Report – Homework 4

Students:

Luca Marseglia

https://github.com/marseluca/homework4-rl.git

Benito Vodola

https://github.com/BenitoVodola/homework_4.git

Domenico Tuccillo

https://github.com/DomenicoTuccillo/homework4.git

Debora Ippolito

https://github.com/DeboraIppolito/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, y = 5, yaw = −90 deg with respect to the map frame. The argument for the yaw in the call of spawn_model is Y.

We modified “spawn_fra2mo_gazebo.launch” in the following way:

- We modified the default values for the x_pose and y_pose arguments as to match the desired position.
- We added a new argument “yaw” and set the default value as −1.57, we then passed it to the urdf_spawner node as for all the other arguments.

```
<!-- Modifichiamo la posa del robot -->
<arg name="x_pos" default="-3.0"/>
<arg name="y_pos" default="5.0"/>
<arg name="z_pos" default="0.1"/>
<!-- Inseriamo un nuovo argomento per l'orientamento Yaw -->
<arg name="yaw" 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 -->
<!-- Passiamo il nuovo argomento -->
<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"/>
```

(b) Modify the world file of rl_racefield moving the obstacle 9 in position x = −17, y = 9, z = 0.1, yaw = 3.14

We modified the pose of obstacle 9 in the rl_race_field.world file:

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

(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.

Firstly, we downloaded the svg file of the marker from the linked website and converted it to a png file. We then created the sdf model of the marker:

```xml
<?xml version="1.0" ?>
<sdf version="1.4">
  <model name="marker_new">
    <static>true</static>
    <link name="marker">
      <gravity>false</gravity>
      <visual name="tag">
        <geometry>
          <box>
            <size>0.1 0.1 0.002</size>
          </box>
        </geometry>
        <material>
          <script>
            <uri>model://marker_new/material/scripts</uri>
            <uri>model://marker_new/material/textures</uri>
            <name>marker_new</name>
          </script>
        </material>
      </visual>
      <visual name="support">
        <geometry>
          <box>
            <size>0.12 0.12 0.001</size>
          </box>
        </geometry>
        <material>Gazebo/White</material>
      </visual>

      <collision name="collision">
        <geometry>
          <box>
            <size>0.32 0.32 0.001</size>
          </box>
        </geometry>
      </collision>
    </link>
  </model>
</sdf>
```

In marker_new/material/scripts we added a script that loads the png file as the texture of the marker, the picture is stored in marker_new/material/textures.

Lastly, we added the marker to the rl_race_field.world in the desired pose:

```xml
<!-- AR markers -->
<include>
  <name>tool_0_tag</name>
  <uri>model://marker_new</uri>
  <pose>-17 8.2 0.2 0 1.57 3.14</pose>
</include>
```

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

We added static publishers in the spawn_fra2mo_gazebo.launch file for the goals, the conversion from RPY angles to quaternion was done using an online tool.

```xml

<!--Static tf publisher for goal-->
<!-- Definiamo nodi publisher per pubblicare le trasformazioni tra frame -->
<node pkg="tf" type="static_transform_publisher" name="goal_1_pub" args="-10 3 0 0 0 0 1 map goal1 100" />
<node pkg="tf" type="static_transform_publisher" name="goal_2_pub" args="-15 7 0 0 0 0.258819 0.9659258 map goal2 100" />
<node pkg="tf" type="static_transform_publisher" name="goal_3_pub" args="-6 8 0 0 0 1 0 map goal3 100" />
<node pkg="tf" type="static_transform_publisher" name="goal_4_pub" args="-17.5 3 0 0 0 0.6087614 0.7933533 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.

Following the reasoning already implemented in the code, we created new private variables in the TF_nav class to store the poses of the goals, which have been also initialized in the constructor:

```cpp
// Aggiunte variabili contenenti le tf
Eigen::Vector3d _goal1_pos;
Eigen::Vector4d _goal1_or;

Eigen::Vector3d _goal2_pos;
Eigen::Vector4d _goal2_or;

Eigen::Vector3d _goal3_pos;
Eigen::Vector4d _goal3_or;

Eigen::Vector3d _goal4_pos;
Eigen::Vector4d _goal4_or;
```

We also defined new listener functions that for each goal wait for the tf transform and then store it in the corresponding variables:

```cpp
class TF_NAV {

    public:
        TF_NAV();
        void run();
        void tf_listener_fun();
        void position_pub();
        // Aggiunti listener
        void goal_listener_1();
        void goal_listener_2();
        void goal_listener_3();
        void goal_listener_4();
        void send_goal();
```

```cpp
void TF_NAV::goal_listener_1() {
    ros::Rate r( 1 );
    tf::TransformListener listener;
    tf::StampedTransform transform;

    while ( ros::ok() )
    {
        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;
        }

        _goal1_pos << transform.getOrigin().x(), transform.getOrigin().y(), transform.getOrigin().z();
        _goal1_or << transform.getRotation().w(),  transform.getRotation().x(), transform.getRotation().y(), transform.getRotation().z();

        // Debug Print
        ROS_INFO("Goal Position: %f %f %f", _goal1_pos[0], _goal1_pos[1], _goal1_pos[2]);
        ROS_INFO("Goal Orientation: %f %f %f %f", _goal1_or[0], _goal1_or[1], _goal1_or[2], _goal1_or[3]);

        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.

We used an ActionClient to send a move_base_msgs::MoveBaseGoalmessage containing the pose of the goal to move_base. For each goal we created a new ActionClient and modified the content of the message. After the robot reaches one goal, the next one is sent.

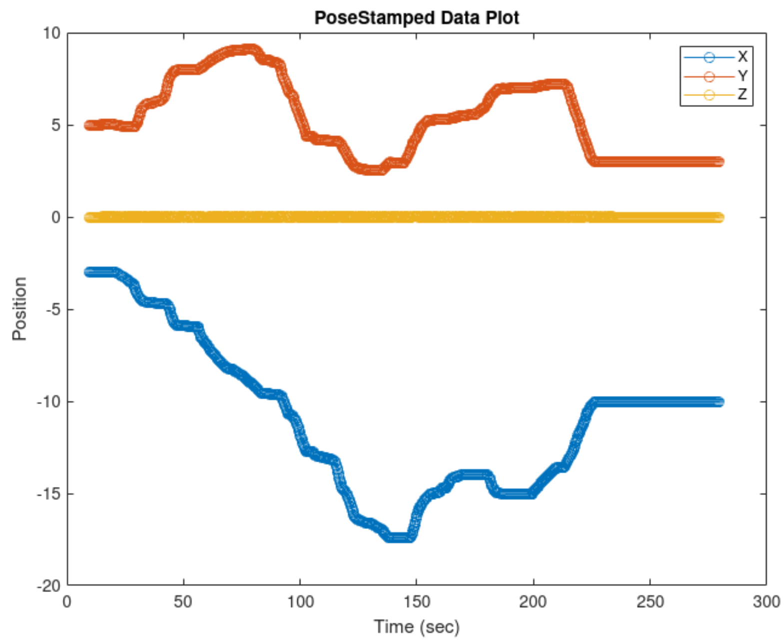
Then following the example provided to us we created the corresponding threads in the TF_NAV::run function in order to listen to the goals.

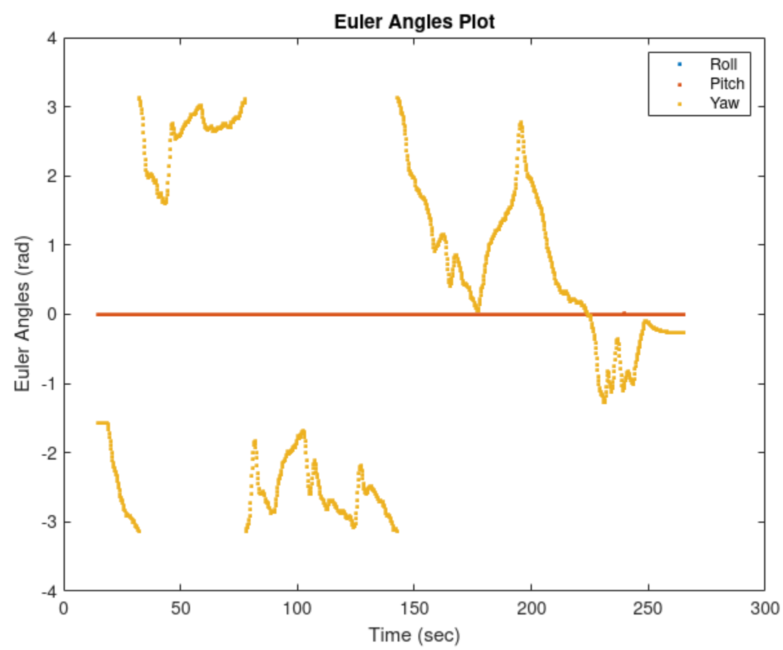
```
void TF_NAV::run() {
    boost::thread tf_listener_fun_t( &TF_NAV::tf_listener_fun, this );
    boost::thread broadcast_listener_t( &TF_NAV::broadcast_listener, this );
    boost::thread tf_listener_goal1_t( &TF_NAV::goal_listener_1, this );
    boost::thread tf_listener_goal2_t( &TF_NAV::goal_listener_2, this );
    boost::thread tf_listener_goal3_t( &TF_NAV::goal_listener_3, this );
    boost::thread tf_listener_goal4_t( &TF_NAV::goal_listener_4, this );
    boost::thread tf_listener_goal5_t( &TF_NAV::goal_listener_5, this );
    boost::thread tf_listener_goal6_t( &TF_NAV::goal_listener_6, this );
    boost::thread tf_listener_goal7_t( &TF_NAV::goal_listener_7, this );
    boost::thread tf_listener_goal8_t( &TF_NAV::goal_listener_8, this );
    boost::thread send_goal_t( &TF_NAV::send_goal, this );
    ros::spin();
}
```

We created a rosbag from the terminal recording the /fra2mo/pose topic which publishes the current pose of the robot:

```
debora@debora-Vivobook-ASUSLaptop-X1502ZA-F1502ZA:~/catkin_ws$ source devel/setup.bash
debora@debora-Vivobook-ASUSLaptop-X1502ZA-F1502ZA:~/catkin_ws$ rosbag record /fra2mo/pose
[ INFO] [1702541668.645923801]: Subscribing to /fra2mo/pose
[ INFO] [1702541668.941245733, 43.436000000]: Recording to '2023-12-14-09-14-28.bag'.
```

We used a MATLAB script to plot the results:





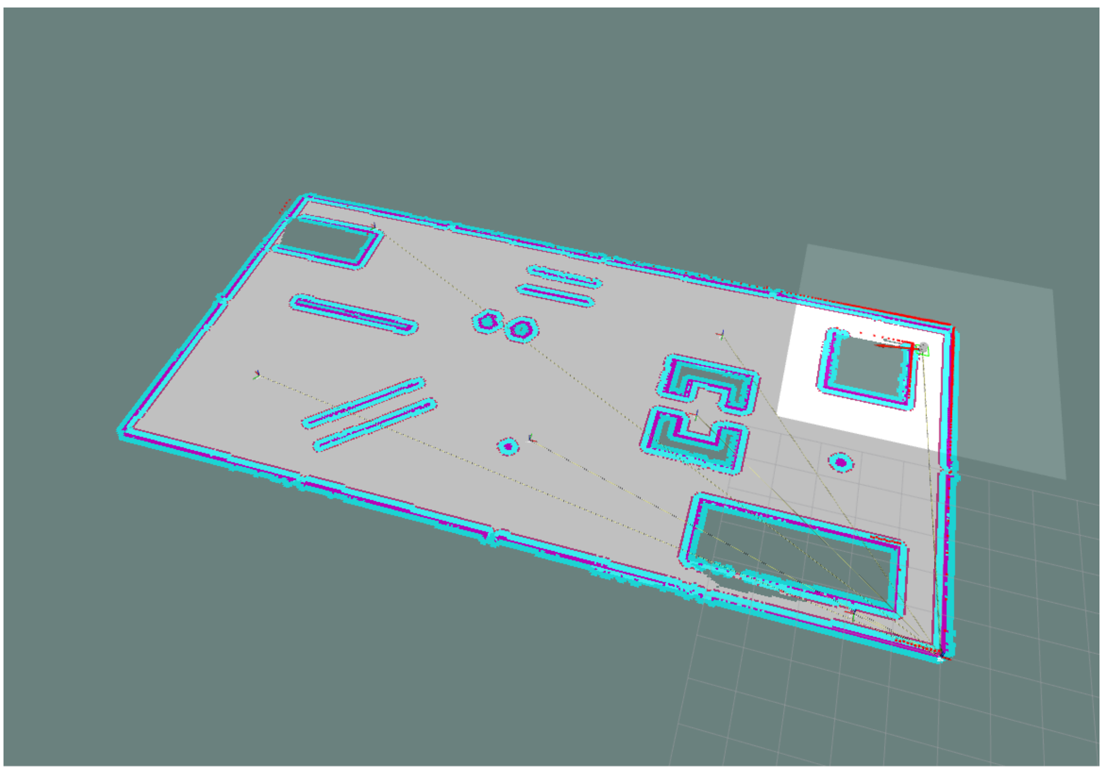
3. Map the environment tuning the navigation stack's parameters

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

We modified the 2nd goal and added three new ones:

```
<!--Static tf publisher for goal-->
<!-- Definiamo nodi publisher per pubblicare le trasformazioni tra frame -->
<node pkg="tf" type="static_transform_publisher" name="goal_1_pub" args="-10 3 0 0 0 0 1 map goal1 100" />
<node pkg="tf" type="static_transform_publisher" name="goal_2_pub" args="-17 9.5 0 0 0 0.965926 0.258819 map goal2 100" />
<node pkg="tf" type="static_transform_publisher" name="goal_3_pub" args="-6 8 0 0 0 1 0 map goal3 100" />
<node pkg="tf" type="static_transform_publisher" name="goal_4_pub" args="-17.5 3 0 0 0 0.965926 0.258819 map goal4 100" />
<node pkg="tf" type="static_transform_publisher" name="goal_5_pub" args="-6.144480 4.990364 0 0 0 1 0 map goal5 100" />
<node pkg="tf" type="static_transform_publisher" name="goal_6_pub" args="-1.831288 0.470925 0 0 0 1 0 map goal6 100" />
<node pkg="tf" type="static_transform_publisher" name="goal_7_pub" args="-0.70 9 0 0 0 1 0 map goal7 100" />
```

Exploring these goals in the order: Goal 3 →Goal 4 → Goal 2 → Goal 1 →Goal 5 → Goal 6 → Goal 7 the map is completely explored:



(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.

These are the parameters that we changed and the differences we noticed:

- obstacle_range from 7.0 to 2.0 --> the robot needs to be closer to the obstacle to detect it
- min_obstacle_dist: from 0.1 to 0.5 --> the robot tries to stay further away from the obstacles, it fails to reach some of the goals that are in narrower spaces.
- Max_global_plan_lookhead_dist from 2 to 7 --> the trajectory is planned looking further ahead.
- Max_vel_x from 0.6 to 1.5 --> obviously the robot moves faster.
- Xy_goal_tolerance from 0.15 to 1--> the robot stops further away from the goal frame, it considers the goal reached as soon as it enters the tolerance radius.
- Height and width in local costmap, from 7 to 12 --> the local trajectory is planned using a bigger map, this helps with avoiding the obstacles.

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.

We uncommented the camera lines in the fra2mo.xacro

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

We then modified the usb_cam.aruco.launch file used in the previous homework, in particular:

- The id_marker is 115
- The camera topic is depth_camera/depth_camera/image_raw
- The reference_frame is map since we want to retrive the pose of the aruco with respect to the map frame
- The camera_frame is camera_depth_optical_frame as can be seen from the xacro of the camera

```
<launch>

    <arg name="markerId"        default="115"/>
    <arg name="markerSize"      default="0.1"/>    <!-- in m -->
    <arg name="camera"          default="depth_camera/depth_camera"/>
    <arg name="marker_frame"    default="aruco_marker_frame"/>
    <arg name="ref_frame"       default="map"/>   <!-- vogliamo la posa rispetto al map frame-->
    <arg name="corner_refinement" default="LINES" />

    <node pkg="aruco_ros" type="single" name="aruco_single">
        <remap from="/camera_info" to="/$(arg camera)/camera_info" />
        <remap from="/image" to="/$(arg camera)/image_raw" />
        <param name="image_is_rectified" value="True"/>
        <param name="marker_size"        value="$(arg markerSize)"/>
        <param name="marker_id"          value="$(arg markerId)"/>
        <param name="reference_frame"    value="$(arg ref_frame)"/>   <!-- frame in which the marker pose will be refered -->
        <param name="camera_frame"       value="camera_depth_optical_frame"/> <!-- frame ricavato dall'urdf della camera-->
        <param name="marker_frame"       value="$(arg marker_frame)" />
        <param name="corner_refinement"  value="$(arg corner_refinement)" />
    </node>

</launch>
```

(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 = xm + 1, y = ym , where x m , y m are the marker coordinates.***

We created a new goal, close to the aruco and sent it to the robot as we did for the previous point.

```
<!-- goal per portare il robot vicino al marker -->
<node pkg="tf" type="static_transform_publisher" name="goal_8_pub" args="-16 8.5 0 0 0 1 0 map goal8 100" />
```

When the aruco is detected by the node its pose is published on the aruco_single/pose topic. Therefore, we created a subscriber to this topic, the callBackFunction simply stores the retrived pose on a vector called aruco_pose and prints it.

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

```cpp
void arucoPoseCallback(const geometry_msgs::PoseStamped & msg)
{
    //salviamo la posa letta nel vettore aruco_pose
    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);

    // ROS_INFO("Posizione (x, y, z): %.2f, %.2f, %.2f", aruco_pose[0], aruco_pose[1], aruco_pose[2]);
    // ROS_INFO("Orientazione (x, y, z, w): %.2f, %.2f, %.2f, %.2f", aruco_pose[3], aruco_pose[4], aruco_pose[5], aruco_pose[6]);

}
```

Once the robot arrived in the proximity of the aruco marker and retrieved its pose, we defined a new goal with the desired pose.

```cpp
            //Una volta arrivato lo facciamo muovere a xm+1 ym
            std::cout<<"\nInsert 1 to send aruco goal  "<<std::endl;
            std::cout<<"Inser your choice"<<std::endl;
            std::cin>>a_cmd;
            if ( a_cmd == 1) {
                MoveBaseClient ac_a("move_base", true);
                while(!ac_a.waitForServer(ros::Duration(5.0))){
                ROS_INFO("Waiting for the move_base action server to come up");
                }
                goal.target_pose.pose.position.x = aruco_pose[0]+1;
                goal.target_pose.pose.position.y = aruco_pose[1];
                goal.target_pose.pose.position.z = _goal8_pos[2];

                goal.target_pose.pose.orientation.w = _goal8_or[0];
                goal.target_pose.pose.orientation.x = _goal8_or[1];
                goal.target_pose.pose.orientation.y = _goal8_or[2];
                goal.target_pose.pose.orientation.z = _goal8_or[3];

                ROS_INFO("Sending goal aruco");
                ac_a.sendGoal(goal);

                ac_a.waitForResult();

                if(ac_a.getState() == actionlib::SimpleClientGoalState::SUCCEEDED)
                    ROS_INFO("The mobile robot arrived in the TF aruco goal");
                else
                    ROS_INFO("The base failed to move for some reason");


            }else{ROS_INFO("Wrong input!");}
```

(c) Publish the ArUco pose as TF

Following the example of the linked website, we added to our node a new callback where we defined a static tf broadcaster and a tf transform variable in which we saved the aruco marker pose. Then we sent it in broadcast with the method sendBroadcast.

To run this callback we defined a new subscriber in tf_nav main, in this way, running the tf_nav node we will also run the broadcaster.

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

```
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;
    q.setX(msg.pose.orientation.x);
    q.setY(msg.pose.orientation.y);
    q.setZ(msg.pose.orientation.z);
    q.setW(msg.pose.orientation.w);
    transform.setRotation(q);
    br.sendTransform(tf::StampedTransform(transform, ros::Time::now(), "map", "aruco_frame"));
}
```

Then, to verify if the message has been broadcasted, we added a new listener to receive the message and launched a thread in the TF_NAV::run function(as we did for the previous listeners).

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

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

        }
        catch( tf::TransformException &ex ) {
            ROS_ERROR("%s", ex.what());
            r.sleep();
            continue;
        }

        ROS_INFO("Aruco broadcasted pose: %f, %f, %f, %f, %f, %f, %f", transform.getOrigin().x(), transfo
        r.sleep();
    }

}
```

```
boost::thread broadcast_listener_t( &TF_NAV::broadcast_listener, this );
```

<u>On the repos you'll find the video for simulation of point 2 and 4.</u>