pose"* topic and so recall *"arucoPoseCallback"* callback that writes on the aruco_pose and switches *"aruco_pose_available"* boolean parameter on true.

```
void arucoPoseCallback(const geometry_msgs::PoseStamped & msg)
{
    aruco_pose_available = true;
    aruco_pose.clear();
    aruco_pose.push_back(msg.pose.position.x);
    aruco_pose.push_back(msg.pose.position.y);
    aruco_pose.push_back(msg.pose.position.z);
    aruco_pose.push_back(msg.pose.orientation.x);
    aruco_pose.push_back(msg.pose.orientation.y);
    aruco_pose.push_back(msg.pose.orientation.z);
    aruco_pose.push_back(msg.pose.orientation.w);

}
```

In order to retrieve the marker pose in the map frame, we have added the *"tf_listener_aruco"* function. First of all, a listener has been defined to retrieve the camera_depth_optical_frame with respect to map frame. It has been chosen this frame because this one is where visual information are referred to. Once the marker has been detected, its pose in the camera frame can be transformed in the map frame.

21

This reasoning has been implemented as shown below:

```cpp
void TF_NAV::tf_listener_aruco() {
    ros::Rate r( 5 );
    tf::TransformListener listener;
    tf::StampedTransform transform;

    while ( ros::ok() ) {
        try {
            listener.waitForTransform( "map", "camera_depth_optical_frame", ros::Time(0), ros::Duration(10.0) );
            listener.lookupTransform( "map", "camera_depth_optical_frame", ros::Time(0), transform );
        }
        catch( tf::TransformException &ex ) {
            ROS_ERROR("%s", ex.what());
            r.sleep();
            continue;
        }

        _camera_pos << transform.getOrigin().x(), transform.getOrigin().y(), transform.getOrigin().z();
        _camera_or << transform.getRotation().w(),  transform.getRotation().x(), transform.getRotation().y(), transform.getRotation().z();

        if(aruco_pose_available) {
            KDL::Frame cam_T_object(KDL::Rotation::Quaternion(aruco_pose[3], aruco_pose[4], aruco_pose[5], aruco_pose[6]), KDL::Vector(aruco_pose[0], aruco_pose[1], aruco_pose[2]));
            KDL::Frame map_T_cam(KDL::Rotation::Quaternion(_camera_or[1], _camera_or[2], _camera_or[3], _camera_or[0]), KDL::Vector(_camera_pos[0], _camera_pos[1], _camera_pos[2]));
            KDL::Frame map_T_object(KDL::Rotation::Quaternion(_aruco_or[1], _aruco_or[2], _aruco_or[3], _aruco_or[0]), KDL::Vector(_aruco_pos[0], _aruco_pos[1], _aruco_pos[2]));

            map_T_object.p = map_T_cam.p + map_T_cam.M*cam_T_object.p;
            _aruco_pos = toEigen(map_T_object.p);

            map_T_object.M = map_T_cam.M*cam_T_object.M;
            double x,y,z,w;
            map_T_object.M.GetQuaternion(x,y,z,w);
            _aruco_or << x, y, z, w;

            std::cout<<"Aruco position wrt map frame:"<<std::endl<< _aruco_pos <<std::endl;
            //std::cout<<"Aruco quaternion wrt map frame:"<< _aruco_or <<std::endl;
            std::cout<<"Camera position wrt map frame:"<<std::endl<< toEigen(map_T_cam.p) <<std::endl;
            //std::cout<<"Aruco position wrt camera frame:"<< toEigen(cam_T_object.p) <<std::endl;
        }

        r.sleep();
    }
}
```
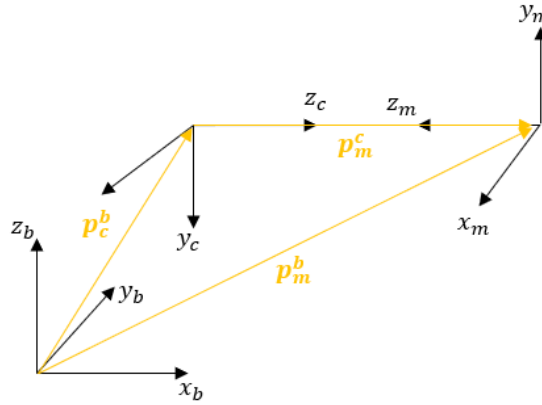
Frames configuration:

$$p_m^b = p_c^b + R_c^b p_m^c$$

The last step is to send mobile robot in front of the Aruco with a pose relative at the aruco pose, and in particular it must be:

$y = y_m, x = x_m + 1$

This task has been implemented in the node *"tf_nav.cpp"* in the *"send_goal"* function in this way: when the mobile robot arrived at the *"goal4b"* goal, on the terminal is asked to insert the command "4" to bring the robot to the pose that has been defined above.
This task was implemented as follows:

- Knowing marker pose, the goal orientation can be assigned in a fixed way as x = 0, y = 0, z = 1, w = 0 (properly commented in the code below).

Alternatively, a more robust solution takes care also of the aruco orientation. First of all, to define the desired base_footprint frame, the Aruco Marker frame has been suitably rotated.
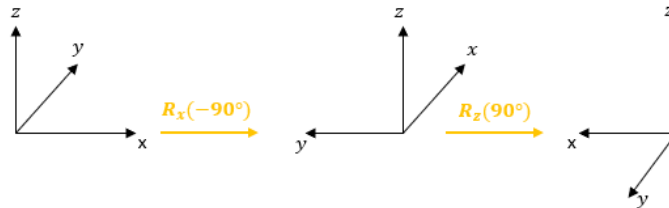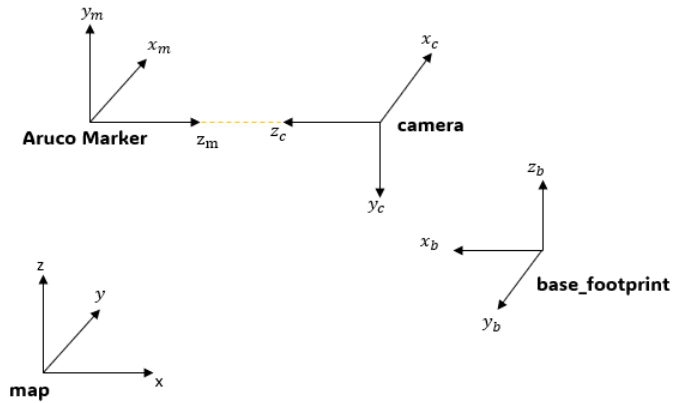
Due to approximation and measurement errors, it is not possible to directly extract quaternion refer-ence, since it would end up having roll and pitch angles different from zero. To overcome this problem, a new rotation matrix has been defined retrieving its yaw angle, while setting roll and pitch to zero. At this point, the extracted quaternion reference is feasible for the robot. As shown below:

```
805        // When the mobile robot arrived in the goal4b it moves at this pose (x = x_m + 1 and y = y_m) when the user insert the command "4"
806
807        std::cout<<"\nInsert 4 to send Aruco goal"<<std::endl;
808        std::cin>>a_cmd;
809        if ( a_cmd == 4) {
810            MoveBaseClient ac_a("move_base", true);
811            while(!ac_a.waitForServer(ros::Duration(5.0))){
812                ROS_INFO("Waiting for the move_base action server to come up");
813            }
814            goal.target_pose.header.frame_id = "map";
815            goal.target_pose.header.stamp = ros::Time::now();
816
817            goal.target_pose.pose.position.x = _aruco_pos[0] + 1;
818            goal.target_pose.pose.position.y = _aruco_pos[1];
819            goal.target_pose.pose.position.z = 0.0;
820  /*
821            goal.target_pose.pose.orientation.w = 0.0;
822            goal.target_pose.pose.orientation.x = 0.0;
823            goal.target_pose.pose.orientation.y = 0.0;
824            goal.target_pose.pose.orientation.z = 1.0;
825  */
826            KDL::Rotation Ar_rot=KDL::Rotation::Quaternion(_aruco_or[1], _aruco_or[2], _aruco_or[3], _aruco_or[0]);
827            KDL::Rotation Rb_des=Ar_rot*KDL::Rotation::RotX(-1.5708)*KDL::Rotation::RotZ(1.5708);
828            double R,P,Y,x,y,z,w;
829            Rb_des.GetRPY(R,P,Y);
830            Rb_des=KDL::Rotation::RPY(0,0,Y);
831            Rb_des.GetQuaternion(x,y,z,w);
832
833            goal.target_pose.pose.orientation.w = w;
834            goal.target_pose.pose.orientation.x = x;
835            goal.target_pose.pose.orientation.y = y;
836            goal.target_pose.pose.orientation.z = z;
837
838            ROS_INFO("Sending goal aruco");
839            ac_a.sendGoal(goal);
840
841            ac_a.waitForResult();
842
843            if(ac_a.getState() == actionlib::SimpleClientGoalState::SUCCEEDED)
844                ROS_INFO("The mobile robot arrived in the TF aruco goal");
845            else
```

Used commands
  $ cd catkin_ws
  $ roslaunch fra2mo_2dnav fra2mo_nav_bringup.launch
  $ rosrun fra2mo_2dnav tf_nav
  $ roslaunch aruco_ros usb_cam_aruco.launch camera:=/depth_camera/depth_camera markerId:=115
  $ rqt_image_view

In the folder Simulations within github repository there are loaded the videos of the tests of this 4b point that has been implemented.

c) Publish the ArUco pose as TF following the example at [this link](#).

In order to implement this 4c point we added a new node "*tf_nav_4c.cpp*" in the package "*fra2mo_2dnav*" and so we added executable of this node in "*CMakeLists.txt*" file.
At the end of "*tf_listener_aruco*" function, the function "*position_aruco_pub*" is called and the latter publishes the aruco pose via "*_position_aruco_pub*" publisher on the topic "*aruco_frame/pose*".

```
10    _position_aruco_pub = _nh.advertise<geometry_msgs::PoseStamped>( "aruco_frame/pose", 1 );
```

We created a subscriber "*aruco_pose_sub_broadc*" that subscribes to "*aruco_frame/pose*" topic and it calls the "*poseCallback*" function that is shown in following figure.

```
910    ros::Subscriber aruco_pose_sub_broadc = n.subscribe("/aruco_frame/pose", 1, poseCallback);
```

```
67   // poseCallback function for 4c point (we got it from the website adviced in the pdf)
68   void poseCallback(const geometry_msgs::PoseStamped & msg) {
69       static tf::TransformBroadcaster br;
70       tf::Transform transform;
71       transform.setOrigin( tf::Vector3(msg.pose.position.x,msg.pose.position.y,msg.pose.position.z));
72       tf::Quaternion q(msg.pose.orientation.x,msg.pose.orientation.y,msg.pose.orientation.z,msg.pose.orientation.w);
73       transform.setRotation(q);
74       br.sendTransform(tf::StampedTransform(transform, ros::Time::now(), "map", "aruco_frame"));
75   }
```

Used commands
  $ cd catkin_ws
  $ roslaunch fra2mo_2dnav fra2mo_nav_bringup.launch
  $ rosrun fra2mo_2dnav tf_nav_4c
  $ roslaunch aruco_ros usb_cam_aruco.launch camera:=/depth_camera/depth_camera markerId:=115
  $ rqt_image_view

In the folder Simulations within github repository there are loaded the videos of the tests of this 4c point that has been implemented.