Robotics Lab: Homework 4
Control a mobile robot to follow a trajectory

Ludovica Ruggiero P38000207

repository: ludruggiero/homework4
videos: videos.md
collaborators: Luigi Catello, Stefano Covone, Cristiana Punzo,

Contents

The task in this homework is to control a mobile robot through an autonomous navigation software framework.

1 Spawning the world

1.a Spawning the robot

To change the spawning position and orientation of the robot, the `rl_fra2mo_description/spawn_fra2mo_gazebo.launch` file was modified. The already existing `x_pos` and `y_pos` args were set to -3.0 and 5.0 respectively. A new `yaw` argument was created and set to -1.57.

```
<arg name="x_pos" default="-3.0"/>
<arg name="y_pos" default="5.0"/>
<arg name="z_pos" default="0.1"/>
<!-- yaw angle-->
<arg name="yaw" default="-1.57"/>
```

The spawner node was modified, adding the yaw parameter:

```
<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"/>
```

1.b Modifying the obstacle position

The modification of the obstacle position was straightforward. Once identified the respective `<include>` block in the `rl_racefield/rl_race_field.world`, the position was set to -17, 9, and 0.1, with yaw equals to 3.14.

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

1.c Adding the ArUco Marker

The material, configuration and SDF files for the ArUco Marker object were taken from the previous homework. The marker image was generated on the ArUco Gen website using the Original ArUco dictionary. The resulting marker is shown in Figure 1.1.
Once the ArUco marker has been generated, it has been defined in the `rl_race_field.world` file, assigning it the name `aruco_marker` and including the model saved in the `model.sdf` file. The position of the ArUco has been chosen in such a way as to be above the obstacle modified at the previous point.

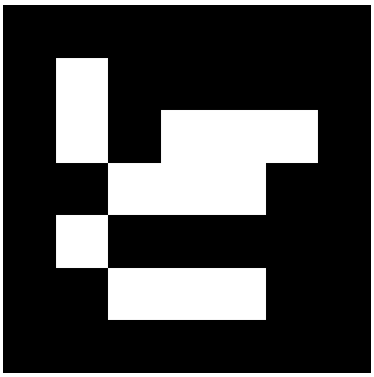


Figure 1.1: ArUco Marker 115

```
<include>
   <name>aruco_marker</name>
   <uri>model://aruco_marker</uri>
   <pose>-17 8.35851 0.451317 0 -0 0</pose>
</include>
```

2 Autonomous navigation

2.a Inserting the static goals

To insert the static goals, a new launch file was created. The new `rl_fra2mo_description/goal_fra2mo_gazebo.launch` launches the four `tf/static_transform_publisher` nodes, along with a parameter specifying the total number of goals to follow. This parameter will be useful later in Section 3.a.

```
<param name="numberOfGoals" value="4"/>
<node pkg="tf" type="static_transform_publisher" name="goal_1_publisher"
              args="-10 3 0.1 0 0 0 map goal_frame_1 100" />
<node pkg="tf" type="static_transform_publisher" name="goal_2_publisher"
              args="-15 7 0.1 0.523 0 0 map goal_frame_2 100" />
<node pkg="tf" type="static_transform_publisher" name="goal_3_publisher"
              args="-6 8 0.1 3.14 0 0 map goal_frame_3 100" />
<node pkg="tf" type="static_transform_publisher" name="goal_4_publisher"
              args="-17.5 3 0.1 1.309 0 0 map goal_frame_4 100" />
```

It is important to notice that, if the node is instantiated with the rotation expressed as roll, pitch and yaw, the yaw goes first[1]. Then, the launch file was included in `spawn_fra2mo_gazebo.launch`.

[1] Documentation here.

2.b Tf listeners

In order to achieve a flexible framework supporting any number of goals, the `std::vector` class was used to define a vector of transforms storing the goal frames.

```
void TF_NAV::goal_listener() {
    ros::Rate r( 1 );
    tf::TransformListener listener;
    std::vector<tf::StampedTransform*> transform;
    transform.resize(_totalNumberOfGoals);
    for (int i = 0; i < _totalNumberOfGoals; ++i) {
        transform[i] = new tf::StampedTransform();
    }
    ...
```

The vector is composed of pointers to `StampedTransform`. This is in order to dynamically increase the size of the vector. Since the `goal_listener()` function is set up to constantly run on its thread, a vector of flags was used to only print the debug information once:

```
    static std::vector<bool> hasLogged;
    hasLogged.resize(_totalNumberOfGoals);
    for (int i = 0; i < _totalNumberOfGoals; ++i) {
        hasLogged[i] = false;
    }
```

Then, as long as ROS is running, the thread loops through each of the `_totalNumberOfGoals` goals. The listener waits for the `goal_frame` to be available and saves its frame in the vector.

```
    listener.waitForTransform( "map", "goal_frame_" + std::to_string(goal_number + 1),
            ros::Time( 0 ), ros::Duration( 10.0 ) );
    listener.lookupTransform("map", "goal_frame_" + std::to_string(goal_number + 1),
            ros::Time( 0 ), *(transform[goal_number]));
```

The operation is performed in a try/catch block to handle possible exceptions. If it is the first time logging, the position and orientation of the goal transform are printed on the terminal.

```
    if (!hasLogged[goal_number]) {
        ROS_INFO("transform[%d]: pos (%f, %f, %f), rot (%f, %f, %f, %f)\n",
            goal_number,
             transform[goal_number]->getOrigin().x(),
             transform[goal_number]->getOrigin().y(),
             transform[goal_number]->getOrigin().z(),
             transform[goal_number]->getRotation().x(),
             transform[goal_number]->getRotation().y(),
             transform[goal_number]->getRotation().z(),
             transform[goal_number]->getRotation().w());
        hasLogged[goal_number] = true;
    }
```

Finally, the position and orientation are stored in the `_goal_pos` and `_goal_or` vectors defined in the class header file.

```
    _goal_pos.at(goal_number) <<
                        transform[goal_number]->getOrigin().x(),
                        transform[goal_number]->getOrigin().y(),
                        transform[goal_number]->getOrigin().z();
    _goal_or.at(goal_number) <<
                        transform[goal_number]->getRotation().w(),
                        transform[goal_number]->getRotation().x(),
                        transform[goal_number]->getRotation().y(),
                        transform[goal_number]->getRotation().z();
```

2.c Moving towards the goals

Since the goals needed to be reached in a given order (*Goal_3* $\rightarrow$ *Goal_4* $\rightarrow$ *Goal_2* $\rightarrow$ *Goal_1*), an array containing the ordered indexes was defined in the class header file:

```
#define NUM_GOALS 4
...
int goalOrder[NUM_GOALS] = {3, 4, 2, 1};
```

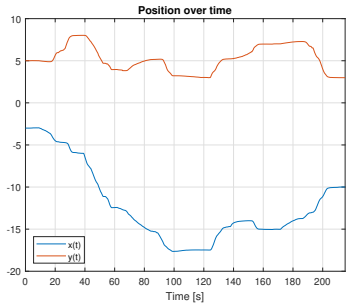
The core of the goal sending mechanism was implemented in the `TF_NAV::send_goal()` function, which is executed on its own thread. The main loop, running `while (ros::ok())`, first asks the user to choose between three options:

1. Send the goals set in the launch file;
2. Send the spawn position of the robot as the goal;
3. Send the approximate position of the ArUco Marker as goal.

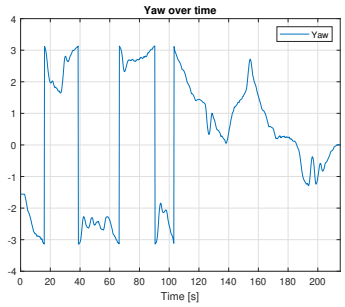
The last option will be useful in Section 4.b. For what concerns the aim of this section, only option 1 will be described.
After having waited for the `move_base` action server to be ready, the function loops through each of the `_totalNumberOfGoals` goals, setting the value of the `move_base_msgs::MoveBaseGoal goal;` object according to the correct target. Since in Section 3.a it is required to explore the map using different goals, a flag `_allowExploration` is defined in the class header file and set in the class constructor, its value taken from the launch file. Then, according to whether the robot is in *exploration mode* or not, the goals are sent in the same order as the launch file or in the order specified in `goalOrder`.

```
for (int goal_index = 0; goal_index < _totalNumberOfGoals; ++goal_index) {
    goal.target_pose.header.frame_id = "map";
    goal.target_pose.header.stamp = ros::Time::now();

    if (!_allowExploration) {
```



(a) Robot position plot



(b) Robot yaw plot

Figure 2.1: Plots of position and rotation

```
        // if not in exploration mode, the order of the goals is the one
        //specified in class header
        goal.target_pose.pose.position.x = _goal_pos.at(goalOrder[goal_index] - 1)[0];
        ...
        goal.target_pose.pose.orientation.z = _goal_or.at(goalOrder[goal_index] - 1)[3];
        ROS_INFO("Sending goal %d", goalOrder[goal_index]);
    } else {
        // if in exploration mode, the order is 1, ..., _totalNumberOfGoals
        goal.target_pose.pose.position.x = _goal_pos.at(goal_index)[0];
        ...
        goal.target_pose.pose.orientation.z = _goal_or.at(goal_index)[3];
        ROS_INFO("Sending goal %d", goal_index);
    }
    ac.sendGoal(goal);

    ac.waitForResult();
    if (ac.getState() == actionlib::SimpleClientGoalState::SUCCEEDED) {
        // conditional operator to print the correct message
        ROS_INFO("The mobile robot arrived in the TF goal number %d",
            (!_allowExploration) ? goalOrder[goal_index] : goal_index);
    } else {
        ROS_INFO("The base failed to move for some reason");
        continue;
    }
}
```

As shown in the above code each goal is sent to the `move_base` server, which commands the global planner to move the mobile robot to the desired location. The thread waits for the server for the results of the motion and, after an appropriate log on the ROS terminal, it moves to the next goal.

Simulation results The trajectory executed by the mobile robot was recorded in a bagfile and processed in an appropriately modified version of the MATLAB

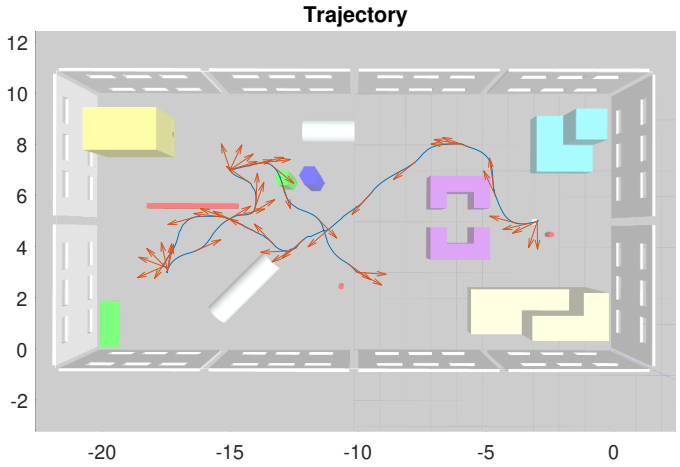


Figure 2.2: 2D trajectory of the robot following the 4 goals defined in Section 2.a

function already used in the last two homeworks. Along with the temporal evolution of robot position (Figure 2.1a) and orientation (Figure 2.1b), in Figure 2.2 is shown the 2D trajectory of the mobile robot on a X-Y plot. A semi-transparent picture of the gazebo environment is added to the plot for clarity.

The robot successfully reaches each of the 4 goals in the given order. Nevertheless, the obtained trajectory highlights the differential drive robot limitations: when performing a straight motion the robot tends to drift and it is poorly reactive to correct its orientation.

A screencast of the Gazebo simulation is available here.

3 Environment mapping

3.a Modifying the previous goals

Several goals were chosen ad hoc in order to explore the entire map. To pick suitable values for position orientation, the 2D Pose Estimate tool in RViz was used (Figure 3.1). It allows the user to input a certain position/orientation through the use of the mouse, then it publishes the information on the `/initialpose` topic. Using the `rostopic echo /initialpose` command, the exact values were retrieved and inserted in an appropriate `<node>` in the `goal_fra2mo_gazebo.launch` file. Since more than 4 goals are needed to explore the environment completely, the `numberOfGoals` parameter was specified. The `<param>` and `<node>` elements were gathered in a `group` tag.

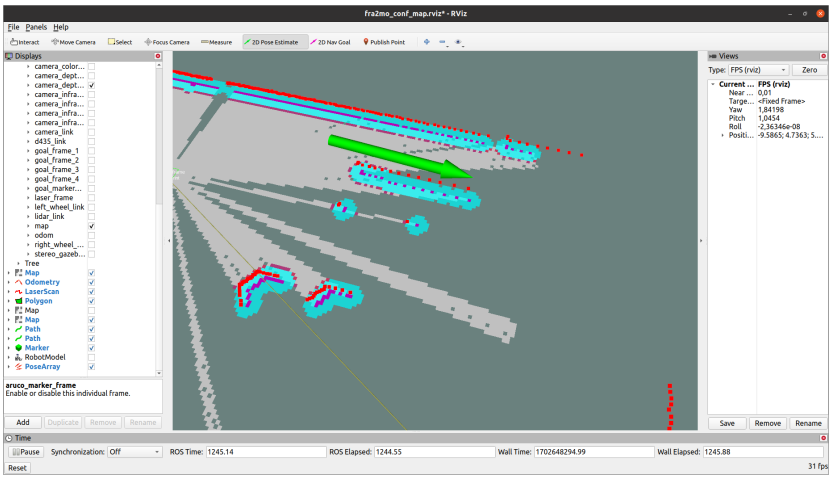


Figure 3.1: 2D Pose Estimation tool in RViz

```
<group if="$(arg AllowExplorationArg)">
    <param name="numberOfGoals" value="6"/>
    <node pkg="tf"
        type="static_transform_publisher"
        name="goal_1_publisher"
        args="-6.05 5.2 0.1 0 0 -1 0 map goal_frame_1 100" />
    ...
    <node pkg="tf"
        type="static_transform_publisher"
        name="goal_6_publisher"
        args="-0.49 0.49 0.1 0 0 -1 0.03 map goal_frame_6 100" />
</group>
```

After having reached each goal in the list, the explored map appeared as shown in Figure 3.2. A screencast of the Gazebo simulation is available here.

3.b Trying different parameters

The simulation in the previous section highlighted how the robot tended to stay too close to obstacles and, due to high speeds, exhibited drift in orientation.

Four different configuration have been tried in order to improve robot performance; each of them is discussed in the following paragraphs. Table 1 compares the values of each configuration.

Configuration 1 Starting from the default parameter configuration in the first attempt, the minimum distance from obstacles was increased to 0.2, and the zone with non-zero penalty around them was expanded to 0.35. Additionally, concerning the local costmap, its update frequency was increased, and

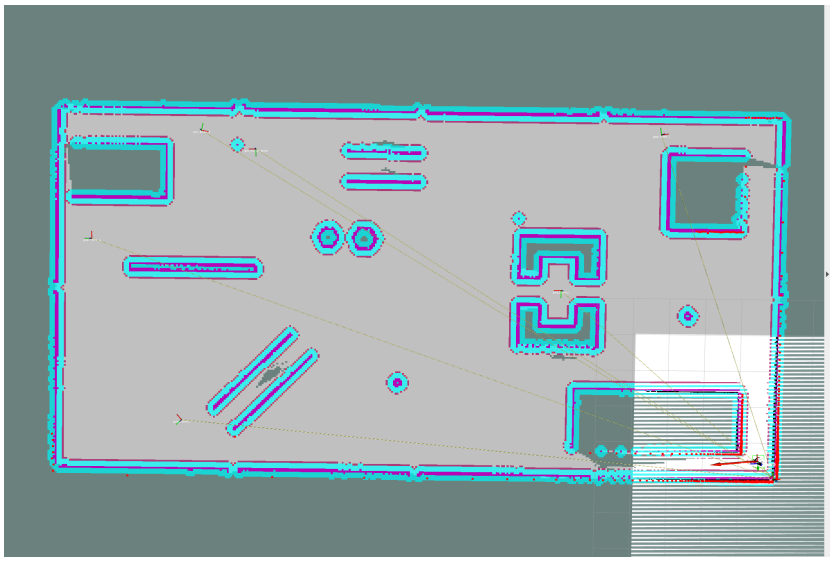


Figure 3.2: Fully explored map

File name	Parameter	Default	C1	C2	C3	C4
teb_locl_planner_params	min_obstacle_dist	0.1	0.20	0.15	0.15	0.15
	inflation_dist	0.1	0.35	0.1	0.2	0.3
	max_vel_x	0.6	0.6	0.4	0.3	0.4
	dt_ref	0.5	0.5	0.5	0.8	0.6
	acc_lim_x	0.1	0.1	0.05	0.05	0.1
local_costmap_params	update_frequency	5.0	10.0	7.0	7.0	8.0
	resolution	0.03	0.01	0.01	0.01	0.01
global_costmap_params	max_vel_x	0.6	0.6	0.4	0.3	0.4
	publish_frequency	2.0	7.0	7.0	7.0	10.0
	update_frequency	5.0	7.0	7.0	7.0	15.0
costmap_common_params	obstacle_range	7.0	7.0	10.0	7.0	7.0
	raytrace_range	8.0	8.0	11.0	8.0	8.0

Table 1: Comparison of four different configurations.

its resolution was improved. Similarly, adjustments were made to the update frequency of the global costmap.

Following the parameter modifications, the more restrictive constraints on obstacles prevented it from navigating through narrower spaces, despite allowing for a better obstacle avoidance.

Configuration 2 To improve the results of the previous simulation, the default value for the width of the zone with non-zero penalty was restored, and the minimum distance from obstacles was set to an intermediate value of 0.15. To reduce the robot's drift, the maximum values for velocity and acceleration were decreased. To avoid instability in costmap generation, a more conservative update frequency was chosen. Finally, to simulate the use of a better laser scan, the maximum viewing ranges were increased.

The use of a better sensor did not show significant improvements; on the contrary, reducing the robot's velocity improved its kinematic behavior.

Configuration 3 Given the promising results in terms of trajectory accuracy from the previous configuration, it was decided to further decrease the maximum values for velocity and acceleration, slightly tightening the constraints on obstacles. The characteristics of the laser sensor were reverted to default values, while the optimization interval for trajectory planning by the local planner was increased.

Following the simulation, the robot proved to be excessively slow in its movements, while still experiencing some difficulty in rotations around its axis. Moreover, increasing the optimization interval caused a peculiar phenomenon of trajectory instability, as depicted in Figure 3.3.

Configuration 4 As a final attempt to enhance the robot's performance without causing trajectory instability, it was decided to increase the frequencies of both local and global costmaps, expand the penalty zone around obstacles, and slightly raise the maximum velocities compared to the previous configuration.

The simulation demonstrated a more balanced behavior of the robot, although none of the attempts significantly improved its performance, particularly in terms of its ability to follow trajectories set by the local planner.

4 Vision-based navigation

4.a Activating robot camera

The D435 camera model is already included in the robot URDF provided within the `rl_fra2mo_description` package. Including it in the Gazebo simulation can be simply done by uncommenting the relative lines in the `fra2mo.xacro` file:

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

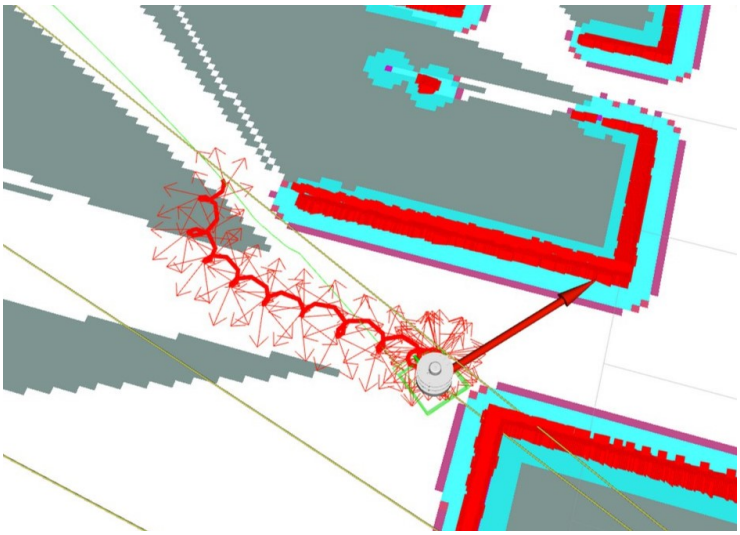


Figure 3.3: Higher values in the optimizator temporal distance resulted in trajectory instability, where strange rotations were imposed to the mobile robot

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

To enable the ArUco detection, the `aruco_ros` package was used. With respect to the package version published on the organization repository, it was only necessary to change the arguments in the `usb_cam_aruco` launchfile containing the ArUco marker number to detect and the robot camera name, respectively:

```
<arg name="markerId" default="115"/>
...
<arg name="camera" default="/depth_camera/depth_camera"/>
```

Once launched, when the ArUco marker is detected by the mobile robot camera, the node publishes its pose with respect to the camera frame on the `/aruco_single/pose` topic via `geometry_msgs::PoseStamped` messages. It is important to note that the ArUco pose is computed with respect to the frame `camera_depth_optical_frame`, which is both rotated and translated with respect to the `base_footprint` frame. The solution to this issue will be discussed in the following section.

Finally, to include both the `aruco_ros` node and the `rqt_image_view` to the Gazebo simulation, the following lines were added to the `fra2mo_nav_bringup.launch` file:

```
<include file="$(find aruco_ros)/launch/usb_cam_aruco.launch"/>
<node name="rqt_image_view" pkg="rqt_image_view" type="rqt_image_view" />
```

Figure 4.1: The ArUco marker is detected by the robot camera

When the ArUco marker is visible from the robot camera, its pose is effectively detected, as shown in Figure 4.1.

4.b 2D navigation task

Using the same technique as Section 3.a, a point reasonably close to the marker was chosen and added to the launch file. When the third mode listed in Section 2.c is selected, the robot is prompted to reach this new objective. As soon as the goal is reached and the robot is close enough to the ArUco marker, the `aruco_ros/single` node starts publishing the coordinates of the marker. This message is received and processed by the `TF_NAV` object.

Class constructor

The subscriber is defined in the class constructor:

```
_aruco_pose_sub = _nh.subscribe("/aruco_single/pose", 1,
                  &TF_NAV::arucoPoseCallback, this);
```

The class constructor also includes a logic to retrieve the transformation matrix between the robot `base_footprint` and the camera optical frame, with respect to which the marker pose is computed by `aruco_ros`. Since this transformation is fixed, it is sufficient to compute it just once, using the `tf_listener` package, and then store it in a private member data of the class:

```
// acquires the transformation matrix between camera and base_footprint
ros::Rate r( 5 );
```

```
tf::TransformListener listener;
tf::StampedTransform tfBaseCamera;

try {
    listener.waitForTransform( "base_footprint",
           "camera_depth_optical_frame", ros::Time(0), ros::Duration(10.0));
    listener.lookupTransform( "base_footprint",
           "camera_depth_optical_frame", ros::Time(0), tfBaseCamera );
} catch ( tf::TransformException &ex ) {
    ROS_ERROR("%s", ex.what());
    r.sleep();
    return;
}

_tfBaseCamera = tfBaseCamera;
```

Callback function

The callback function is a public member of the class. First it stores the position and orientation contained in the message.

```
void TF_NAV::arucoPoseCallback(const geometry_msgs::PoseStamped & msg) {
    tf::Vector3 ArucoPosition(msg.pose.position.x,
               msg.pose.position.y, msg.pose.position.z);
    tf::Quaternion ArucoOrientation(msg.pose.orientation.x,
               msg.pose.orientation.y, msg.pose.orientation.z,
               msg.pose.orientation.w);
```

Then it creates a `tf::Transform` object.

```
    tf::Transform tfCameraAruco =
           tf::Transform(ArucoOrientation, ArucoPosition);
```

The ArUco pose is computed with respect to the `camera_depth_optical_frame`, so in order to retrieve its pose in the map frame the following transformation is performed, where `_tfBase` stores the pose of the robot `base_footprint` with respect to the `map` frame:

```
    // aruco wrt world frame
    _tfAruco = _tfBase * _tfBaseCamera * tfCameraAruco;
```

The pose of each frame is shown in Figure 4.2: the marker frame is oriented differently from the robot base footprint, so in the following will be necessary to compute the rotation matrix between the to frames to properly assign a goal with respect to the marker position.
The callback function also includes a `tf_broadcaster`, whose implementation will be discussed in Section 4.c.

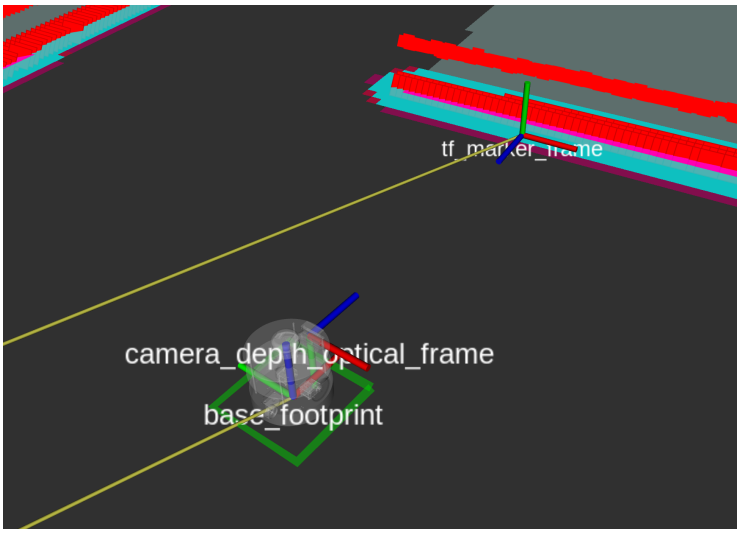


Figure 4.2: Rviz representation of robot base, camera and ArUco marker frames

goal_listener

The `goal_listener` function now includes a try catch block to acquire the goal discussed above from the `/tf` topic. The goal position and orientation are then stored in private members of the class:

```
try {
    listener.waitForTransform( "map", "goal_marker_frame",
                      ros::Time( 0 ), ros::Duration( 10.0 ));
    listener.lookupTransform("map", "goal_marker_frame",
                      ros::Time( 0 ), tfGoalAruco);
    _aruco_goal_pos << tfGoalAruco.getOrigin().x(),
        tfGoalAruco.getOrigin().y(), tfGoalAruco.getOrigin().z();
    _aruco_goal_or << tfGoalAruco.getRotation().w(),
        tfGoalAruco.getRotation().x(), tfGoalAruco.getRotation().y(),
        tfGoalAruco.getRotation().z();
} catch( tf::TransformException &ex ) {
    ROS_ERROR("goal_aruco %s", ex.what());
    r.sleep();
    continue;
}
```

This goal is always listened by the function, so the robot would be able to exploit the vision-based task also after the tasks defined in the previous sections.

tf_listener_fun

The only change to `tf_listener` consists in storing the `base_footprint` pose in a `tf::Transform` private member, in order to perform matrix operations

between frames:

```
listener.waitForTransform( "map", "base_footprint", ros::Time(0),
                                        ros::Duration(10.0) );
listener.lookupTransform( "map", "base_footprint", ros::Time(0),
                                        transform );
...
// store the tf matrix of base footprint into object data
_tfBase = transform;
```

send_goal

The `send_goal` function now allows the user to select a third operative mode, where the robot performs the vision based task. Firstly, the goal near the ArUco marker, acquired by the `goal_listener` thread, is sent to the `move_base` client:

```
// First, send the robot near the Aruco marker
goal.target_pose.header.frame_id = "map";
goal.target_pose.header.stamp = ros::Time::now()

goal.target_pose.pose.position.x = _aruco_goal_pos[0];
...
goal.target_pose.pose.orientation.z = _aruco_goal_or[3];
```

The robot goes nearby the marker, ah shown in Figure 4.1. At this point, it is able to retrieve the marker pose and convert it into the map frame via the `arucoPoseCallback`.

Now, as requested, the robot has to move 1 meter in front of the ArUCo marker. With respect to the marker frame, the goal position is described by the transformation matrix:

$$T_{OFF} = \begin{bmatrix} 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & -0.15 \\ -1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1}$$

Where the first 3×3 block represents the rotation matrix between the marker frame and the desired robot base footprint frame, and the 3×1 vector in the fourth column represents the desired robot offset from the marker. The desired robot pose in world frame is obtained as:

$$T_{DES} = T_{ARUCO} * T_{OFF} \tag{2}$$

The C++ implementation is shown in the following code, where it is important to notice that the robot first waits for 2 seconds in the previously reached goal, in order to compute the ArUco pose more reliably:

```
tf::Quaternion rotOffset(-0.5, 0.5, 0.5, 0.5);
tf::Vector3 posOffset(0,-_tfBaseCamera.getOrigin().z(),1);
tf::Transform tfOffset(rotOffset,posOffset)
```

```
    ros::Duration(2.0).sleep();

    tf::Transform tfDesiredPose = _tfAruco*tfOffset;
```

Finally, the desired goal is sent to the `move_base` client in order for the planner to compute the desired motion:

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

    goal.target_pose.pose.position.x = tfDesiredPose.getOrigin().x();
    goal.target_pose.pose.position.y = tfDesiredPose.getOrigin().y();
    goal.target_pose.pose.position.z = tfDesiredPose.getOrigin().z();

    goal.target_pose.pose.orientation.w =
                  std::round(tfDesiredPose.getRotation().w()*10)/10;
    goal.target_pose.pose.orientation.x =
                  std::round(tfDesiredPose.getRotation().x()*10)/10;
    goal.target_pose.pose.orientation.y =
                  std::round(tfDesiredPose.getRotation().y()*10)/10;
    goal.target_pose.pose.orientation.z =
                  std::round(tfDesiredPose.getRotation().z()*10)/10;
```

The desired orientation is rounded to the first decimal digit in order to avoid numerical errors on the z-axis, which may cause the planner to consider the trajectory unfeasibile.

Finally, the thread waits for the robot to execute its motion successfully and then it terminates.

Simulation results As shown in Figure 4.3 the robot moves in front of the marker, but the actual distance from it is affected by some error due to the fact that the robot camera is not well calibrated and tends to underestimate the marker distance from it.
A screencast of the Gazebo simulation is available here.

4.c Publishing the ArUco pose as tf

In order to visualize the ArUco marker pose in Rviz, it is useful to publish the transformation matrix computed by the `arucoPoseCallback` function using a `tf::TransformBroadcaster` object.

Following the ROS wiki, it is quite straightforward to include the broadcast in the callback function. First a `tf::TransformBroadcaster` object is defined, then the `_tfAruco` transform is sent to the `/tf` topic with the name `tf_marker_frame`:

```
static tf::TransformBroadcaster arucoTfBroadcaster;
arucoTfBroadcaster.sendTransform(
```

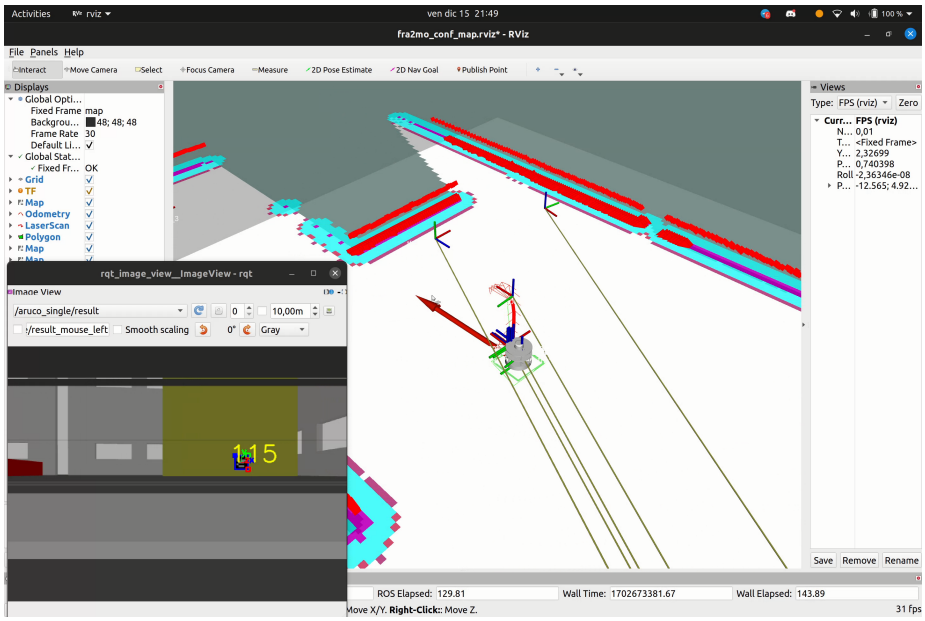


Figure 4.3: Once the robot reaches the ArUco marker, it positions itself 1 meter in front of it

```
        tf::StampedTransform(_tfAruco,
                             ros::Time::now(),
                             "map",
                             "tf_marker_frame"));
```

The same logic is also resumed in the `send_goal` function, where the goal in front of the marker is also published on `/tf` to visualize it in Rviz, as it is shown in Figure 4.3.