Report - Homework 4

Control a mobile robot to follow a trajectory

Emanuele Cuzzocrea P38000196

Group:
Emanuele Cuzzocrea
Silvia Leo
Vito Daniele Perfetta
Giulia Gelsomina Romano

The code of this project and the related video can be found in my public github repository:
`https://github.com/EmanueleCuzzocrea/Homework4.git`

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

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

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

spawn_fra2mo_gazebo.launch

```
...
  <arg name="x_pos" default="-3.0"/>
  <arg name="y_pos" default="5.0"/>
  <arg name="z_pos" default="0.1"/>
  <arg name="yaw" default="-1.57"/>
...
  <node name="urdf_spawner" pkg="gazebo_ros" type="spawn_model" respawn=
     "false" output="screen"
        args="-urdf -model fra2mo -x $(arg x_pos) -y $(arg y_pos) -z $(
           arg z_pos) -Y $(arg yaw) -param robot_description"/>
...
```

Used command

```
$ roslaunch fra2mo_2dnav fra2mo_nav_bringup.launch
```

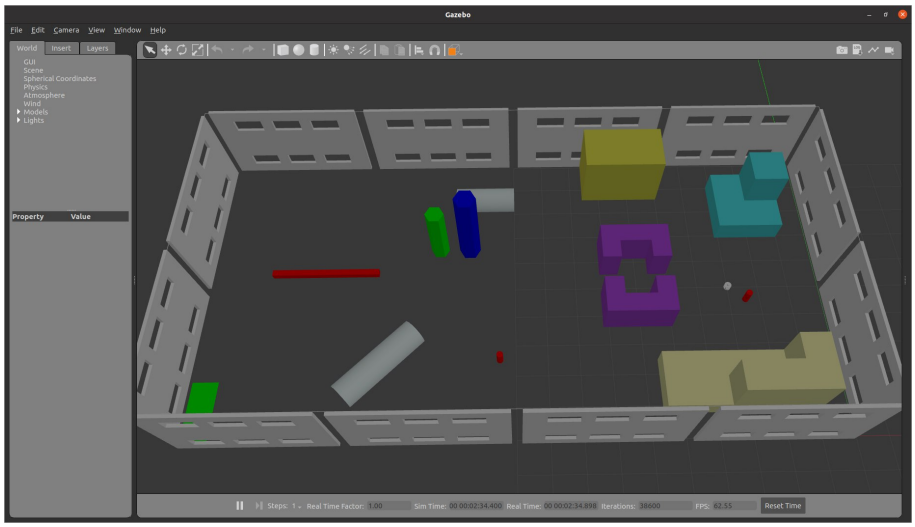

Figure 1: Mobile robot in Gazebo world

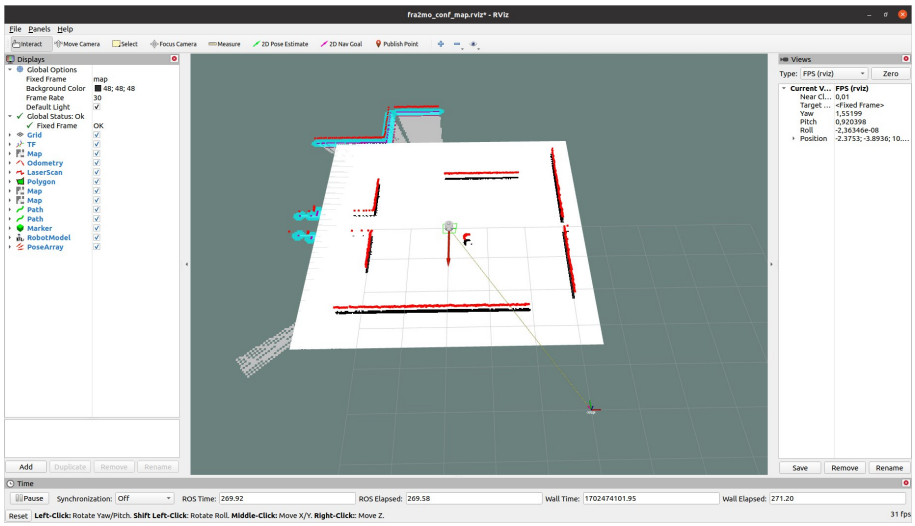

Figure 2: Mobile robot in Rviz

(b) Modify the world file of rl_racefield moving the obstacle 9 in position:

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

rl_race_field.world

```
...
    <include>
      <name>obstacle_09</name>
      <pose> -17 9 0.1 0 0 3.14159</pose>
      <uri>model://obstacle_09</uri>
    </include>
...
```

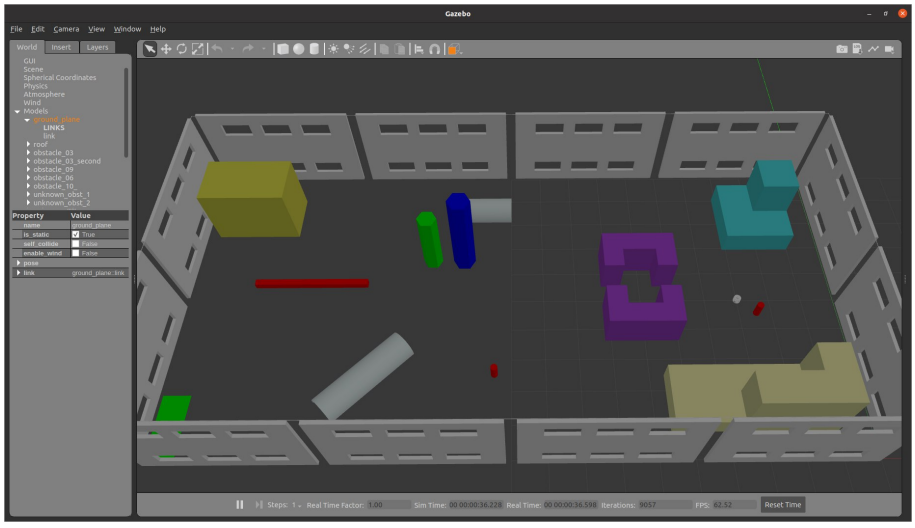

Figure 3: Mobile robot in modified Gazebo world

(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.

Using the given link, the ArUco marker number 115 .svg has been downloaded choosing as Dictionary "Original ArUco", Marker ID "115", and Marker size mm "100". Then, it has been converted in a .png, and included in the following model

marker_115.sdf

```
<?xml version="1.0" ?>
<sdf version="1.4">
  <model name="marker_115">
    <static>true</static>
    <link name="marker">
      <gravity>false</gravity>

      <visual name="front_visual">
        <pose>0 0 0.0005 0 0 0</pose>
        <geometry>
          <box>
           <size>0.10 0.10 0.001</size>
         </box>
        </geometry>
        <material>
          <script>
           <uri>model://marker_115/material/scripts</uri>
            <uri>model://marker_115/material/textures</uri>
            <name>marker_115</name>
          </script>
        </material>
      </visual>

      <!-- Hide the marker from the back -->
      <visual name="rear_visual">
        <pose>0 0 -0.0005 0 0 0</pose>
        <geometry>
          <box>
            <size>0.10 0.10 0.001</size>
          </box>
        </geometry>
      </visual>

      <collision name="collision">
        <geometry>
          <box>
            <size>0.10 0.10 0.001</size>
          </box>
        </geometry>
      </collision>
    </link>
  </model>
</sdf>
```

The previous model has been included in the world as follows

rl_racefield.world

```
...
    <!-- Aruco marker 115 -->
    <include>
```

```
      <name>tool_0_tag</name>
      <uri>model://marker_115</uri>
      <pose>-17 8.25 0.2 3.14 -1.57 0</pose>
    </include>
...
```

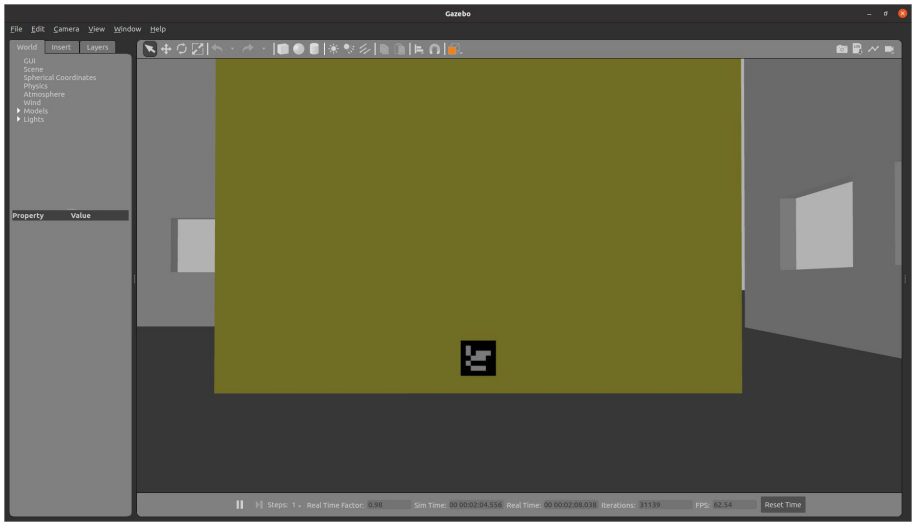

Figure 4: Aruco marker 115 on the obstacle 9

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: $x = -10 \quad y = 3 \quad yaw = 0$ deg
- Goal_2: $x = -15 \quad y = 7 \quad yaw = 30$ deg
- Goal_3: $x = -6 \quad y = 8 \quad yaw = 180$ deg
- Goal_4: $x = -17.5 \quad y = 3 \quad yaw = 75$ deg

Follow the example provided in the launch file rl_fra2mo_description/launch/spawn_fra2mo_-gazebo.launch of the simulation.

spawn_fra2mo_gazebo.launch

```
...
    <node name= "goal_1" pkg="tf" type="static_transform_publisher" args
       ="-10 3 0 0 0 0 map goal1 100"/>
    <node name= "goal_2" pkg="tf" type="static_transform_publisher" args
       ="-15 7 0 0.5236 0 0 map goal2 100"/>
    <node name= "goal_3" pkg="tf" type="static_transform_publisher" args
       ="-6 8 0 3.14 0 0 map goal3 100"/>
    <node name= "goal_4" pkg="tf" type="static_transform_publisher" args
       ="-17.5 3 0 1.309 0 0 map goal4 100"/>
...
```

Used command

```
$ roslaunch fra2mo_2dnav fra2mo_nav_bringup.launch
```

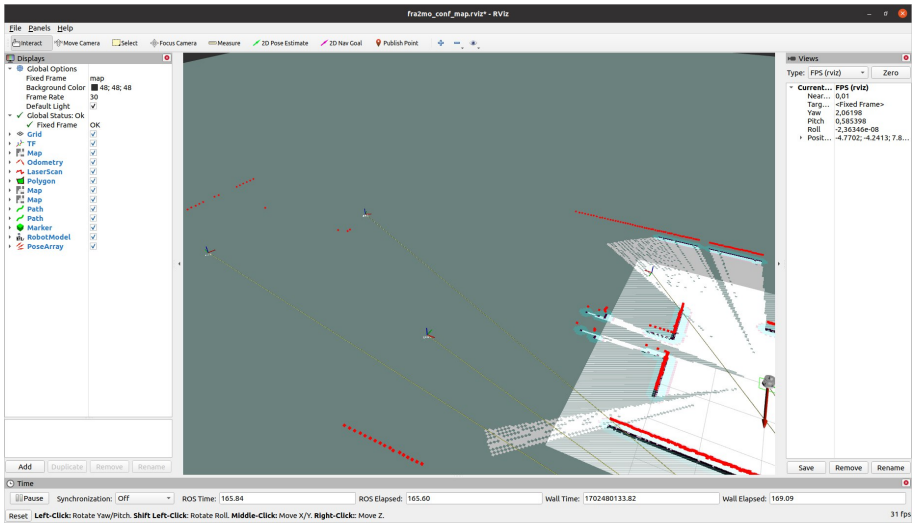

Figure 5: Frame of each goal visualized in Rviz

(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 order to implement this point and the next one, the additional node tf_nav_4goals.cpp has been created

CMakeLists.txt

```
...
    add_executable(tf_nav_4goals src/tf_nav_4goals.cpp)
    target_link_libraries(tf_nav_4goals ${catkin_LIBRARIES})
...
```

Two solutions have been implemented. In the first one, a single listener with a switch-case structure has been used. In the second one, four different listeners have been coded, one for each tf frame instead. It can be noticed that the former solution is more flexible, so that this solution has been adopted to fulfill the next points.

tf_nav_4goals.cpp

```
...
//Solution with 1 listener
void TF_NAV::goal_listener() {

    ros::Rate r( 1 );
    tf::TransformListener listener;
    tf::StampedTransform transform;

    while ( ros::ok() ){

        switch(choice){

            case 1:
            {
                try{
                    listener.waitForTransform( "map", "goal3", ros::Time
                        ( 0 ), ros::Duration( 10.0 ) );
                    listener.lookupTransform( "map", "goal3", ros::Time(
                        0 ), transform );
                }
                catch( tf::TransformException &ex ){
                    ROS_ERROR("%s", ex.what());
                    r.sleep();
                    continue;
                }
                break;
            }
            case 2:
            {
                try{
                    listener.waitForTransform( "map", "goal4", ros::Time
                        ( 0 ), ros::Duration( 10.0 ) );
                    listener.lookupTransform( "map", "goal4", ros::Time(
                        0 ), transform );
                }
                catch( tf::TransformException &ex ){
                    ROS_ERROR("%s", ex.what());
                    r.sleep();
                    continue;
                }
                break;
            }
            case 3:
            {
                try{
```

```
                    listener.waitForTransform( "map", "goal2", ros::Time
                        ( 0 ), ros::Duration( 10.0 ) );
                    listener.lookupTransform( "map", "goal2", ros::Time(
                        0 ), transform );
                }
                catch( tf::TransformException &ex ){
                    ROS_ERROR("%s", ex.what());
                    r.sleep();
                    continue;
                }
                break;
            }
            case 4:
            {
                try{
                    listener.waitForTransform( "map", "goal1", ros::Time
                        ( 0 ), ros::Duration( 10.0 ) );
                    listener.lookupTransform( "map", "goal1", ros::Time(
                        0 ), transform );
                }
                catch( tf::TransformException &ex ){
                    ROS_ERROR("%s", ex.what());
                    r.sleep();
                    continue;
                }
                break;
            }
            default:
                std::cout << "Error goal selection"<<std::endl;
        }
        _goal_pos << transform.getOrigin().x(), transform.getOrigin().y
            (), transform.getOrigin().z();
        _goal_or << transform.getRotation().w(),  transform.getRotation
            ().x(), transform.getRotation().y(), transform.getRotation().
            z();
        // std::cout<<"Goal pos: "<<std::endl<<_goal_pos<<std::endl;
        // std::cout<<"Goal or: "<<std::endl<<_goal_or<<std::endl;
        r.sleep();
    }
}
...
```

For the sake of brevity, only one listener is reported below

tf_nav_4goals.cpp

```
...
// Solution with 4 listeners
void TF_NAV::goal3_listener() {
    ros::Rate r( 1 );
    tf::TransformListener listener;
    tf::StampedTransform transform;

    while ( ros::ok() ){
        try{
                listener.waitForTransform( "map", "goal3", ros::Time( 0
                    ), ros::Duration( 10.0 ) );
                listener.lookupTransform( "map", "goal3", ros::Time( 0 )
                    , transform );
```

```
        }
        catch( tf::TransformException &ex ){
                ROS_ERROR("%s", ex.what());
                r.sleep();
                continue;
        }
        _goal3_pos << transform.getOrigin().x(), transform.getOrigin().y
            (), transform.getOrigin().z();
        _goal3_or << transform.getRotation().w(),  transform.getRotation
            ().x(), transform.getRotation().y(), transform.getRotation().
            z();
        r.sleep();
    }
}
...
```

(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.

According to the previous solutions two different send_goal functions have been coded. In the first solution, to properly update the desired goal in each step, a while loop has been exploited. In this way, the problem of reassigning the same goal can be avoided. While, in the second solution, a simple sequence of sending goal has been used.

tf_nav_4goals.cpp

```
...
//Solution with 1 listener
void TF_NAV::send_goal() {
    ros::Rate r( 5 );
    int cmd;
    move_base_msgs::MoveBaseGoal goal;
    Eigen::Vector3d goalpos_old; goalpos_old<<0.0,0.0,0.0;

    while ( ros::ok() ){
        std::cout<<"\nInsert 1 to send sequence of goals from TF
            (3->4->2->1)"<<std::endl;
        std::cout<<"Insert 2 to send home position goal "<<std::endl;
        std::cout<<"Insert your choice"<<std::endl;
        std::cin>>cmd;

        if ( cmd == 1) {

            MoveBaseClient ac("move_base", true);

            for (int i=1;i<5;i++){
                choice=i;
                //std::cout<<"choice "<<choice<<std::endl; //DEBUG

                while(goalpos_old ==_goal_pos){
                    std::cout<<"WAITING"<<std::endl;
                    r.sleep();
                }
                goalpos_old=_goal_pos;
```

```
                while(!ac.waitForServer(ros::Duration(5.0))){
                    ROS_INFO("Waiting for the move_base action server to
                        come up");
                }

                goal.target_pose.header.frame_id = "map";
                goal.target_pose.header.stamp = ros::Time::now();

                goal.target_pose.pose.position.x = _goal_pos[0];
                goal.target_pose.pose.position.y = _goal_pos[1];
                goal.target_pose.pose.position.z = _goal_pos[2];

                goal.target_pose.pose.orientation.w = _goal_or[0];
                goal.target_pose.pose.orientation.x = _goal_or[1];
                goal.target_pose.pose.orientation.y = _goal_or[2];
                goal.target_pose.pose.orientation.z = _goal_or[3];

                std::cout<<"Goal: "<<goal<<std::endl;

                ROS_INFO("Sending goal");
                ac.sendGoal(goal);

                ac.waitForResult();

                if(ac.getState() == actionlib::SimpleClientGoalState::
                   SUCCEEDED){
                    ROS_INFO("The mobile robot arrived in the TF goal");
                    if(i == 4) ROS_INFO("The sequence of goals has been
                       completed");
                }
                else{
                    ROS_INFO("The base failed to move for some reason");
                    i=5;
                }
            }
            choice=1;
        }
        ...
```

tf_nav_4goals.cpp

```
...
//Solution with 4 listeners
void TF_NAV::send_goal() {
    ros::Rate r( 5 );
    tf::TransformListener listener;
    tf::StampedTransform transform;
    int cmd;
    move_base_msgs::MoveBaseGoal goal;

    while ( ros::ok() ){
        std::cout<<"\nInsert 1 to send sequence of goals from TF
           (3->4->2->1)"<<std::endl;
        std::cout<<"Insert 2 to send home position goal "<<std::endl;
        std::cout<<"Insert your choice"<<std::endl;
        std::cin>>cmd;

        if ( cmd == 1) {
```

```
MoveBaseClient ac("move_base", true);

while(!ac.waitForServer(ros::Duration(5.0))){
    ROS_INFO("Waiting for the move_base action server to
       come up");
}

goal.target_pose.header.frame_id = "map";
goal.target_pose.header.stamp = ros::Time::now();

goal.target_pose.pose.position.x = _goal3_pos[0];
goal.target_pose.pose.position.y = _goal3_pos[1];
goal.target_pose.pose.position.z = _goal3_pos[2];

goal.target_pose.pose.orientation.w = _goal3_or[0];
goal.target_pose.pose.orientation.x = _goal3_or[1];
goal.target_pose.pose.orientation.y = _goal3_or[2];
goal.target_pose.pose.orientation.z = _goal3_or[3];

std::cout<<"goal3: "<<goal<<std::endl;
ROS_INFO("Sending goal3");
ac.sendGoal(goal);

ac.waitForResult();

if(ac.getState() == actionlib::SimpleClientGoalState::
   SUCCEEDED){
    ROS_INFO("The mobile robot arrived in the TF goal");

    while(!ac.waitForServer(ros::Duration(5.0))){
        ROS_INFO("Waiting for the move_base action server to
           come up");
    }

    goal.target_pose.header.frame_id = "map";
    goal.target_pose.header.stamp = ros::Time::now();

    goal.target_pose.pose.position.x = _goal4_pos[0];
    goal.target_pose.pose.position.y = _goal4_pos[1];
    goal.target_pose.pose.position.z = _goal4_pos[2];

    goal.target_pose.pose.orientation.w = _goal4_or[0];
    goal.target_pose.pose.orientation.x = _goal4_or[1];
    goal.target_pose.pose.orientation.y = _goal4_or[2];
    goal.target_pose.pose.orientation.z = _goal4_or[3];

    std::cout<<"goal4: "<<goal<<std::endl;
    ROS_INFO("Sending goal4");
    ac.sendGoal(goal);

    ac.waitForResult();
    ...
```

Used commands
Run each command in a different terminal

```
$ roslaunch fra2mo_2dnav fra2mo_nav_bringup.launch
```

```
$ rosrun fra2mo_2dnav tf_nav_4goals
```

```
$ rosbag record /fra2mo/pose
```

This bagfile has been exported in Matlab to retrieve the following plots

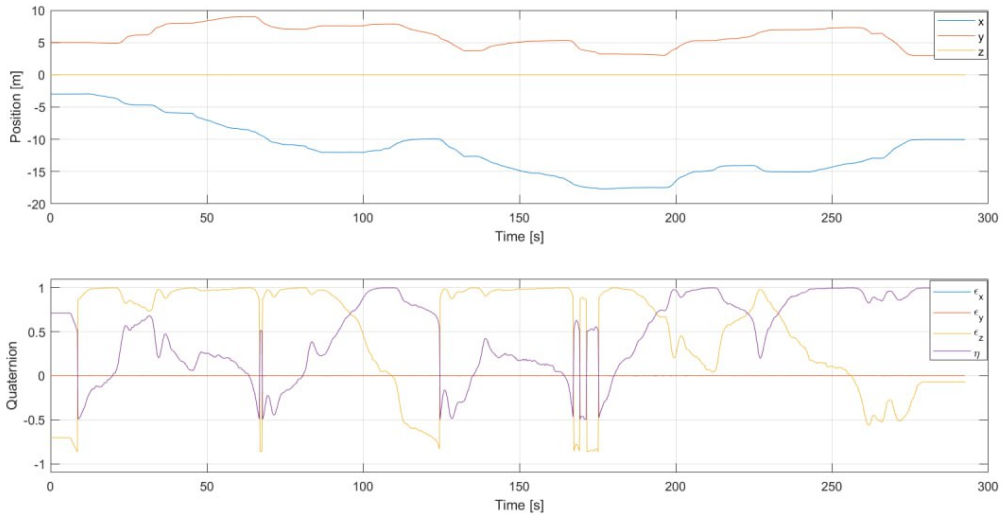

Figure 6: Recorded Robot pose

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 order to get a complete map, goal_1 and goal_2 have been changed, meanwhile goal_5 and goal_6 have been added. In order to reach these 6 goals, an additional node called tf_nav_map.cpp has been used. Its structure is the same of the previous tf_nav_4goals.cpp node.

spawn_fra2mo_gazebo.launch

```
<!-- tf_nav_map  -->
<node pkg="tf" type="static_transform_publisher" name="goal_1" args="
    -7 1 0 0 0 0 map goal1 100" />
<node pkg="tf" type="static_transform_publisher" name="goal_2" args="
    -16 9.5 0 0.5236 0 0 map goal2 100" />
<node pkg="tf" type="static_transform_publisher" name="goal_3" args="
    -6 8 0 3.14 0 0 map goal3 100" />
<node pkg="tf" type="static_transform_publisher" name="goal_4" args="
    -17.5 3 0 1.309 0 0 map goal4 100" />
<node pkg="tf" type="static_transform_publisher" name="goal_5" args="
    -0.5 9.5 0 0 0 0 map goal5 100" />
<node pkg="tf" type="static_transform_publisher" name="goal_6" args="
    -0.5 0.5 0 0 0 0 map goal6 100" />
```

Used commands
Run each command in a different terminal

```
$ roslaunch fra2mo_2dnav fra2mo_nav_bringup.launch
```

```
$ rosrun fra2mo_2dnav tf_nav_map
```

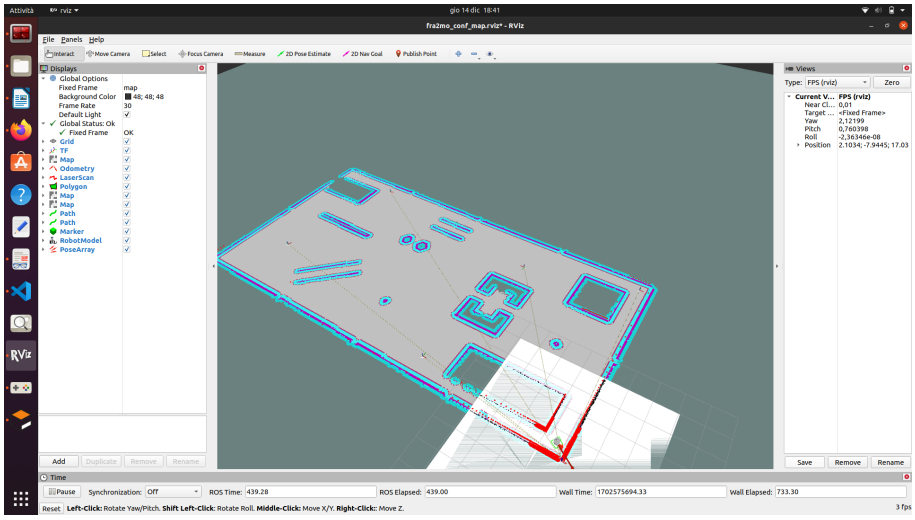

Figure 7: Complete map of the environment

(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.

The following configurations have been attempted:

1.
teb_local_planner_params.yaml

```
...
 # Robot
 max_vel_x: 10.6
 max_vel_x_backwards: 10.3
 max_vel_theta: 10.3
 acc_lim_x: 7.1
 acc_lim_theta: 6.3
...
```

Increasing too much the max velocities and accelerations values, it can be noticed that the robot loses control, and it is not able to properly react to obstacles presence. The final result is a collision of the robot on an obstacle.

2.
teb_local_planner_params.yaml

```
...
 # Obstacles
 min_obstacle_dist: 0.05
...
```

A similar collision occurs even if the min_obstacle_dist parameter is reduced too much. In this case, due to planner approximations and a too small margin from obstacles, the robot does not manage to reach even the first goal.

3.
teb_local_planner_params.yaml

```
...
 # Robot
...
 footprint_model:
  type: "polygon"
  vertices: [[0.2, -0.2],
             [-0.2, -0.2],
             [-0.2, 0.2],
             [0.2, 0.2]]
...
```

To try to compensate planner approximation, it is possible to increase the footprint_model dimension. Nevertheless, if this value is too high, the robot stucks even if no obstacle is touched. This happens because the robot overestimates its actual dimensions, preventing it from passing through obstacles.

4.
local_costmap_params.yaml

```
local_costmap:
  global_frame: map
```

```
  robot_base_frame: base_footprint
  update_frequency: 10.0
  publish_frequency: 20.0
  static_map: false
  rolling_window: true
  width: 20
  height: 20
  resolution: 0.03
```

global_costmap_params.yaml

```
global_costmap:
  global_frame: map
  robot_base_frame: base_footprint
  update_frequency: 10.0
  publish_frequency: 5.0
  #always_send_full_costmap: false #default is false
  rolling_window: false
  resolution: 0.05
  width: 50
  height: 50
  origin_x: -15
  origin_y: -15
```

Changing the frequency and dimension of the local and global costmaps, the performance of the robot does not change significantly.

5.
teb_local_planner_params.yaml

```
...
 # Robot
 max_vel_x: 2
 max_vel_x_backwards: 1
 max_vel_theta: 1
 acc_lim_x: 1
 acc_lim_theta: 0.5
...
```

teb_local_planner_params.yaml

```
...
 # Obstacles
 min_obstacle_dist: 0.15
...
```

local_costmap_params.yaml

```
local_costmap:
  global_frame: map
  robot_base_frame: base_footprint
  update_frequency: 5.0
  publish_frequency: 10.0
  static_map: false
  rolling_window: true
  width: 10
  height: 10
  resolution: 0.03
```

costmap_common_params.yaml

```
...
obstacle_range: 10
raytrace_range: 5
...
```

A final configuration is obtained by changing parameters in almost all suggested files. Using a trial and error approach, these reported values manage to improve the robot performance in terms of speed and obstacle avoidance.

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-noetic-realsense2-description

fra2mo.xacro

```
...
  <!-- Uncomment if you want to add also a D435 camera -->
  <!-- RGBD Sensor -->
  <xacro:if value="${DEPTH}" >
    <xacro:d435_gazebo_sensor parent="d435_link" />
  </xacro:if>
...
```

To avoid an "Unsupported Gazebo ImageFormat" error, the image format in camera xacro has been modified as follows

d435_gazebo_macro.xacro

```
...
<format>R8G8B8</format>
...
```

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 send the robot in the proximity of obstacle 9, a new tf frame has been defined

spawn_fra2mo_gazebo.launch

```
  <!-- tf_nav_visual  -->
  <node pkg="tf" type="static_transform_publisher" name="goal_camera"
     args="-15 8 0 3.14 0 0 map goal_camera 100" />
```

In order to accomplish this point, the tf_nav_visual.cpp node has been implemented. First of all, the previous nodes structure has been followed to approach obstacle 9

tf_nav_visual.cpp

```
...
void TF_NAV::goal_listener() {
    ros::Rate r( 1 );
    tf::TransformListener listener;
    tf::StampedTransform transform;

    while ( ros::ok() ) {
        try {
```

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

        _goal_pos << transform.getOrigin().x(), transform.getOrigin().y
            (), transform.getOrigin().z();
        _goal_or << transform.getRotation().w(),  transform.getRotation
            ().x(), transform.getRotation().y(), transform.getRotation().
            z();
        r.sleep();
    }
}
...
```

To retrieve marker pose, a subscriber to /aruco_single/pose has been implemented

```
int main( int argc, char** argv ) {
    ros::init(argc, argv, "tf_navigation");
    ros::NodeHandle n;
    TF_NAV tfnav;

    ros::Subscriber aruco_pose_sub = n.subscribe("/aruco_single/pose",
        1, arucoPoseCallback);
    tfnav.run();

    return 0;
}
```

```
...
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, tf_listener_aruco function has been coded. 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. This reasoning has been implemented as shown below

```
...
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();
    }
}
...
```

Once the robot has approached the obstacle and the marker has been detected, a new goal has been defined with respect to marker position and orientation

```
...
void TF_NAV::send_goal() {
```

```
    ros::Rate r( 5 );
    int cmd;
    move_base_msgs::MoveBaseGoal goal;

    while ( ros::ok() ) {
        std::cout<<"\nInsert 1 to send goal from TF "<<std::endl;
        std::cout<<"Insert 2 to send home position goal "<<std::endl;
        std::cout<<"Inser your choice"<<std::endl;
        std::cin>>cmd;

        if ( cmd == 1) {
            MoveBaseClient ac("move_base", true);
            while(!ac.waitForServer(ros::Duration(5.0))){
            ROS_INFO("Waiting for the move_base action server to come up
                ");
            }
            goal.target_pose.header.frame_id = "map";
            goal.target_pose.header.stamp = ros::Time::now();

            goal.target_pose.pose.position.x = _goal_pos[0];
            goal.target_pose.pose.position.y = _goal_pos[1];
            goal.target_pose.pose.position.z = _goal_pos[2];

            goal.target_pose.pose.orientation.w = _goal_or[0];
            goal.target_pose.pose.orientation.x = _goal_or[1];
            goal.target_pose.pose.orientation.y = _goal_or[2];
            goal.target_pose.pose.orientation.z = _goal_or[3];

            ROS_INFO("Sending goal to approach Aruco");
            ac.sendGoal(goal);

            ac.waitForResult();

            if(ac.getState() == actionlib::SimpleClientGoalState::
                SUCCEEDED && aruco_pose_available){
                ROS_INFO("The mobile robot arrived in the TF goal");

                while(!ac.waitForServer(ros::Duration(5.0))){
                ROS_INFO("Waiting for the move_base action server to
                    come up");
                }
                goal.target_pose.header.frame_id = "map";
                goal.target_pose.header.stamp = ros::Time::now();

                goal.target_pose.pose.position.x = _aruco_pos[0] + 1;
                goal.target_pose.pose.position.y = _aruco_pos[1];
                goal.target_pose.pose.position.z = 0.0;
/*
                goal.target_pose.pose.orientation.w = 0.0;
                goal.target_pose.pose.orientation.x = 0.0;
                goal.target_pose.pose.orientation.y = 0.0;
                goal.target_pose.pose.orientation.z = 1.0;
*/
                KDL::Rotation Ar_rot=KDL::Rotation::Quaternion(_aruco_or
                    [1], _aruco_or[2], _aruco_or[3], _aruco_or[0]);
                KDL::Rotation Rb_des=Ar_rot*KDL::Rotation::RotX(-1.5708)
                    *KDL::Rotation::RotZ(1.5708);
                double R,P,Y,x,y,z,w;
```

```
                Rb_des.GetRPY(R,P,Y);
                Rb_des=KDL::Rotation::RPY(0,0,Y);
                Rb_des.GetQuaternion(x,y,z,w);

                goal.target_pose.pose.orientation.w = w;
                goal.target_pose.pose.orientation.x = x;
                goal.target_pose.pose.orientation.y = y;
                goal.target_pose.pose.orientation.z = z;

                //std::cout<<"Goal2:"<<std::endl<< goal <<std::endl;

                ROS_INFO("Sending goal wrt Aruco pose");
                ac.sendGoal(goal);

                ac.waitForResult();

                if(ac.getState() == actionlib::SimpleClientGoalState::
                    SUCCEEDED) {
                     ROS_INFO("The mobile robot arrived in the TF goal");
                     ROS_INFO("The sequence of goals has been completed")
                         ;
                }
                else
                     ROS_INFO("The base failed to move for some reason");
            }
            else
                ROS_INFO("The base failed to move for some reason");
        }
        ...
```

Knowing marker pose, the goal orientation can be assigned in a fixed way as $x = 0, y = 0, z = 1, w = 0$. Alternatively, a more robust solution takes care also of the aruco orientation. First of all, to define the desired base_footprint frame, the marker frame has been suitably rotated.

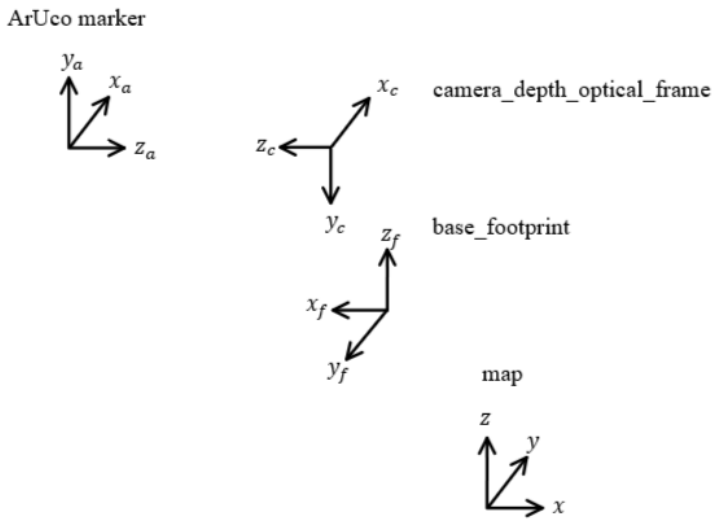

Figure 8: Frames configuration

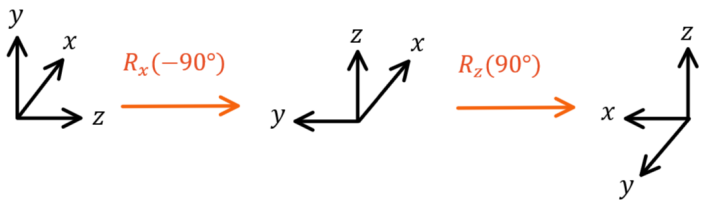

Figure 9: Definition of base_footprint reference

Due to approximation and measurement errors, it is not possible to directly extract quaternion reference, 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.

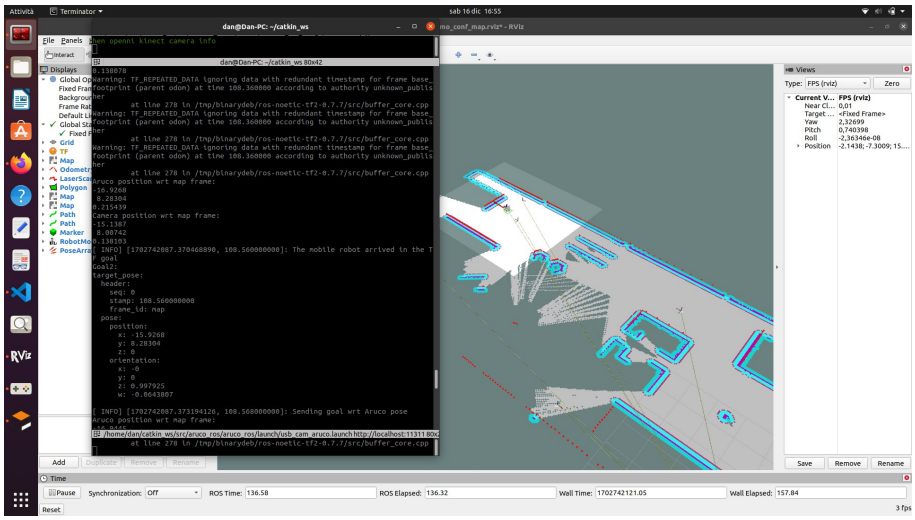

Figure 10: Printing of the target pose

Used commands

Run each command in a different terminal

```
$ roslaunch fra2mo_2dnav fra2mo_nav_bringup.launch
```

```
$ rosrun fra2mo_2dnav tf_nav_visual
```

```
$ roslaunch aruco_ros usb_cam_aruco.launch camera:=/depth_camera/
   depth_camera markerId:=115
```

```
$ rosrun rqt_image_view
```

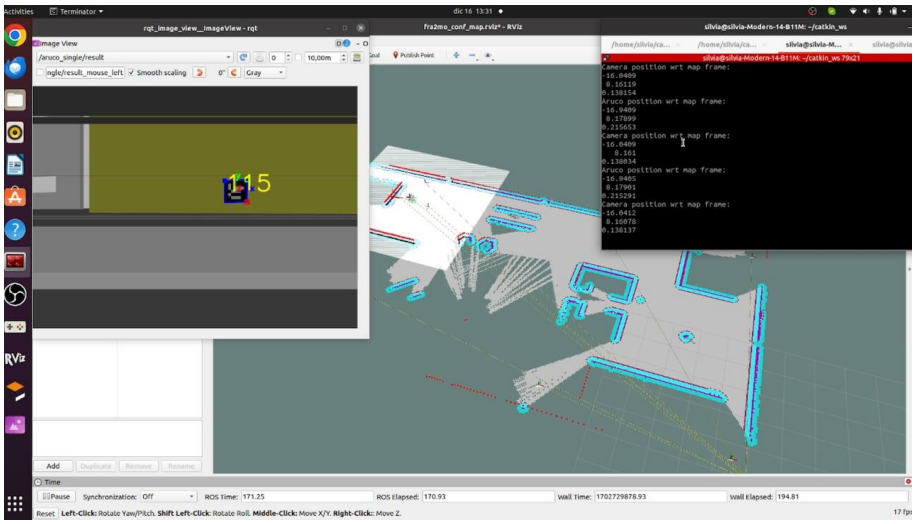

Figure 11: Robot in the goal spot computed with respect to the ArUco marker pose

c) Publish the ArUco pose as TF following the example at this link.

tf_nav.h

```
#include <tf/transform_broadcaster.h>
```

To create tf frame, it is firstly necessary to publish the aruco pose on a proper topic: aruco_frame/pose

tf_nav_visual.cpp

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

```
void TF_NAV::position_aruco_pub() {

    geometry_msgs::PoseStamped pose;

    pose.header.stamp = ros::Time::now();
    pose.header.frame_id = "map";

    pose.pose.position.x = _aruco_pos[0];
    pose.pose.position.y = _aruco_pos[1];
    pose.pose.position.z = _aruco_pos[2];

    pose.pose.orientation.w = _aruco_or[0];
    pose.pose.orientation.x = _aruco_or[1];
    pose.pose.orientation.y = _aruco_or[2];
    pose.pose.orientation.z = _aruco_or[3];

    _position_aruco_pub.publish(pose);
}
```

```
void TF_NAV::tf_listener_aruco() {
...
        position_aruco_pub();
        r.sleep();
```

```
    }
}
```

Once the aruco pose is published, a subscriber has been used in order to retrieve it

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

In the corresponding callback, the received message is broadcasted as a tf frame as shown below

```
void poseCallback(const geometry_msgs::PoseStamped & msg) {
  static tf::TransformBroadcaster br;
  tf::Transform transform;
  transform.setOrigin( tf::Vector3(msg.pose.position.x,msg.pose.position
      .y,msg.pose.position.z));
  tf::Quaternion q(msg.pose.orientation.x,msg.pose.orientation.y,msg.
      pose.orientation.z,msg.pose.orientation.w);
  transform.setRotation(q);
  br.sendTransform(tf::StampedTransform(transform, ros::Time::now(), "
      map", "aruco_frame"));
}
```

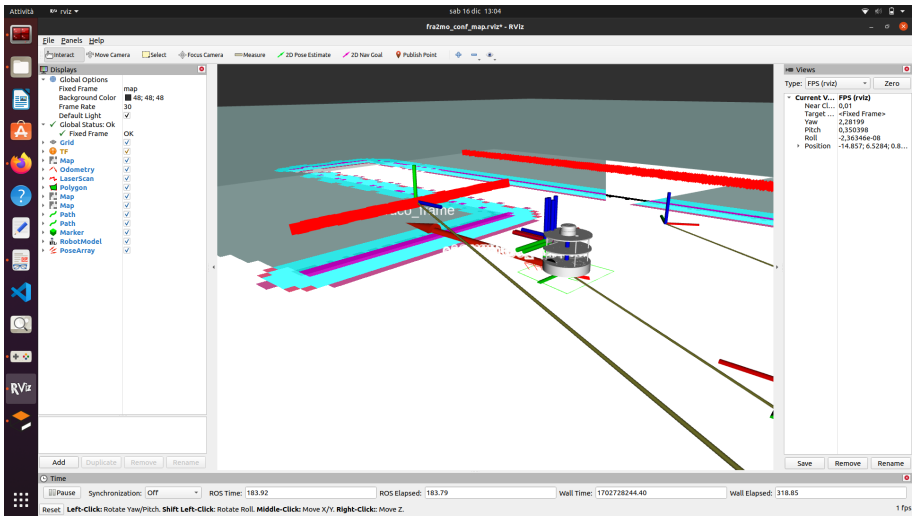

Figure 12: aruco_frame tf visualization

Figure 13: /aruco_frame/pose topic