Report – Hands-on setup class

Students:

Nimrod Millenium Ndulue

Salvatore Del Peschio

Salvatore Oliviero

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:*

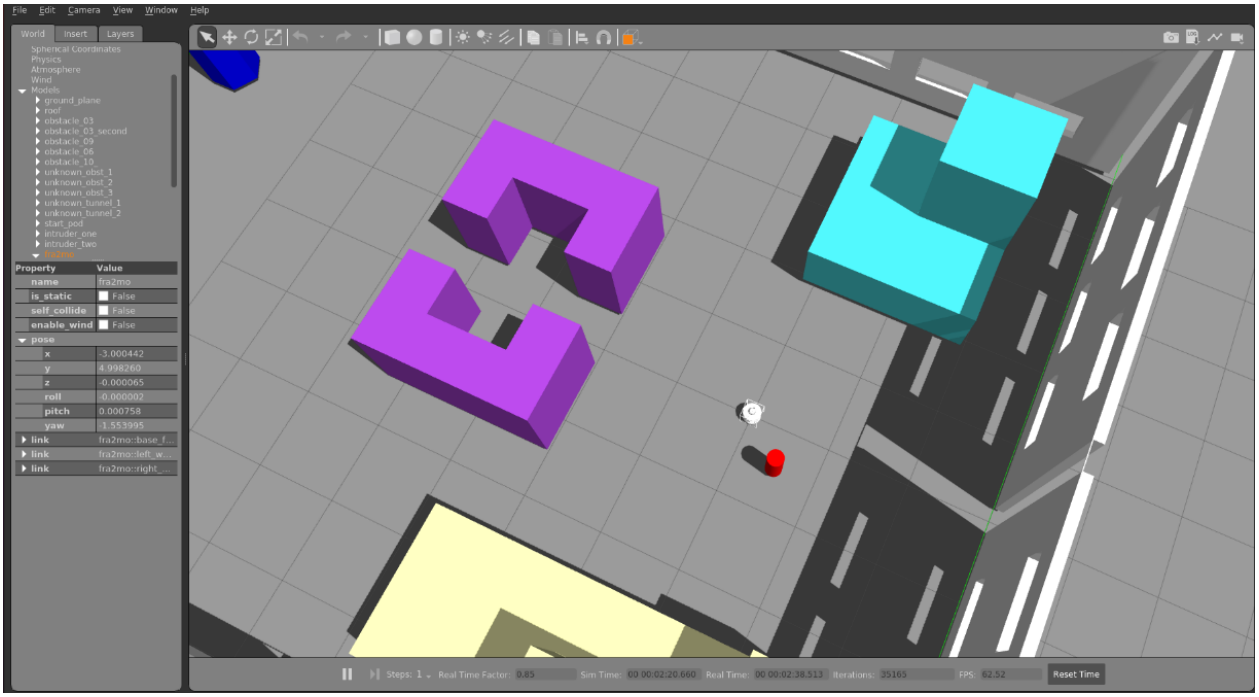
$$x = -3;\ y = 5;\ yaw = -90°$$

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

To define the robot's position and orientation as required, it's necessary to establish four parameters within the ‘spawn_frame_gazebo.launch’ file, each set to the relevant values. These values will then be passed to the ‘urdf_spawner’ node, responsible of the robot's instantiation. Below is the outlined implementation:

```
<!-- these are the arguments you can pass this launch file, for example paused:=true -->
<arg name="paused" default="false"/>
<arg name="use_sim_time" default="true"/>
<arg name="gui" default="true"/>
<arg name="headless" default="false"/>
<arg name="debug" default="false"/>
<arg name="x_pos" default="-3.0"/>
<arg name="y_pos" default="5.0"/>
<arg name="z_pos" default="0.1"/>
<arg name="yaw_pos" default="-1.57"/>
<env name="GAZEBO_MODEL_PATH" value="$(find rl_racefield)/models:$(optenv GAZEBO_MODEL_PATH)"/>
```

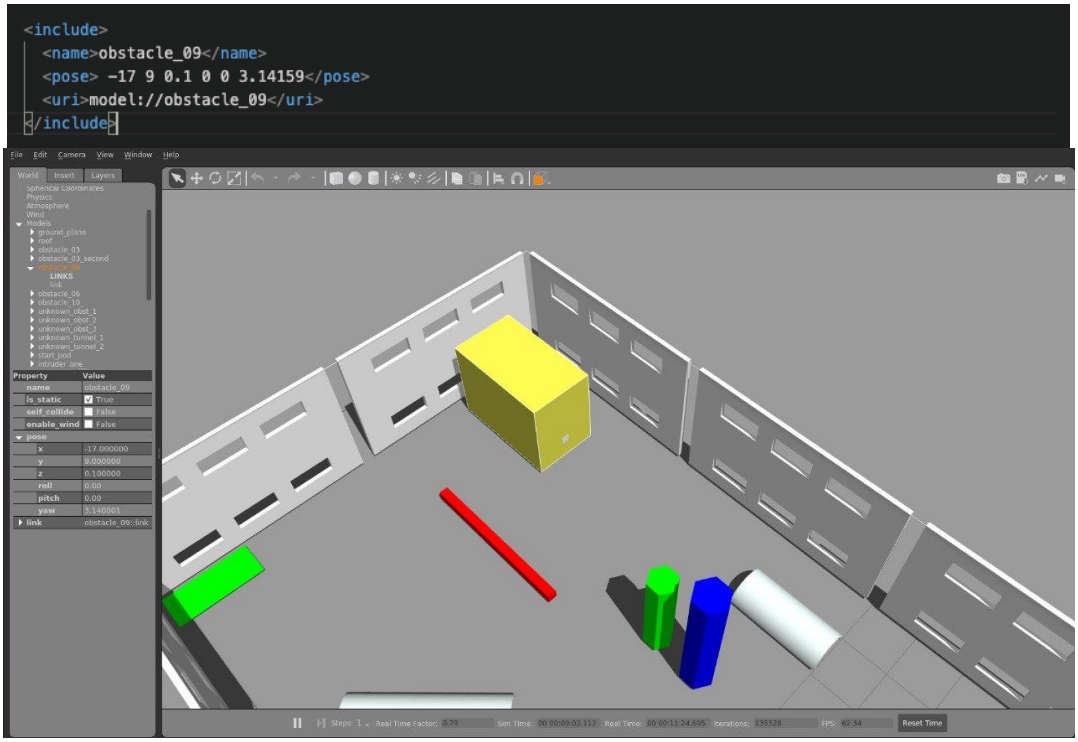
```
<!-- Run a python script to the send a service call to gazebo_ros to spawn a URDF robot -->
<node name="urdf_spawner" pkg="gazebo_ros" type="spawn_model" respawn="false" output="screen"
args="-urdf -model fra2mo -x $(arg x_pos) -y $(arg y_pos) -z $(arg z_pos) -Y $(arg yaw_pos) -param robot_descrip
```



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

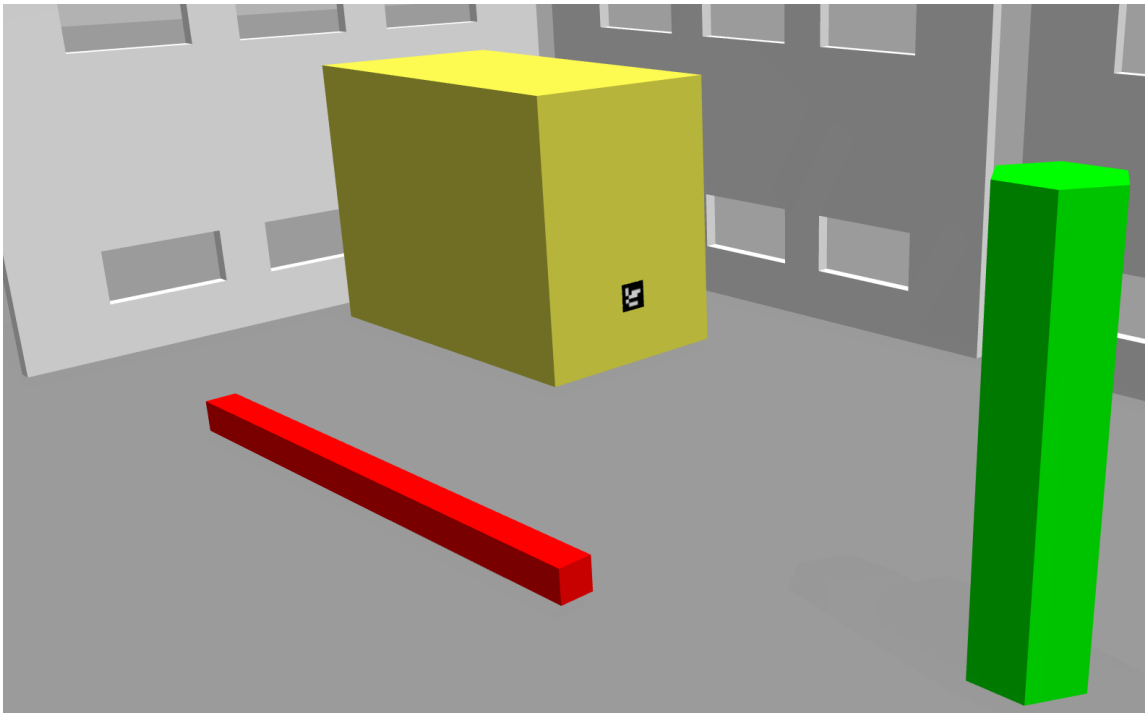
$$x = -17 \;\; y = 9 \;\; z = 0.1 \;\; yaw = 3.14$$

To position obstacle 9 at the requested location, modifications need to be made within the file ‘rl_race_field.world’. This file contains information about the objects to be included in the environment. Therefore, adjustments must be made to the pose of obstacle 9 relative to the world. Subsequently are reported the code and the result.



c) *Place the ArUco marker number 115 1 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.*

For this task, we've established a directory named “materials” within the folder that holds the description of obstacle 9, where we added two directories specifying both "scripts" and "textures". Within the scripts section, we've incorporated the aruco 115, delineating its name and format to guarantee its visibility in Gazebo. Simultaneously, within the textures section, we've inserted the image of the aruco in PNG format. Following this, in the SDF file of obstacle 9, we've made references to these features using the <uri>...</uri> tag.



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:* $x = -10\ y = 3\ yaw = 0\,\text{deg}$
- *Goal 2:*$x = -15\ y = 7\ yaw = 30\,\text{deg}$
- *Goal 3:* $x = -6\ y = 8\ yaw = 180\,\text{deg}$
- *Goal 4:* $x = -17.5\ y = 3\ yaw = 75\,\text{deg}$

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

In line with the provided example, within the 'spawn_fra2mo_gazebo.launch' file, we have generated four transformation frames (tf) delineating the intended position and orientation.

Subsequently is shown the code:

```
<!--Static tf publisher for goals-->
<node pkg="tf" type="static_transform_publisher" name="goal_1_pub" args="-10   3 0 0      0 0 map goal1 100" />
<node pkg="tf" type="static_transform_publisher" name="goal_2_pub" args="-15   7 0 0.5235 0 0 map goal2 100" />
<node pkg="tf" type="static_transform_publisher" name="goal_3_pub" args="-6    8 0 3.1415 0 0 map goal3 100" />
<node pkg="tf" type="static_transform_publisher" name="goal_4_pub" args="-17.5 3 0 1.3090 0 0 map goal4 100" />
```

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.*

The implementation of TF listeners and the sequential sending of the goals, as requested in question 2.c has been executed within a file named 'four_goals.cpp'. This section will specifically address the implementation of listeners. The subsequent paragraph will elucidate how the sending of goals through 'move_base' has been realized.

Below is the implemented code, highlighting the lines of code related to question 2.b.

fra2mo_2dnav > src > four_goals.cpp

```cpp
#include <ros/ros.h>
#include "../include/tf_nav.h"
#include <vector>
#include <cmath>
#include <tf2/LinearMath/Quaternion.h>
#include <move_base_msgs/MoveBaseAction.h>
#include <actionlib/client/simple_action_client.h>

typedef actionlib::SimpleActionClient<move_base_msgs::MoveBaseAction> MoveBaseClient;
std::vector<std::string> traj = {"goal3", "goal4", "goal2", "goal1"};

const float toRadians = M_PI/180.0;

int main(int argc, char** argv){
  ros::init(argc, argv, "simple_navigation_goals");
  tf::TransformListener listener;
  tf::StampedTransform transform;
  move_base_msgs::MoveBaseGoal goal;

  ros::Rate r(1);

  //tell the action client that we want to spin a thread by default
  MoveBaseClient ac("move_base", true);

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

  for(int i = 0; i < traj.size(); i++){
    try{
      listener.waitForTransform( "map", traj[i], ros::Time(0), ros::Duration(10.0));
      listener.lookupTransform( "map", traj[i], ros::Time(0), transform);
      ROS_INFO("Current position:\t\t %f, %f, %f", transform.getOrigin().x(), transform.getOrigin().y(), transform.getOrigin().z(
      ROS_INFO("Current orientation [x,y,z,w]:\t %f, %f, %f, %f", transform.getRotation().x(), transform.getRotation().y(), trans
    }
    catch( tf::TransformException &ex ) {
      ROS_ERROR("%s", ex.what());
      r.sleep();
      continue;
    }
```

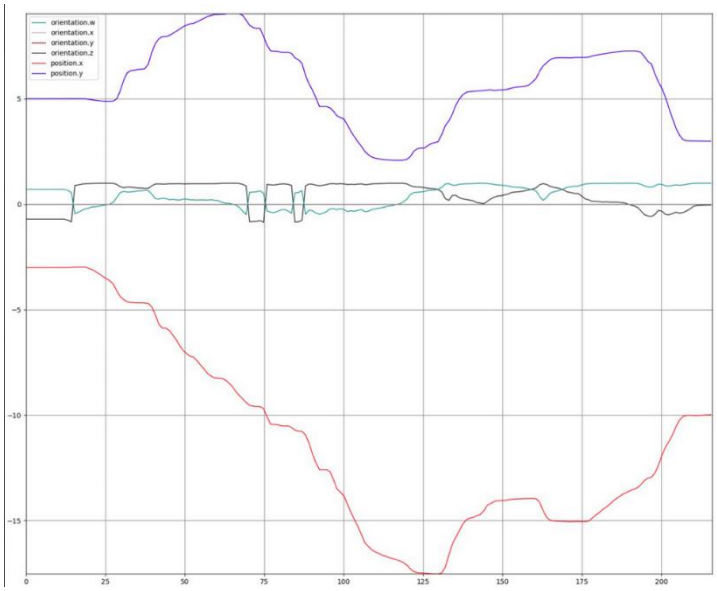
The initialization begins by defining a vector of strings named traj that encapsulates the names of predetermined goals within the map. Subsequently, a TransformListener object, denoted as listener, is instantiated to monitor transformations within the TF (Transform) framework. Alongside, a StampedTransform object named transform is declared to retain transformation data. Utilizing a 'for' loop, the code iterates through the preset goals. Within this loop, the functions waitForTransform and lookupTransform are employed to procure the transformation data between the "map" frame and the current goal denoted by traj[i]. This try-catch block retrieves the pertinent position and orientation information for the respective goal. Following this retrieval, the obtained data is printed to the terminal as debug using the ROS_INFO function.

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.*

```
for(int i = 0; i < traj.size(); i++){
  try{
    // Collect goals and print to terminal as debug
    listener.waitForTransform( "map", traj[i], ros::Time(0), ros::Duration(10.0));
    listener.lookupTransform( "map", traj[i], ros::Time(0), transform);
    ROS_INFO("Current goal position:\t %f, %f, %f", transform.getOrigin().x(), transform.getOrigin
    ROS_INFO("Current goal orientation [x,y,z,w]:\t %f, %f, %f, %f\n", transform.getRotation().x(
  }
  catch( tf::TransformException &ex ) {
    ROS_ERROR("%s", ex.what());
    r.sleep();
    continue;
  }

  // Set and send the goal
  goal.target_pose.header.frame_id = "map";
  goal.target_pose.header.stamp = ros::Time::now();
  goal.target_pose.pose.position.x = transform.getOrigin().x();
  goal.target_pose.pose.position.y = transform.getOrigin().y();
  goal.target_pose.pose.orientation.x = transform.getRotation().x();
  goal.target_pose.pose.orientation.y = transform.getRotation().y();
  goal.target_pose.pose.orientation.z = transform.getRotation().z();
  goal.target_pose.pose.orientation.w = transform.getRotation().w();
  ROS_INFO("Sending goal");
  ac.sendGoal(goal);

  ac.waitForResult();

  if(ac.getState() == actionlib::SimpleClientGoalState::SUCCEEDED){
    ROS_INFO("Hooray, the base moved\n");
    ros::Duration(1.0).sleep();
  }
  else{
    ROS_INFO("The base failed to move for some reason");
    break;
  }
}
```

In reference to the image shown above, the implementation of the algorithm handling the transmission of goals was performed using an instance of the MoveBaseGoal class, which will subsequently hold the information regarding the goal pose relative to the map frame. This information will then be passed to the 'ac' instance of the MoveBaseClient class, which, through the method "sendGoal", will command the robot to achieve the required pose and with the waitForResult method will block the execution of the node till a result from the Action server is received.

Below the plot of the executed robot trajectory:

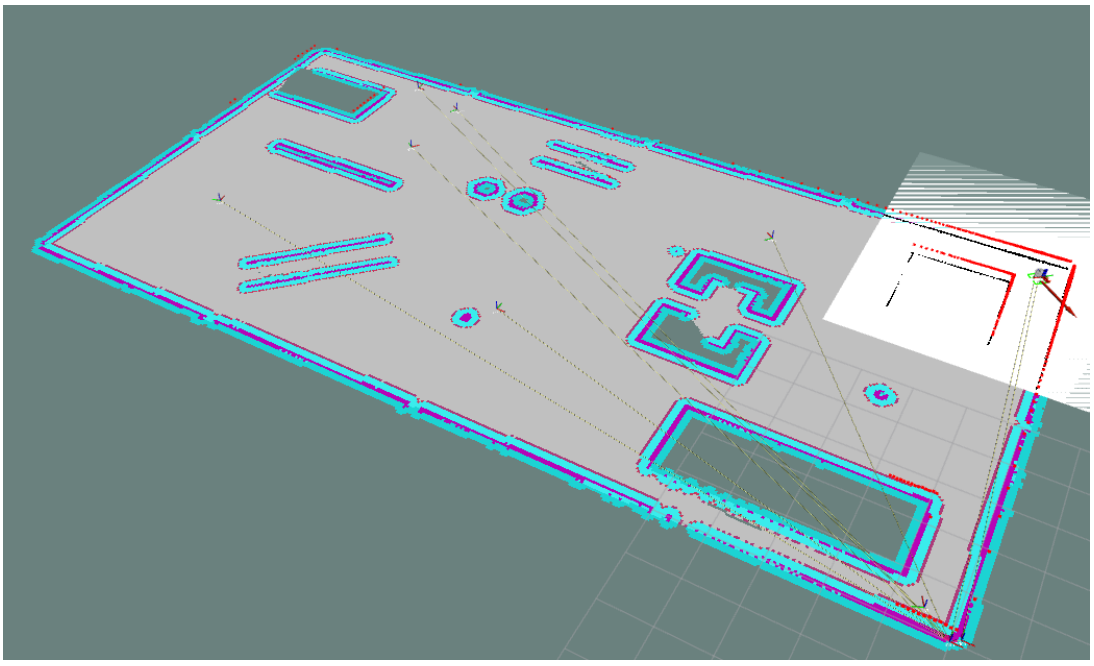


3. Map the environment tuning the navigation stack's parameters

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

To obtain a complete map of the environment, three additional goals have been added at strategic positions. Below are the coordinates of the various goals and the image of the fully explored map.

```
<!-- Goals for exploration -->
<node pkg="tf" type="static_transform_publisher" name="goal_5_pub" args="-0.5  0.4 0 1.57 0 0 map goal5 100" />
<node pkg="tf" type="static_transform_publisher" name="goal_6_pub" args="-16.5 9.4 0 0    0 0 map goal6 100" />
<node pkg="tf" type="static_transform_publisher" name="goal_7_pub" args="-0.7  9.5 0 0    0 0 map goal7 100" />
```



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 tra*jectory, 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 anges and footprint coherently as done in planner parameters.

Four different configurations have been considered interesting and are explained below:

1) Race mode:
   In teb_local_planner_params.yaml:

- max_vel_x: 1.0
- max_vel_theta: 1.0
- acc_lim_x: 0.5
- acc_lim_theta: 0.5

Running the simulation with this configuration, we obtained a much more agile performance by the robot but sudden decelerations near obstacles were observed.

2) Obstacle avoidance:
   In teb_local_planner_params.yaml:

- min_obstacle_dist: 0.4 #0.25

Remarkable obstacle avoidance, unfeasible trajectories may occur when the robot try to go through narrow corridors with the consequent abort of the current goal.

3) Sharp curves:
   In local_costmap_params.yaml:

- resolution: 0.01
  In teb_local_planner_params.yaml:
- dt_hysteresis: 0.1

We noticed a big radius of curvature of the robot despite the min_turning_radius parameter was set to 0.0 so we decided to act on the planner modifying the dt_hysteresis (represents the hysteresis time, indicating a delay in the planner's response.) and the resolution of the local costmap to obtain a planning algorithm with sharp curves near the corners of the obstacles , and simultaneously obtaining a more detailed representation of the environment but potentially requiring more computational resources. It is important to notice that a more detailed representation of the environment requires more computational resources.

4) Minimal look ahead:
   In teb_local_planner_params.yaml:

- max_global_plan_lookahead_dist: 1.0
- acc_lim_x 0.7
- max_vel_x: 2.0

With this configuration, the robot ends up hitting obstacles given that decreasing the parameter 'max_global_plan_lookahead_dist', which determines the maximum distance the planner looks ahead along the local plan for trajectory calculation, and increasing the acceleration limit the robot gains too much speed and is unable to stop in time before hitting the obstacle.

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.*

In the 'fra2mo.xacro' file the following lines of code were uncommented and 'd435_gazebo_macro.xacro' was included:

```
<!-- RGBD Sensor -->
<xacro:if value="${DEPTH}" >
  <xacro:d435_gazebo_sensor parent="d435_link" />
</xacro:if>
```

```
<xacro:include filename="$(find rl_fra2mo_description)/urdf/d435_gazebo_macro.xacro" />
```

In the launch file 'fra2mo_nav_bringup.launch', it is necessary to include the launch file 'usb_cam_aruco.launch', which handles the configuration for ArUco marker detection using a USB camera.

```
<include file="$(find aruco_ros)/launch/usb_cam_aruco.launch">
    <arg name="markerId"    value="115"/>
    <arg name="markerSize" value="0.2"/>
    <arg name="camera"      value="depth_camera/depth_camera"/>
</include>
```

The <arg> tags within the <include> block set the marker ID to 115, the marker size to 0.2, and the camera topic to "depth_camera/depth_camera."

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*

$$x = x_m + 1 \; y = y_m$$

where x_m, y_m *are the marker coordinates.*

To bring the robot to a distance of one meter from the ArUco marker, we first had to make sure that the robot moved in a location near the ArUco marker. Thus, we set a goal for the robot to reach a position near the one where the code is located, that was done by adding a static transform in the file "spawn_fra2mo_gazebo.launch". Subsequently, the position of this goal was obtained using the try-catch approach, using the "waitForTransform" and "lookupTransform" methods of the TransformListener class.

```
try{
  // Collect goals and print to terminal as debug
  listener.waitForTransform( "map", "aruco_prox", ros::Time(0), ros::Duration(10.0));
  listener.lookupTransform( "map", "aruco_prox", ros::Time(0), prox_goal);
  ROS_INFO("Current goal position:\t %f, %f, %f", prox_goal.getOrigin().x(), prox_goal.getOrigin().y(), p
  ROS_INFO("Current goal orientation [x,y,z,w]:\t %f, %f, %f, %f\n", prox_goal.getRotation().x(), prox_go
}
catch( tf::TransformException &ex ){
  ROS_ERROR("%s", ex.what());
  ros::shutdown();
}
goal.target_pose.header.frame_id = "map";
goal.target_pose.header.stamp = ros::Time::now();
goal.target_pose.pose.position.x = prox_goal.getOrigin().x();
goal.target_pose.pose.position.y = prox_goal.getOrigin().y();
goal.target_pose.pose.orientation.x = prox_goal.getRotation().x();
goal.target_pose.pose.orientation.y = prox_goal.getRotation().y();
goal.target_pose.pose.orientation.z = prox_goal.getRotation().z();
goal.target_pose.pose.orientation.w = prox_goal.getRotation().w();
ROS_INFO("Sending goal");
ac.sendGoal(goal);
```

The goal to reach was sent to the robot using the 'ac' instance of the MoveBaseClient class, as in question 2.c, it is important to notice that once robot sees the ArUco marker, it will stop moving towards the goal to retrieve the aruco pose and compute the next gal.

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

  if(goal_execution){
    if(ac.getState() == actionlib::SimpleClientGoalState::SUCCEEDED){
      ROS_INFO("Hooray, the base moved");
      ros::shutdown();
    }
    else if(ac.getState() == actionlib::SimpleClientGoalState::ABORTED){
      ROS_INFO("The base failed to move for some reason");
      ros::shutdown();
    }
  }

  // ROS_INFO("aruco_pose_available: %d", int(aruco_pose_available));
  if(aruco_pose_available){ // Every time we detect the aruko marker we publish its tf
    if(goal_execution){
      ROS_INFO("Aruco Marker detected!");
      ac.cancelGoal();
      goal_execution = false;
    }
    // d435 -> aruco_marker
    Eigen::Vector3d p_cam_to_object  = {aruco_pose[0], aruco_pose[1], aruco_pose[2]};
    Eigen::Quaterniond quaternion_cam(aruco_pose[6], aruco_pose[3], aruco_pose[4], aruco_pose[5]);
    Eigen::Matrix3d rot_cam_to_object = quaternion_cam.toRotationMatrix();

    // base_footprint -> d435
    Eigen::Vector3d p_base_to_cam = {0.0975, 0.0, 0.065 + 0.059 /* base_footprint -> base_link -> d435_joint*/};
    Eigen::Matrix3d rot_base_to_cam = (Eigen::AngleAxisd(-90.0 * toRadians, Eigen::Vector3d::UnitZ()) *
                                       Eigen::AngleAxisd(0.0, Eigen::Vector3d::UnitY()) *
                                       Eigen::AngleAxisd(-90.0 * toRadians, Eigen::Vector3d::UnitX())).toRotationMatrix();

    // aruco pose respect to the base_footprint frame
    Eigen::Vector3d p_base_to_object = p_base_to_cam + rot_base_to_cam*p_cam_to_object;
```

```
// map -> base_footprint
try{
  listener.waitForTransform( "map", "base_footprint", ros::Time(0), ros::Duration(10.0));
  listener.lookupTransform( "map", "base_footprint", ros::Time(0), base_footprint_tf);
}
catch( tf::TransformException &ex ) {
  ROS_ERROR("%s", ex.what());
  loop_rate.sleep();
  continue;
}
Eigen::Vector3d p_map_to_base = {base_footprint_tf.getOrigin().x(), base_footprint_tf.getOrigin().y(), base_footprint_tf.getOrigin().z(
Eigen::Quaterniond quaternion_base(base_footprint_tf.getRotation().w(), base_footprint_tf.getRotation().x(), base_footprint_tf.getRotat
Eigen::Matrix3d rot_map_to_base = quaternion_base.toRotationMatrix();

// aruco pose respect to the map frame
Eigen::Vector3d p_map_to_object = p_map_to_base + rot_map_to_base*p_base_to_object;
Eigen::Matrix3d rot_map_to_object = rot_map_to_base*rot_base_to_cam*rot_cam_to_object;
```

The computation of the aruco pose respect to the map frame is shown above, through the information obtained from the camera and the Tf listener of the ‘base_footprint’ we can retrieve the pose of the ArUco marker. It is important to notice that the translation vector of the ‘base_footprint’ to the camera is obtained directly from the urdf of the robot, while the rotation matrix of the camera respect to the ‘base_footprint’ frame is known and represented in the code as a composition of elementary rotation.

```
// Set the desired position the first time we detect the aruco
if(!find_des_pose){
  // Compute the desire position from the aruco pose
  Eigen::Vector3d offset = {1.0, 0.0, 0.0};
  Eigen::Vector3d des_pose = p_map_to_object + offset;
  find_des_pose = true;

  // Set and send the goal
  move_base_msgs::MoveBaseGoal goal;
  goal.target_pose.header.frame_id = "map";
  goal.target_pose.header.stamp = ros::Time::now();
  tf2::Quaternion orientation_quat;
  orientation_quat.setRPY( 0, 0, 180.0*toRadians);
  goal.target_pose.pose.position.x = des_pose[0];
  goal.target_pose.pose.position.y = des_pose[1];
  goal.target_pose.pose.orientation.x = orientation_quat[0];
  goal.target_pose.pose.orientation.y = orientation_quat[1];
  goal.target_pose.pose.orientation.z = orientation_quat[2];
  goal.target_pose.pose.orientation.w = orientation_quat[3];
  ROS_INFO("Sending goal");
  ac.sendGoal(goal);

  ac.waitForResult();

  if(ac.getState() == actionlib::SimpleClientGoalState::SUCCEEDED){
    ROS_INFO("Hooray, the base moved");
  }
  else{
    ROS_INFO("The base failed to move for some reason");
  }
}
```

Once obtained the pose of the marker a new goal is set to reach the desired position.

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

To publish the ArUco pose as TF we use the class TransformBroadcaster. The following code handle the creation and sending of the Tf. All the field of the ‘tf::StampedTransform aruco_pose’ are filled with the right data, in fact a conversion from a rotation matrix to a quaternion was necessary to represent the orientation of the ArUco marker.

```
// ARUCO TF broadcast
aruco_pose_tf.stamp_ = ros::Time::now();
aruco_pose_tf.frame_id_ = "map";
aruco_pose_tf.child_frame_id_ = "aruco_pose";
Eigen::Quaterniond quat_map_to_object(rot_map_to_object);
tf::Quaternion quat_map_to_object_tf(quat_map_to_object.x(), quat_map_to_object.y(), quat_map_to_object.z(), quat_map_to_object.w());
aruco_pose_tf.setOrigin({p_map_to_object[0], p_map_to_object[1], p_map_to_object[2]});
aruco_pose_tf.setRotation(quat_map_to_object_tf);
broadcaster.sendTransform(aruco_pose_tf);
```