



UNIVERSITA' DEGLI STUDI DI  
NAPOLI FEDERICO II

Scuola Politecnica e delle Scienze di Base  
Corso di Laurea Magistrale in Ingegneria Informatica

Robotics Lab Project

## ***HOMEWORK 4***

Academic Year 2024/2025

Relatore

**Ch.mo prof. Mario Selvaggio**

Candidato

**Emmanuel Patellaro**

**P38000239**

Repository

[https://github.com/EmmanuelPat6/Homework\\_4.git](https://github.com/EmmanuelPat6/Homework_4.git)

# Abstract

This project focuses on implementing an Autonomous Navigation Software Framework to control a Mobile Robot. The **rl\_fra2mo\_description** package ([https://github.com/RoboticsLab2024/rl\\_fra2mo\\_description.git](https://github.com/RoboticsLab2024/rl_fra2mo_description.git)) must be used as a starting point for the simulation. First of all, a world in Gazebo was constructed in order to spawn the **Mobile Robot** in a given pose and then to spawn the **115 ArUco Marker** on an object in such a way to be visible by the **Camera Sensor** of the robot. Then the **Nav2 Simple Commander API** has been used to enable an **Autonomous Navigation Task** with subsequent **Mapping** of the surrounding environment. At the end, all the results have been commented in terms of **Robot Trajectories**, **Execution Timings**, **Map Accuracy** etc. etc. . At the end a **Vision-Based Navigation** of the Mobile Platform has been implemented.

# Contents

<b>Abstract</b>	<b>1</b>
<b>Gazebo World</b>	<b>1</b>
Mobile Robot . . . . .	1
Obstacle . . . . .	4
ArUco Marker . . . . .	6
<b>Autonomous Navigation Task</b>	<b>12</b>
Goals . . . . .	12
Implementation . . . . .	16
Bag File and Plot . . . . .	20
<b>Map of the Environment and Parameters Tuning</b>	<b>25</b>
Complete Map . . . . .	25
Parameters Tuning . . . . .	31
<b>Vision-Based Navigation</b>	<b>37</b>
Navigation . . . . .	37
ArUco Pose and TF . . . . .	49

Control a Mobile Robot to Follow a Trajectory

# Gazebo World

## Mobile Robot

First of all, the position of the **fra2mo Robot** was changed according to the project specifications. At the beginning, in the file **gazebo\_fra2mo.launch.py** there was

```
position = [0.0, 0.0, 0.100]

# Define a Node to spawn the robot in the Gazebo simulation
gz_spawn_entity = Node(
    package='ros_gz_sim',
    executable='create',
    output='screen',
    arguments=[ '-topic', 'robot_description',
                '-name', 'fra2mo',
                '-allow_renaming', 'true',
                "-x", str(position[0]),
                "-y", str(position[1]),
                "-z", str(position[2]), ]
```

## Running

```
ros2 launch rl_fra2mo_description gazebo_fra2mo.launch.py
```

the Robot will be spawn in the Origin of the Map.

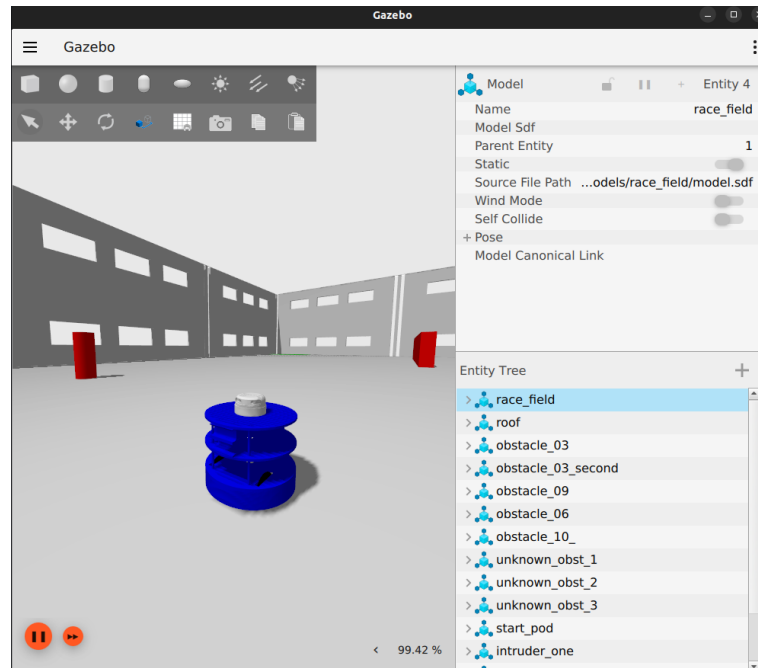
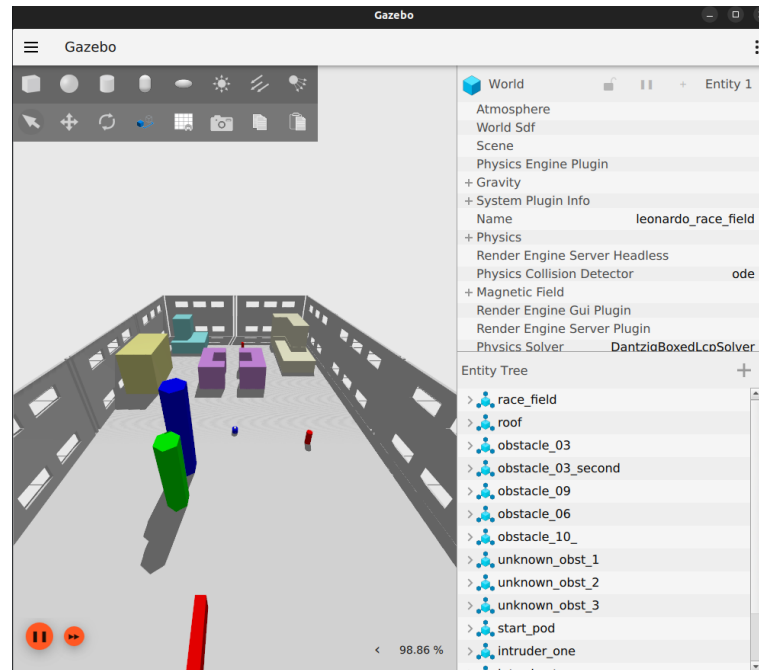


Figure 1: fra2mo Robot in Gazebo before Pose Modification

The goal was to spawn the robot with the following pose (with respect to the **Map Frame**)

$$x = -3m$$

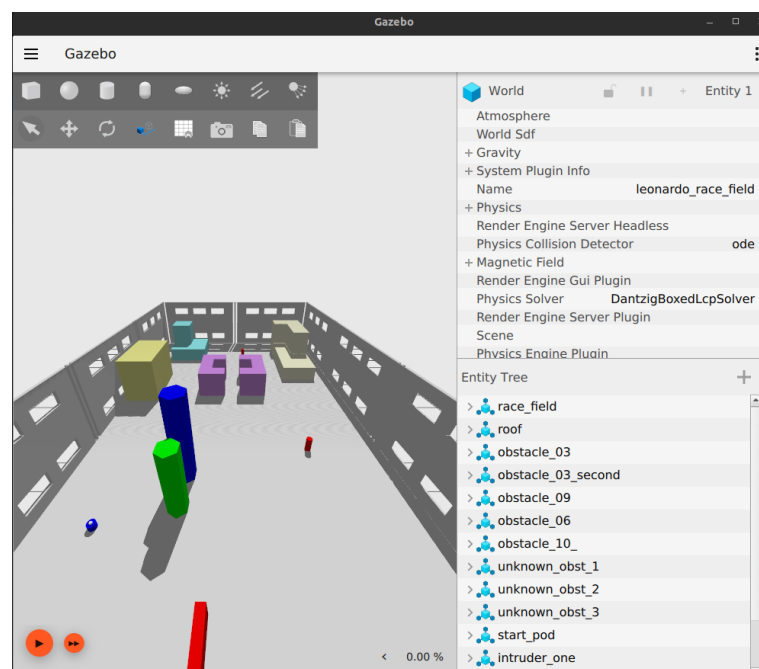
$$y = 3.5m$$

$$Y = -90deg$$

where **Y** is the **yaw** argument of **RPY Angles**. The previous code was modified into

```
position = [-3.0, 3.5, 0.100, -1.57]

# Define a Node to spawn the robot in the Gazebo simulation
gz_spawn_entity = Node(
    package='ros_gz_sim',
    executable='create',
    output='screen',
    arguments=[
        '-topic', 'robot_description',
        '-name', 'fra2mo',
        '-allow_renaming', 'true',
        "-x", str(position[0]),
        "-y", str(position[1]),
        "-z", str(position[2]),
        "-Y", str(position[3]),]
```



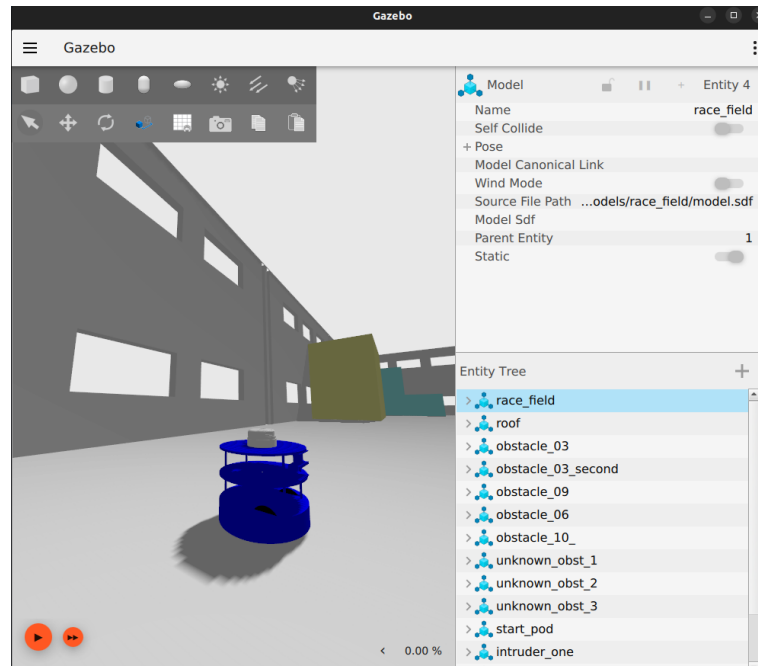


Figure 2: fra2mo Robot in Gazebo after Pose Modification

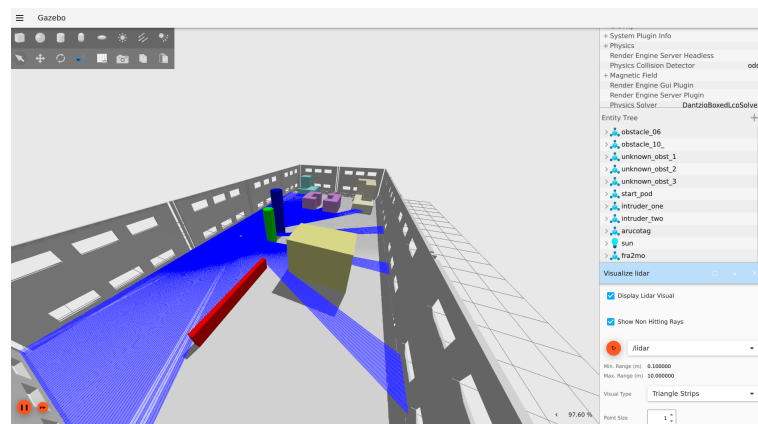


Figure 3: fra2mo Robot and Lidar

## Obstacle

Now, let's modify the world file `leonardo_race_field.sdf` moving the **Obstacle 9** in position

$$x = -3m$$

$$y = -3.3m$$



$$Y = 90deg$$

As before, it is necessary to go in the `.sdf` file written before and modify the part

```
<include>
  <name>obstacle_09</name>
  <pose> 5 5 0.1 0 0 3.14159</pose>
  <uri>model://obstacle_09</uri>
</include>
```

into

```
<include>
  <name>obstacle_09</name>
  <pose> -3 -3.3 0.1 0 0 1.57</pose>
  <uri>model://obstacle_09</uri>
</include>
```

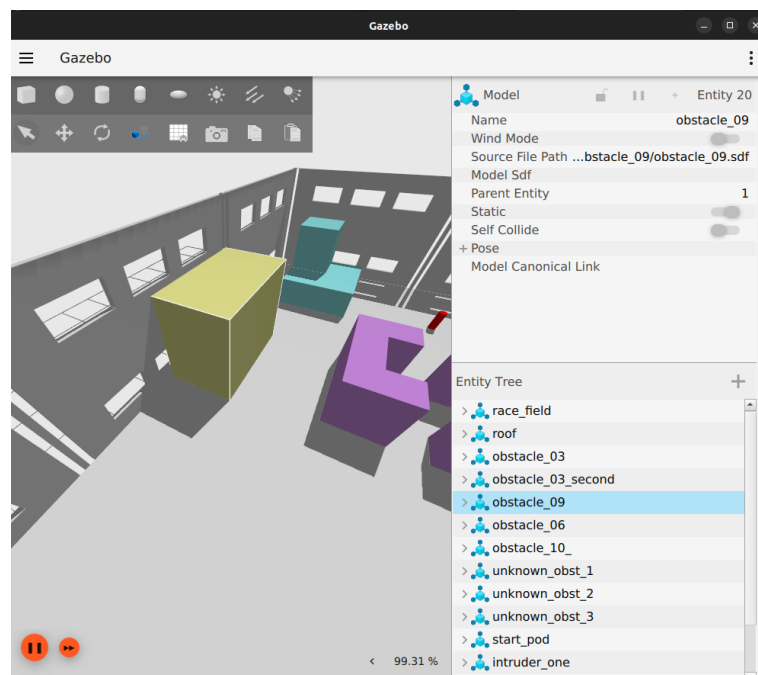


Figure 4: Obstacle 9 in Gazebo before Modification

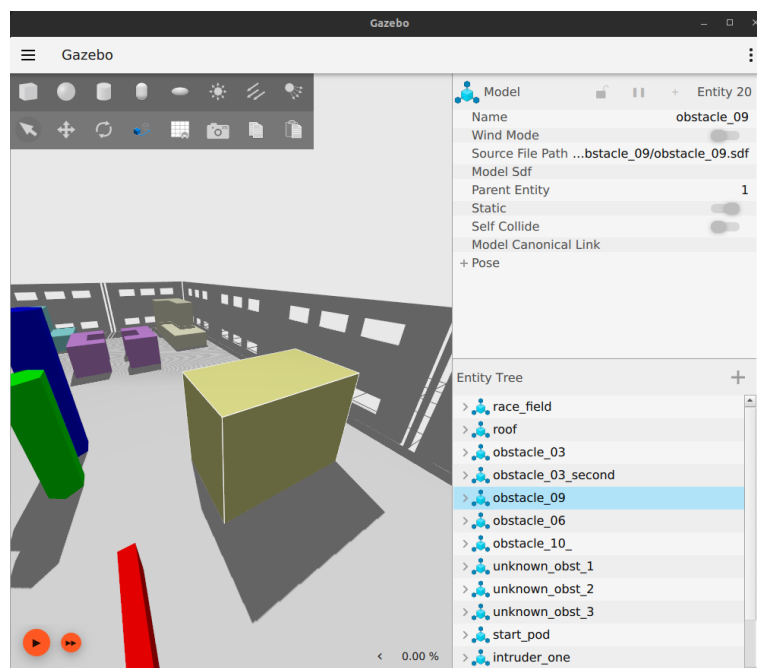


Figure 5: Obstacle 9 in Gazebo after Modification

## ArUco Marker

Now, all that remains is to add the **115 ArUco Tag** into the world, specifically on **Obstacle 9**. Before doing this, we add a **Camera Sensor** to the fra2mo robot to enable it to detect the ArUco Tag in the environment. To achieve this, the same steps shown in **Homework 1** and **Homework 3** have been done.

Let's put the Camera in the same position of the d435 Link (not used). Some files have been created:

**camera.xacro:**

```
<?xml version="1.0"?>

<robot xmlns:xacro="http://www.ros.org/wiki/xacro" name="camera">

  <link name="camera_link">
    <visual>
      <geometry>
        <box size="0.0000010_0.00000003_0.00000003"/>
      </geometry>
      <material name="red"/>
    </visual>
  </link>
</robot>
```

```

</link>

<joint name="camera_joint" type="fixed">
  <parent link="base_link"/>
  <child link="camera_link"/>
  <origin xyz="0.090925_0.0_0.06670" rpy="0_0_0" />
</joint>

</robot>

```

### **fra2mo\_camera.xacro**

```

<?xml version="1.0"?>

<robot xmlns:xacro="http://www.ros.org/wiki/xacro" name="camera_gaz">

  <!--
  <gazebo>
    <plugin filename="gz-sim-sensors-system"
      name="gz::sim::systems::Sensors">
      <render_engine>ogre2</render_engine>
    </plugin>
  </gazebo>
  -->

  <gazebo reference="camera_link">
    <sensor name="camera" type="camera">
      <camera>
        <horizontal_fov>1.047</horizontal_fov>
        <image>
          <width>640</width>
          <height>480</height>
        </image>
        <clip>
          <near>0.1</near>
          <far>100</far>
        </clip>
      </camera>
      <always_on>1</always_on>
      <update_rate>30</update_rate>
      <visualize>true</visualize>
      <topic>camera</topic>
    </sensor>
  </gazebo>

</robot>

```

In this file, the plugin has been commented because this was already included for the **Lidar Sensor** and they would have conflicted. In **fra2mo.urdf.xacro**

```

<xacro:include filename="$(find_ri_fra2mo_description)/urdf/camera.
  xacro"/>

```

```
<xacro:include filename="$(find_rl_fra2mo_description)/urdf/
  fra2mo_camera.xacro"/>
```

and in `gazebo_fra2mo.launch.py`

```
bridge_camera = Node(
    package='ros_ign_bridge',
    executable='parameter_bridge',
    arguments=[
        '/camera@sensor_msgs/msg/Image@gz.msgs.Image',
        '/camera_info@sensor_msgs/msg/CameraInfo@gz.msgs.CameraInfo',
        '--ros-args',
        '-r', '/camera:=/videocamera',
    ],
    output='screen',
)
```

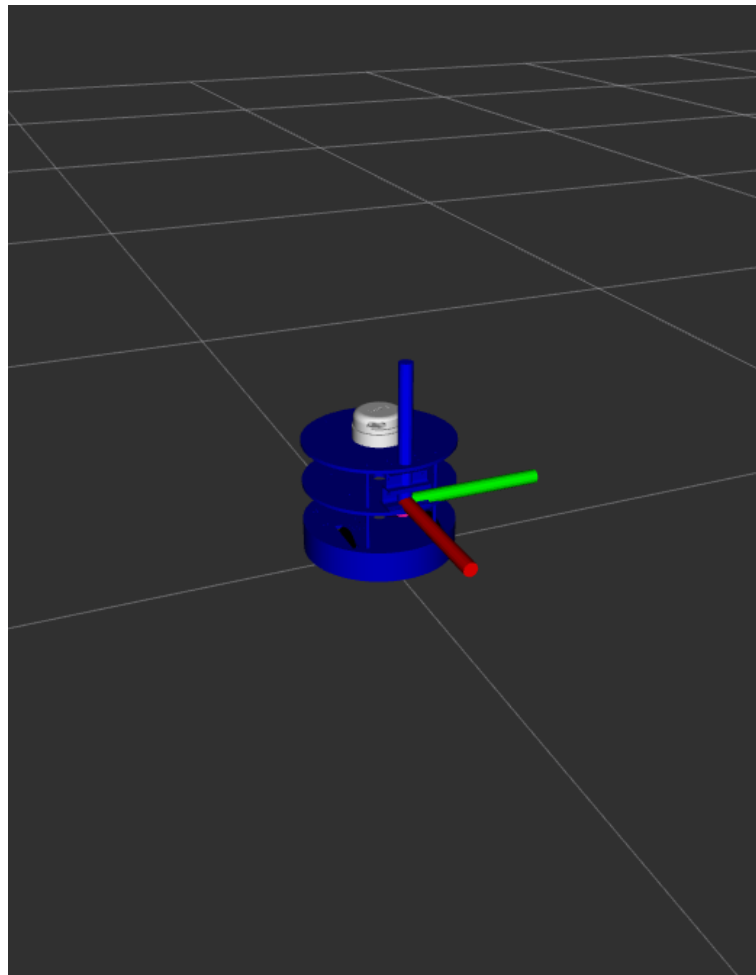


Figure 6: Camera Link

Now, all that remains, was to add the **115 ArUco Tag** in the Gazebo World in

the same way saw in **Homework 3**. An **arucotag** directory has been copied by the one used in the previous Homework. Going to <https://chev.me/arucogen/> it is possible to choose the desired tag and download it as a **.svg** file. It is necessary to convert it in a **.png** file thanks to some Online Converter. In **leonardo\_race\_field.sdf** file

```
<include>
  <uri>
    model://arucotag
  </uri>
  <name>arucotag</name>
  <pose>-3.71 -0.69 0.2 1.57 0.0 3.14</pose>
</include>
```

In **model.sdf**

```
<?xml version="1.0" encoding="UTF-8"?>
<sdf version='1.9'>
  <model name='arucotag'>
    <static>true</static>
    <pose>0 0 0.001 0 0 0</pose>
    <link name='base'>
      <visual name='base_visual'>
        <geometry>
          <plane>
            <normal>0 0 1</normal>
            <size>0.1 0.1</size>
          </plane>
        </geometry>
        <material>
          <diffuse>1 1 1 1</diffuse>
          <specular>0.4 0.4 0.4 1</specular>
          <pbr>
            <metal>
              <albedo_map>model://arucotag/aruco-115.png</albedo_map>
            </metal>
          </pbr>
        </material>
      </visual>
    </link>
  </model>
</sdf>
```

and in **model.config**

```
<?xml version="1.0"?>
<model>
  <name>arucotag</name>
  <version>1.0</version>
  <sdf version="1.9">model.sdf</sdf>
```

```
<author>
  <name>Emmanuel Patellaro</name>
  <email>e.patellaro@studenti.unina.it</email>
</author>
<description>Aruco tag model</description>
</model>
```

The pose of the ArUco tag was chosen so that it is positioned on the object and visible to the robot, which is small in size.

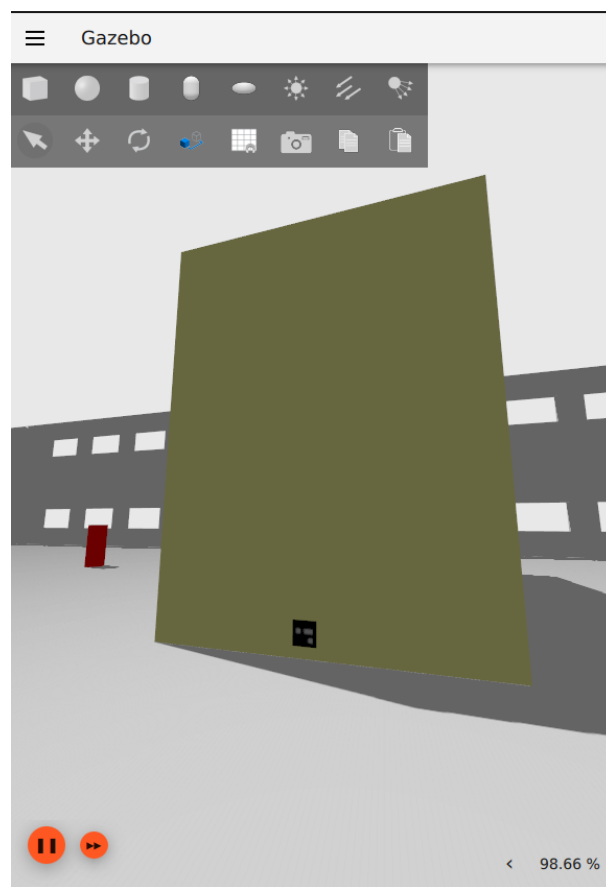


Figure 7: 115-ArUco Tag on Obstacle 9

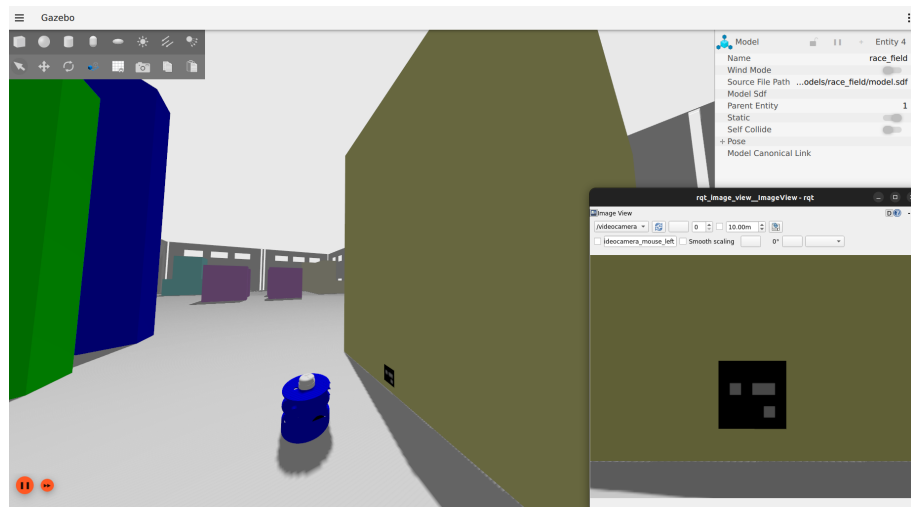


Figure 8: 115-ArUco Tag through the Camera Sensor with fra2mo near to it

# Autonomous Navigation Task

In this chapter, an **Autonomous Navigation Task** has implemented using the **Nav2 Simple Commander API**. This is made thanks to the definition of **4 Position Goals** that fra2mo must reach.

## Goals

First, let's define **4 Goals** in a dedicated **goals.yaml** file. They must have the following poses with respect to the **Map Frame**. As suggested by the professor, since these poses are referenced to the **Map Frame**, the fra2mo robot was respawned at the original position (the **Origin** of the Map Frame) in order to avoid translating all the goals. This is because the **Odometry Frame** has its origin not at the map origin but at the initial position of the robot's first spawn (*AT THE END OF THIS CHAPTER ALSO THE VIDEO STARTING FROM THE PREVIOUS INITIAL POSITION OF CHAPTER 1 WILL BE SHOWN*).

The **Goals** are:

1. **Goal\_1:**     $x = 0m$     $y = 3m$     $Y = 0deg$
2. **Goal\_2:**     $x = 6m$     $y = 4m$     $Y = 30deg$
3. **Goal\_3:**     $x = 6.5m$     $y = -1.4m$     $Y = 180deg$
4. **Goal\_4:**     $x = -1.6m$     $y = -2.5m$     $Y = 75deg$

So, in **goals.yaml**:



```
waypoints:
- position:
  x: 0.0
  y: 3.0
  z: 0.0
  orientation:
  x: 0.0
  y: 0.0
  z: 0.0
  w: 1.0
- position:
  x: 6.0
  y: 4.0
  z: 0.0
  orientation:
  x: 0.0
  y: 0.0
  z: 0.2588190451025207
  w: 0.9659258262890683
- position:
  x: 6.5
  y: -1.4
  z: 0.0
  orientation:
  x: 0.0
  y: 0.0
  z: 1.0
  w: 0.0
- position:
  x: -1.6
  y: -2.5
  z: 0.0
  orientation:
  x: 0.0
  y: 0.0
  z: 0.6087614290087207
  w: 0.7933533402912352
```

To convert **RPY Angles** in **Quaternion** the formula

$$q_x = \sin\left(\frac{R}{2}\right) \cos\left(\frac{P}{2}\right) \cos\left(\frac{Y}{2}\right) - \cos\left(\frac{R}{2}\right) \sin\left(\frac{P}{2}\right) \sin\left(\frac{Y}{2}\right)$$

$$q_y = \cos\left(\frac{R}{2}\right) \sin\left(\frac{P}{2}\right) \cos\left(\frac{Y}{2}\right) + \sin\left(\frac{R}{2}\right) \cos\left(\frac{P}{2}\right) \sin\left(\frac{Y}{2}\right)$$

$$q_z = \cos\left(\frac{R}{2}\right) \cos\left(\frac{P}{2}\right) \sin\left(\frac{Y}{2}\right) - \sin\left(\frac{R}{2}\right) \sin\left(\frac{P}{2}\right) \cos\left(\frac{Y}{2}\right)$$

$$q_w = \cos\left(\frac{R}{2}\right) \cos\left(\frac{P}{2}\right) \cos\left(\frac{Y}{2}\right) + \sin\left(\frac{R}{2}\right) \sin\left(\frac{P}{2}\right) \sin\left(\frac{Y}{2}\right)$$

was used. Then the file **follow\_waypoint.py** was modified in order to send the defined goals to the mobile platform in a given order. The order is:

Goal\_3 → Goal\_4 → Goal\_2 → Goal\_1

So the content of the file is:

```
#!/usr/bin/env python3

from geometry_msgs.msg import PoseStamped
from nav2_simple_commander.robot_navigator import BasicNavigator,
    TaskResult
import rclpy
from rclpy.duration import Duration
import yaml
import os
from ament_index_python.packages import get_package_share_directory

goals_yaml_path=os.path.join(get_package_share_directory('
    rl_fra2mo_description'), "config", "goals.yaml")
with open(goals_yaml_path, 'r') as yaml_file:
    yaml_content = yaml.safe_load(yaml_file)

#waypoints = yaml.safe_load('''
#waypoints:
#  - position:
#    x: 7.0
#    y: 0.0
#    z: 0.0
#    orientation:
#    x: 0.0
#    y: 0.0
#    z: -0.0055409271259092485
#    w: 0.9999846489454652
#  - position:
#    x: -1.8789787292480469
#    y: 0.0
#    z: 0.0
#    orientation:
#    x: 0.0
#    y: 0.0
#    z: 0.010695864295550759
#    w: 0.9999427976074288
#  - position:
#    x: 0.0
#    y: 0.6118782758712769
#    z: 0.0
```

```

#     orientation:
#         x: 0.0
#         y: 0.0
#         z: 0.01899610435153287
#         w: 0.9998195577300264
#''' )

def main():
    rclpy.init()
    navigator = BasicNavigator()

    def create_pose(transform):
        pose = PoseStamped()
        pose.header.frame_id = 'map'
        pose.header.stamp = navigator.get_clock().now().to_msg()
        pose.pose.position.x = transform["position"]["x"]
        pose.pose.position.y = transform["position"]["y"]
        pose.pose.position.z = transform["position"]["z"]
        pose.pose.orientation.x = transform["orientation"]["x"]
        pose.pose.orientation.y = transform["orientation"]["y"]
        pose.pose.orientation.z = transform["orientation"]["z"]
        pose.pose.orientation.w = transform["orientation"]["w"]
        return pose

    goals = list(map(create_pose, yaml_content["waypoints"]))

    # Order: Goal 3 -> Goal 4 -> Goal 2 -> Goal 1
    reordered_goal_indices = [2, 3, 1, 0] # Python indexing starts
    at 0
    goal_poses = [goals[i] for i in reordered_goal_indices]

    # Wait for navigation to fully activate, since autostarting nav2
    navigator.waitUntilNav2Active(localizer="smoother_server")

    # sanity check a valid path exists
    # path = navigator.getPath(initial_pose, goal_pose)

    nav_start = navigator.get_clock().now()
    navigator.followWaypoints(goal_poses)

    i = 0
    while not navigator.isTaskComplete():
        #####
        #
        # Implement some code here for your application!
        #
        #####

        # Do something with the feedback
        i = i + 1
        feedback = navigator.getFeedback()

        if feedback and i % 5 == 0:
            print('Executing_current_waypoint:_' +
                  str(feedback.current_waypoint + 1) + '/' + str(len(
                      goal_poses)))

```

```
now = navigator.get_clock().now()

# Some navigation timeout to demo cancellation
if now - nav_start > Duration(seconds=600):
    navigator.cancelTask()

# Do something depending on the return code
result = navigator.getResult()
if result == TaskResult.SUCCEEDED:
    print('Goal_succeeded!')
elif result == TaskResult.CANCELED:
    print('Goal_was_canceled!')
elif result == TaskResult.FAILED:
    print('Goal_failed!')
else:
    print('Goal_has_an_invalid_return_status!')

# navigator.lifecycleShutdown()

exit(0)

if __name__ == '__main__':
    main()
```

## Implementation

All that remains is to start the simulation. The following will show screenshots of the **Goal Poses** reached by the robot during the simulation. Additionally, a video in 2x will be included (recorded using a mobile phone due to the low graphical performance of my PC, which does not allow screen recording and code execution simultaneously). The instructions to send are (with this order and in different terminals):

```
ros2 launch rl_fra2mo_description gazebo_fra2mo.launch.py
ros2 launch rl_fra2mo_description fra2mo_explore.launch.py
ros2 launch rl_fra2mo_description fra2mo_slam.launch.py
ros2 launch rl_fra2mo_description display_fra2mo.launch.py
ros2 run rl_fra2mo_description follow_waypoints.py
```

The fourth instruction open **Rviz2** automatically with the correct configuration (in this case **explore.rviz**, but also **slam\_view.rviz** could be good).

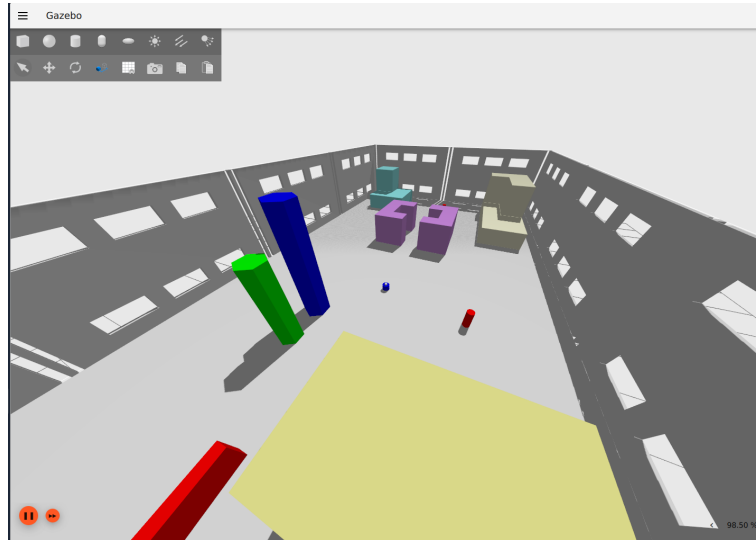


Figure 9: Starting Position

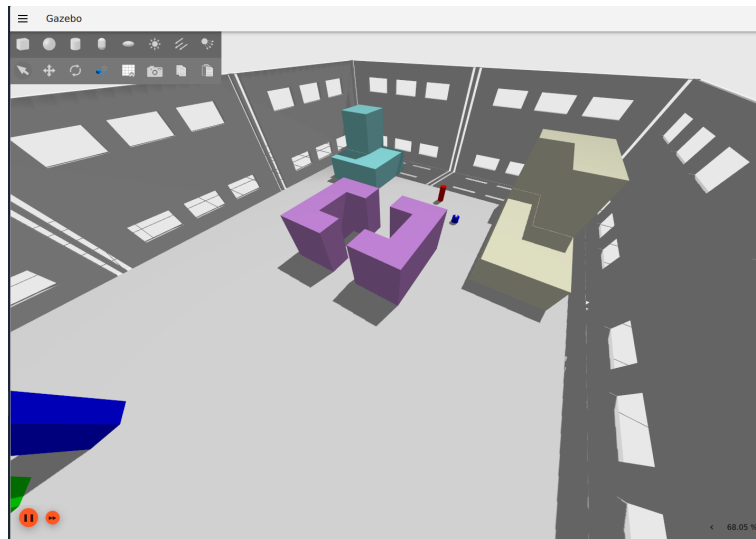


Figure 10: Goal\_3

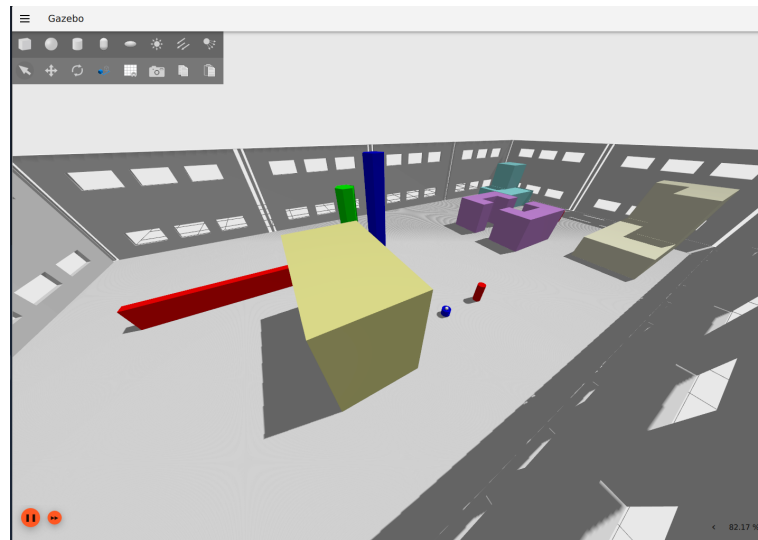


Figure 11: Goal\_4

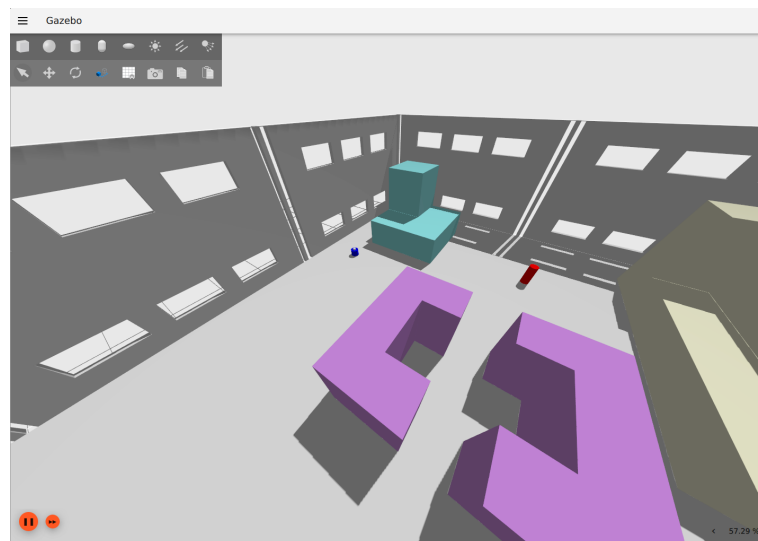


Figure 12: Goal\_2

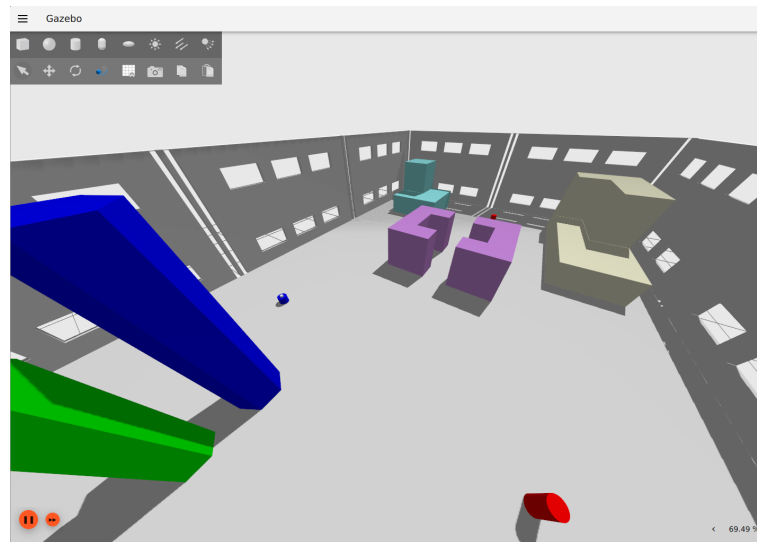


Figure 13: Goal\_1

Here the link to the video:

<https://youtu.be/e124JUfNHG8>

As mentioned earlier, a test was also conducted from the modified Initial Position of the previous chapter. It is necessary to change the Initial Position in **gazebo\_fra2mo.launch.py** as done before (by commenting out the current one and activating the commented one)

```
position = [-3.0, 3.5, 0.100, -1.57]
#position = [0.0, 0.0, 0.100, 0.0]
```

and then launch the file **follow\_waypoints\_no\_origin**, which contains the following modifications.:

```
def create_pose(transform):
    translation = {"x": -3, "y": 3.3}
    rotation = -math.pi / 2

    map_position = transform["position"]
    map_orientation = transform["orientation"]

    x_dash = map_position["x"] - translation["x"]
    y_dash = map_position["y"] - translation["y"]

    #Inverse Rotation
    new_x = math.cos(-rotation) * x_dash - math.sin(-rotation) *
        y_dash
```

```

new_y = math.sin(-rotation) * x_dash + math.cos(-rotation) *
        y_dash

rot_q = {
    "x": 0,
    "y": 0,
    "z": math.sin(-rotation / 2),
    "w": math.cos(-rotation / 2)
}

new_q = {
    "x": rot_q["w"] * map_orientation["x"] + rot_q["x"] *
        map_orientation["w"] + rot_q["y"] * map_orientation["z"]
        - rot_q["z"] * map_orientation["y"],
    "y": rot_q["w"] * map_orientation["y"] - rot_q["x"] *
        map_orientation["z"] + rot_q["y"] * map_orientation["w"]
        + rot_q["z"] * map_orientation["x"],
    "z": rot_q["w"] * map_orientation["z"] + rot_q["x"] *
        map_orientation["y"] - rot_q["y"] * map_orientation["x"]
        + rot_q["z"] * map_orientation["w"],
    "w": rot_q["w"] * map_orientation["w"] - rot_q["x"] *
        map_orientation["x"] - rot_q["y"] * map_orientation["y"]
        - rot_q["z"] * map_orientation["z"]
}

pose = PoseStamped()
pose.header.frame_id = 'map'
pose.header.stamp = navigator.get_clock().now().to_msg()
pose.pose.position.x = new_x
pose.pose.position.y = new_y
pose.pose.position.z = map_position["z"]
pose.pose.orientation.x = new_q["x"]
pose.pose.orientation.y = new_q["y"]
pose.pose.orientation.z = new_q["z"]
pose.pose.orientation.w = new_q["w"]
return pose

```

Here the video:

<https://youtu.be/UgWw3rSzIkk>

## Bag File and Plot

As required by the Homework instructions, a **bagfile** was created to record the **pose** of the fra2mo robot. Subsequently, the robot's **Trajectory** in the **XY-Plane** was plotted using **MATLAB**. During the simulation the topic **/pose** was registered.



```

user@patpc:~/ros2_ws$ ros2 topic list
/amcl_pose
/behavior_server/transition_event
/behavior_tree_log
/bond
/bt_navigator/transition_event
/camera_info
/clicked_point
/clock
/cmd_vel
/cmd_vel_nav
/cmd_vel_teleop
/controller_server/transition_event
/diagnostics
/global_costmap/costmap
/global_costmap/costmap_raw
/global_costmap/costmap_updates
/global_costmap/footprint
/global_costmap/global_costmap/transition_e
/global_costmap/published_footprint
/goal_pose
/initialpose
/joint_states
/lidar
/local_costmap/costmap
/local_costmap/costmap_raw
/local_costmap/costmap_updates
/local_costmap/footprint
/local_costmap/local_costmap/transition_e
/local_costmap/published_footprint
/lookahead_collision_arc
/lookahead_point
/map
/map_updates
/model/fra2mo/odometry
/odom
/parameter_events
/plan
/plan_smoothed
/planner_server/transition_event
/preempt_teleop
/received_global_plan
/robot_description
/rosout
/scan_filtered
/smooth_server/transition_event
/speed_limit
/tf
/tf_static
/velocity_smoother/transition_event
/videocamera
/waypoints

```

Figure 14: Topic List

Like in Homework 2, the plot was generated in MATLAB using the following code, written in `plot_fra2mo_pose.m` file in `bag` directory:

```

unzip("bag_pose.zip");
folderPath = fullfile(pwd,"bag_pose");
bagReader = ros2bagreader(folderPath);
baginfo = ros2("bag","info",folderPath)
msgs = readMessages(bagReader);
bagReader.AvailableTopics
disp(msgs{1}.pose.pose.position)
numMessages = length(msgs);
x_positions = zeros(numMessages, 1);

y_positions = zeros(numMessages, 1);
for i = 1:numMessages
    x_positions(i) = msgs{i}.pose.pose.position.x;

```

```

        y_positions(i) = msgs{i}.pose.pose.position.y;
    end

    plot(x_positions, y_positions, '-o', 'LineWidth', 1.5, 'MarkerSize',
        5);
    xlabel('x[m]');
    ylabel('y[m]');
    title('fra2mo_pose');
    grid on;

    highlight_x = [6.5, -1.6, 6, 0, 0];
    highlight_y = [-1.4, -2.5, 4, 3, 0];

    hold on
    scatter(highlight_x, highlight_y, 100, 'r', 'filled');

```

The starting point and the various goals have been highlighted in red. It can be observed that the robot correctly follows the specified trajectory and successfully reaches the various goals defined at the beginning. Let's show some useful

## MATLAB Screenshots

```

>> baginfo = ros2("bag","info",folderPath)

baginfo =

    struct with fields:
        Path: '/MATLAB Drive/rosbag2_2024_12_14-23_13_02'
        FileInformation: "rosbag2_2024_12_14-23_13_02_0.db3"
        Version: '5'
        StorageId: 'sqlite3'
        Duration: 162.7720
        Start: [1x1 struct]
        End: [1x1 struct]
        Size: 443404
        Messages: 975
        Types: [1x1 struct]
        Topics: [1x1 struct]
        CompressionFormat: 'none'
        CompressionMode: 'none'

```

Figure 15: BagInfo

```

>> bagReader.AvailableTopics

ans =

    1x3 table
           NumMessages      MessageType      MessageDefinition
           _____      _____      _____
    /pose           975      geometry_msgs/PoseWithCovarianceStamped      {...}

```

Figure 16: Available Topic

```

>> disp(msgs{1})
    MessageType: 'geometry_msgs/PoseWithCovarianceStamped'
      header: [1x1 struct]
       pose: [1x1 struct]

>> disp(msgs{1}.pose)
    MessageType: 'geometry_msgs/PoseWithCovariance'
      pose: [1x1 struct]
 covariance: [36x1 double]

>> disp(msgs{1}.pose.pose)
    MessageType: 'geometry_msgs/Pose'
  position: [1x1 struct]
orientation: [1x1 struct]

>> disp(msgs{1}.pose.pose.position)
    MessageType: 'geometry_msgs/Point'
      x: -4.0059e-04
      y: -9.2157e-05
      z: 0

```

Figure 17: Pose Message

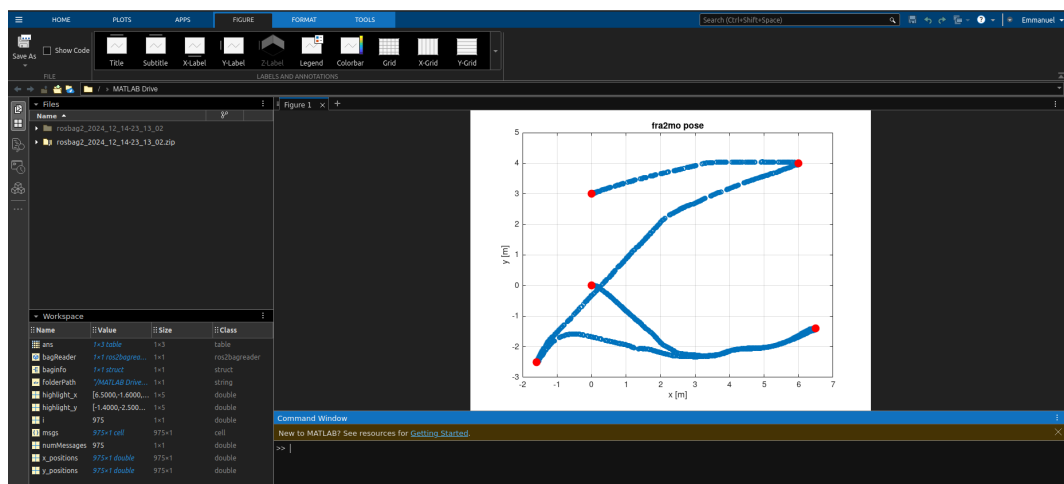


Figure 18: MATLAB Overview

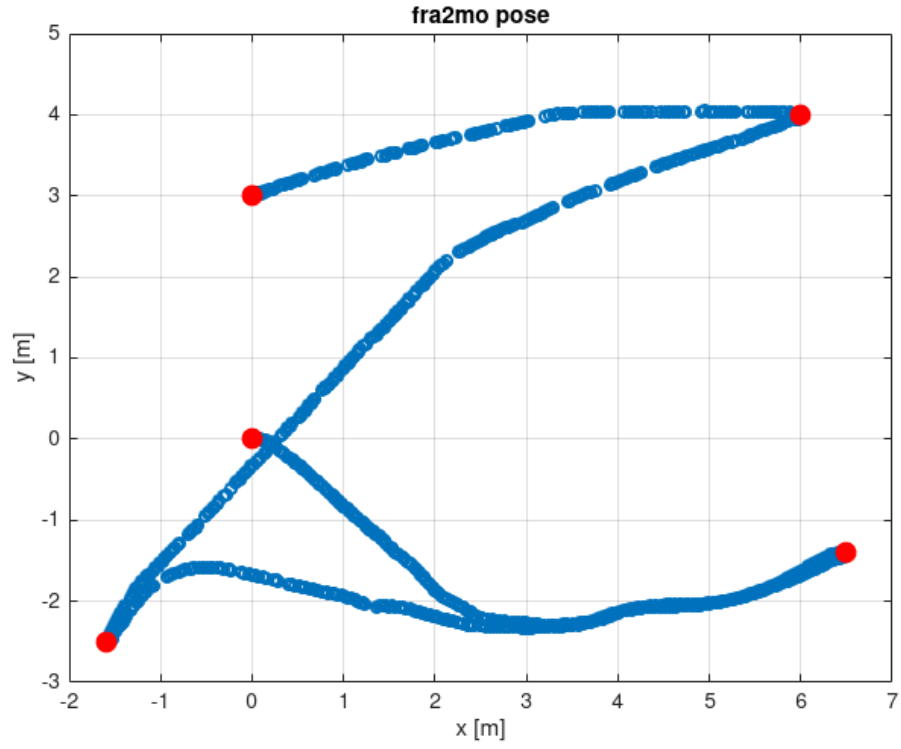


Figure 19: fra2mo Trajectory

As can be seen from the previous figure, the trajectory is exactly the one requested during the design phase. In fact, the robot moves from the initial position  $(0,0)$  to  $(6.5,-1.4)$ (Goal\_3), then to  $(-1.6,-2.5)$ (Goal\_4), then to  $(6,4)$ (Goal\_2), until it reaches the final point  $(0,3)$ (Goal\_1).

# Map of the Environment and Parameters Tuning

In this chapter, a **Complete Map** of the environment has been made using the same approach of the previous chapter changing something. Then, some attempts have been executed in order to highlight all the results due to these changes.

## Complete Map

As mentioned in the introduction of this chapter, various goals were added and modified to allow the robot to **map** the entire **Environment**, including the relevant **Obstacles**. Several trials were conducted, and additional points were added (without giving much consideration to orientation). In the end, 5 of these goals were selected to provide a complete overview of the environment surrounding the robot. All the Goals were written in the file **more\_goals.yaml** in **config** directory:

```
waypoints:
  - position:
      x: 0.0
      y: 3.0
      z: 0.0
    orientation:
      x: 0.0
      y: 0.0
      z: 0.0
      w: 1.0
```

```

- position:
  x: 6.0
  y: 4.0
  z: 0.0
  orientation:
    x: 0.0
    y: 0.0
    z: 0.2588190451025207
    w: 0.9659258262890683
- position:
  x: 6.5
  y: -1.4
  z: 0.0
  orientation:
    x: 0.0
    y: 0.0
    z: 1.0
    w: 0.0
- position:
  x: -1.6
  y: -2.5
  z: 0.0
  orientation:
    x: 0.0
    y: 0.0
    z: 0.6087614290087207
    w: 0.7933533402912352
- position:
  x: 8.0
  y: -4.65
  z: 0.0
  orientation:
    x: 0.0
    y: 0.0
    z: 0.0
    w: 1.0
- position:
  x: 8.0
  y: 4.5
  z: 0.0
  orientation:
    x: 0.0
    y: 0.0
    z: 0.0
    w: 1.0
- position:
  x: 7.0
  y: 0.2
  z: 0.0
  orientation:
    x: 0.0
    y: 0.0
    z: 0.0
    w: 1.0
- position:
  x: -5.0

```

```
    y: 3.5
    z: 0.0
  orientation:
    x: 0.0
    y: 0.0
    z: 0.0
    w: 1.0
- position:
  x: -5.0
  y: -3.5
  z: 0.0
  orientation:
    x: 0.0
    y: 0.0
    z: 0.0
    w: 1.0
```

Only five of these were chosen. They are specified in

**follow\_more\_waypoints.py** file in **scripts** directory.

```
#!/usr/bin/env python3

from geometry_msgs.msg import PoseStamped
from nav2_simple_commander.robot_navigator import BasicNavigator, \
    TaskResult
import rclpy
from rclpy.duration import Duration
import yaml
import os
from ament_index_python.packages import get_package_share_directory

goals_yaml_path=os.path.join(get_package_share_directory('
    rl_fra2mo_description'), "config", "more_goals.yaml")
with open(goals_yaml_path, 'r') as yaml_file:
    yaml_content = yaml.safe_load(yaml_file)

def main():
    rclpy.init()
    navigator = BasicNavigator()

    def create_pose(transform):
        pose = PoseStamped()
        pose.header.frame_id = 'map'
        pose.header.stamp = navigator.get_clock().now().to_msg()
        pose.pose.position.x = transform["position"]["x"]
        pose.pose.position.y = transform["position"]["y"]
        pose.pose.position.z = transform["position"]["z"]
        pose.pose.orientation.x = transform["orientation"]["x"]
        pose.pose.orientation.y = transform["orientation"]["y"]
        pose.pose.orientation.z = transform["orientation"]["z"]
        pose.pose.orientation.w = transform["orientation"]["w"]
        return pose
```

```
goals = list(map(create_pose, yaml_content["waypoints"]))

reordered_goal_indices = [6, 4, 8, 7, 5] # Python indexing
    starts at 0
goal_poses = [goals[i] for i in reordered_goal_indices]

# Wait for navigation to fully activate, since autostarting nav2
navigator.waitForNav2Active(localizer="smoother_server")

# sanity check a valid path exists
# path = navigator.getPath(initial_pose, goal_pose)

nav_start = navigator.get_clock().now()
navigator.followWaypoints(goal_poses)

i = 0
while not navigator.isTaskComplete():
    #####
    #
    # Implement some code here for your application!
    #
    #####

    # Do something with the feedback
    i = i + 1
    feedback = navigator.getFeedback()

    if feedback and i % 5 == 0:
        print('Executing_current_waypoint:_' +
              str(feedback.current_waypoint + 1) + '/' + str(len(
                  goal_poses)))
        now = navigator.get_clock().now()

        # Some navigation timeout to demo cancellation
        if now - nav_start > Duration(seconds=600):
            navigator.cancelTask()

    # Do something depending on the return code
    result = navigator.getResult()
    if result == TaskResult.SUCCEEDED:
        print('Goal_succeeded!')
    elif result == TaskResult.CANCELED:
        print('Goal_was_canceled!')
    elif result == TaskResult.FAILED:
        print('Goal_failed!')
    else:
        print('Goal_has_an_invalid_return_status!')

# navigator.lifecycleShutdown()

exit(0)

if __name__ == '__main__':
    main()
```



The procedure is the same as the one shown in the previous chapter. The resulting image of the **Complete Map** will be presented both by saving it with the appropriate command and through a screenshot taken at the end of the simulation. The instructions to be executed are as follows:

```
ros2 launch rl_fra2mo_description gazebo_fra2mo.launch.py
ros2 launch rl_fra2mo_description fra2mo_explore.launch.py
ros2 launch rl_fra2mo_description fra2mo_slam.launch.py
ros2 launch rl_fra2mo_description display_fra2mo.launch.py
ros2 run rl_fra2mo_description follow_more_waypoints.py
ros2 run nav2_map_server map_saver_cli -f map
```

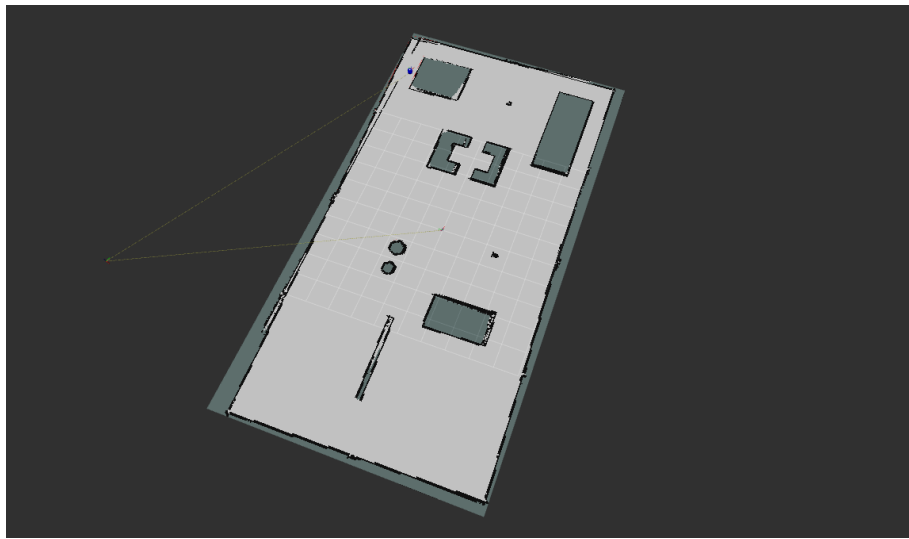


Figure 20: Map with slam\_view.rviz

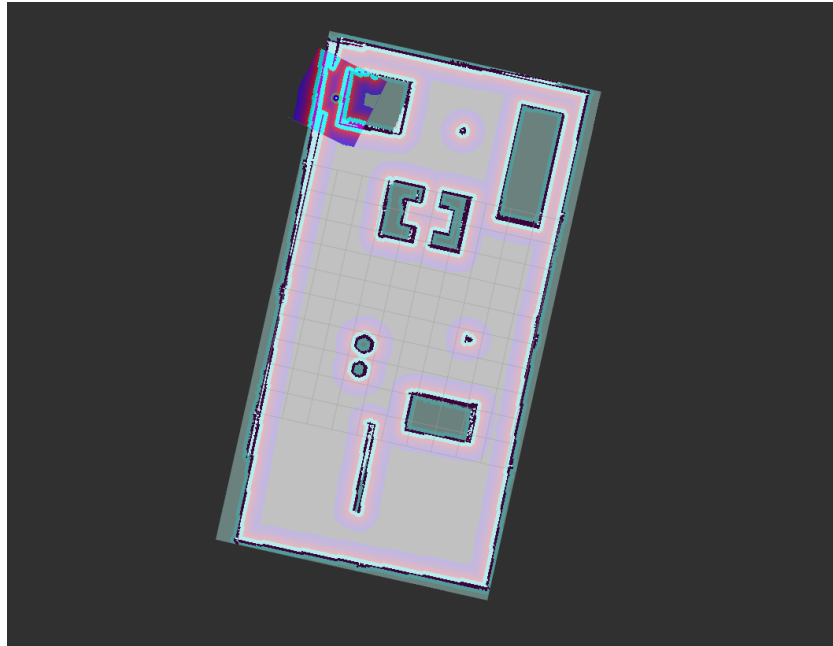


Figure 21: Map with explore.rviz

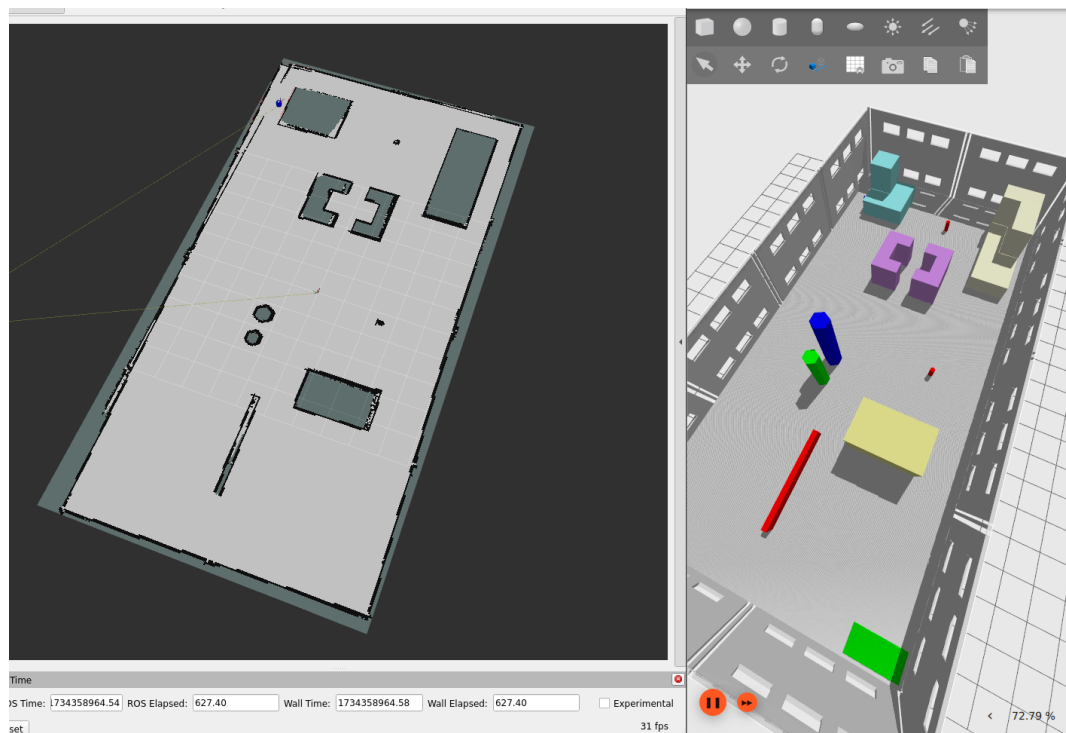


Figure 22: Complete Map and Gazebo Map

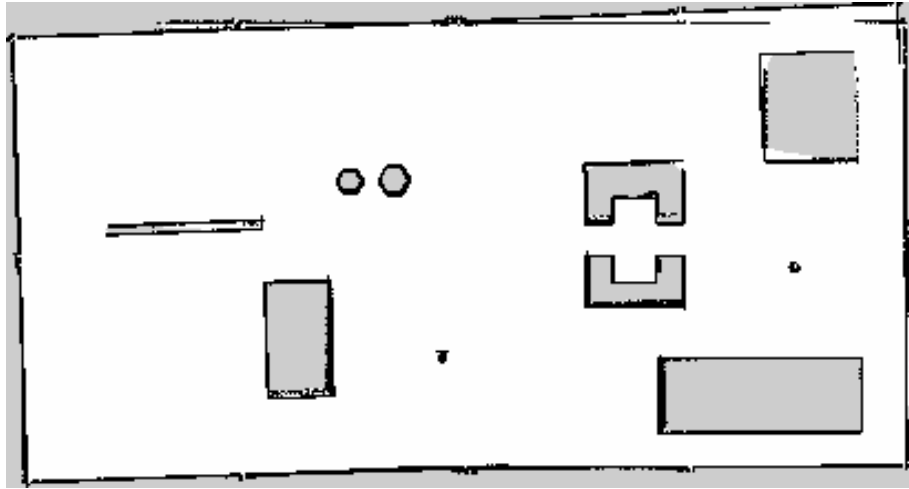


Figure 23: Map by instruction

Here the links to some videos for the mapping (also here in 2x):

- Mapping with **slam\_view.rviz** and **Gazebo**

<https://youtu.be/LTNhsPC4YIk>

- Mapping with **explore.rviz**

<https://youtu.be/AjlYsOHzpd4>

## Parameters Tuning

Various tests were conducted with different **Parameters Configurations** to evaluate the behavior of fra2mo with each change. Only the most significant and notable results will be discussed. In particular, the parameters of interest that were modified were:

- In **slam.yaml** file:
  - **minimum\_travel\_distance**: Minimum distance of travel before processing a new scan
  - **minimum\_travel\_heading**: Minimum changing in heading to justify an update

- **resolution**: Resolution of the 2D occupancy map to generate
- **transform\_publish\_period**: The map to odom transform publish period. 0 will not publish transforms
- In **explore.yaml** file:
  - **inflation\_radius**: The radius in meters to which the map inflates obstacle cost values
  - **cost\_scaling\_factor**: A scaling factor to apply to cost values during inflation. The cost function is computed as follows for all cells in the costmap further than the inscribed radius distance and closer than the inflation radius distance away from an actual obstacle. The function is:
$$\exp(-1.0 * \text{cost\_scaling\_factor} * (\text{distance\_from\_obstacle} - \text{inscribed\_radius})) * (\text{costmap\_2d}::\text{INSCRIBED\_INFLATED\_OBSTACLE} - 1)$$

The modifications to the two files were made individually for each of them. The changes to the parameters in the first file were tested with the default parameters of the second file, and vice versa.

First, the parameters of **slam.yaml** file were changed. The default values are:

```
...
...
transform_publish_period: 0.02
resolution: 0.05
...
...
minimum_travel_distance: 0.01
minimum_travel_heading: 0.01
```

Let's modify in this way, increasing all the 4 parameters:

```
...
...
transform_publish_period: 0.1
resolution: 0.1
...
...
```

```
minimum_travel_distance: 0.05  
minimum_travel_heading: 0.05
```

With this choice, the robot tends to stop much less and moves with fewer jerks.

A significant improvement can be observed in the "tunnel" section located at the top right of the map (between the wall and the obstacle). Previously, the robot tended to stop there for a few seconds. Doing this, the frequency with which the system updates the position was decreased and the resolution of the 2D occupancy map to generate was increased.

By further increasing these values, the robot tends to spin in place and does not move forward. In particular, the most critical parameter is

**transform\_publish\_period.**

```
...  
...  
transform_publish_period: 0.1  
resolution: 0.5  
...  
...  
minimum_travel_distance: 0.1  
minimum_travel_heading: 0.1
```

A possible reason is that there is a too large minimum value before processing a new scan. Furthermore, high resolution can cause various issues related to Obstacle Detection. In fact, the robot tends to perceive obstacles where there are none and remains stalled, believing it cannot move in that particular direction. Here some figures related to this last concept.

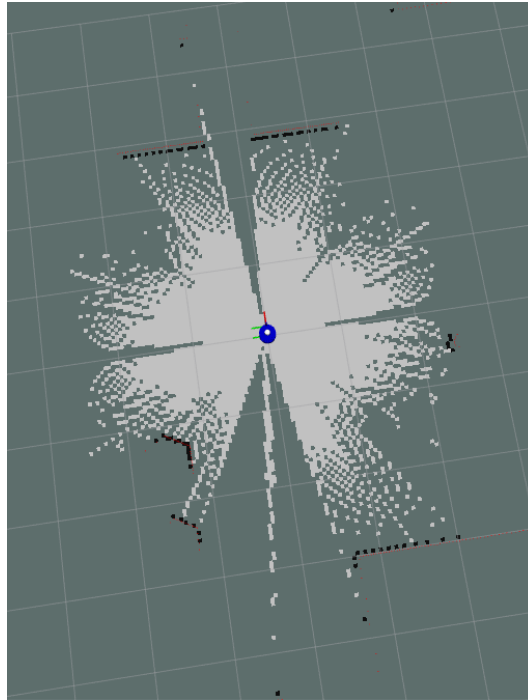


Figure 24: Default Resolution at the start

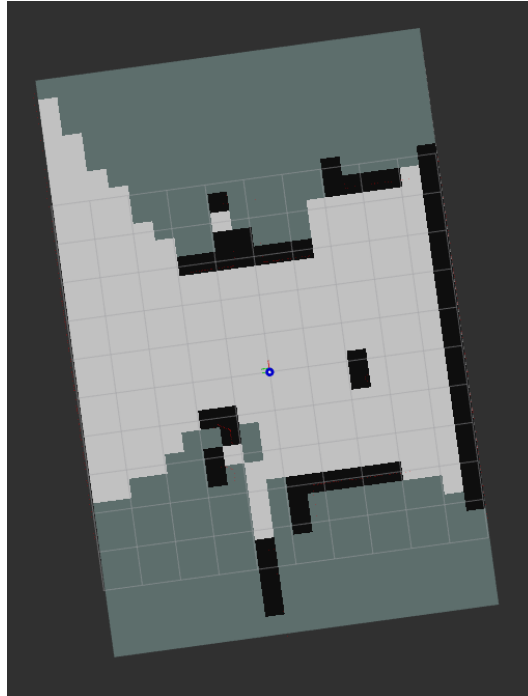


Figure 25: High Resolution at the start

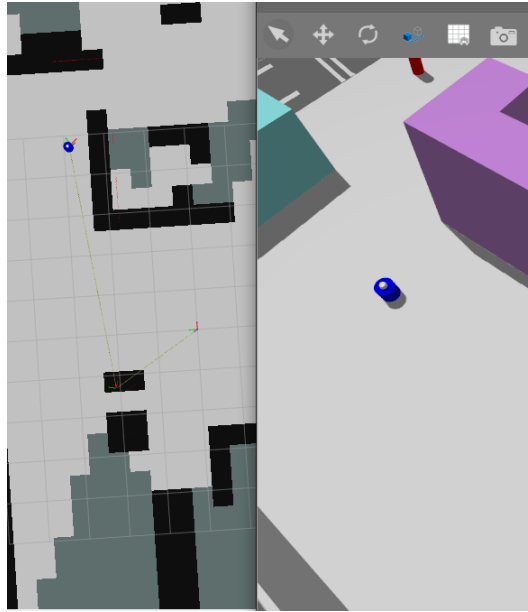


Figure 26: Wrong Obstacle Detection

In addition to this, is possible to observe, even from Figure 25, that the robot will never be able to pass between the obstacles in the center, as it appears to perceive a wall in front of it and considers the path completely blocked.

Now, let's change the parameters of the `explore.yaml` file. The default values are:

```
inflation_radius: 0.75
cost_scaling_factor: 3.0
```

both for **Global** and **Local Costmap**. Let's modify in this way, decreasing all the 4 parameters:

```
inflation_radius: 0.5
cost_scaling_factor: 2.0
```

The **`inflation_radius`** represents the safety area around obstacles, where costs gradually increase as the robot get closer to the obstacle. This ensures that the robot plans paths that avoid not just the obstacles themselves, but also a safety buffer around them to prevent collisions. Instead, for the **`cost_scaling_factor`** the meaning is that the cells containing direct obstacles have very high costs (or infinite), while cells near an obstacle have intermediate costs determined by the

inflation function written before. So this parameter determines how quickly costs decrease as the distance from the obstacle increases. In fact, with this last choice of parameters, for the reasons mentioned above, the robot tends to pass closer to obstacles, and the time it takes to perceive an obstacle improves. Consequently, the robot's movement is noticeably faster and more agile. It is now possible to perform a counter-check using values higher than the default ones:

```
inflation_radius: 1.0  
cost_scaling_factor: 4.0
```

In fact, with this choice, the robot passes much further away from obstacles (especially in reference to obstacles such as the red one at the top of the map, where the robot has to navigate around it). This results in longer trajectories, which are often different from those generated when using default parameters. Therefore, not only does the trajectory length change, but in some cases, the trajectory itself may also change. For excessively high values, the robot tends to stop in front of obstacles before resuming movement. With even higher values, the robot fails to overcome the obstacle and stop in front of it, likely because the cost function becomes too high.



# Vision-Based Navigation

In this chapter, a **Vision-Based Navigation** of the Mobile Platform has been implement. At the end, the **ArUco Pose as TF** has been published following this example

`https://docs.ros.org/en/humble/Tutorials/Intermediate/Tf2/Writing-A-Tf2-Broadcaster-Py.html`

As in **Chapter 2**, an implementation for the robot starting from the Origin and another one for the position introduced in **Chapter 1** have been implemented.

## Navigation

First, a launch file running both the **Navigation** and the **aruco\_ros node** using the robot **Camera** previously added to the Robot Model has been created. During the simulation, a problem was encountered with the previous **ArUco Tag**. Specifically, the ArUco node does not recognize the previously placed tag as **115**. The real ArUco Tag is the following:

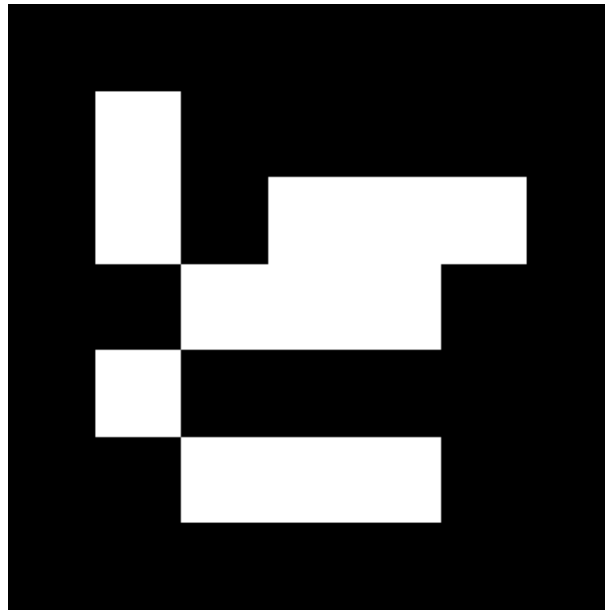


Figure 27: Real 115 ArUco Tag

The launch file is **fra2mo\_aruco.launch.py** and it contains:

```
from launch import LaunchDescription
from launch.actions import DeclareLaunchArgument,
    IncludeLaunchDescription
from launch.launch_description_sources import
    PythonLaunchDescriptionSource
from launch.substitutions import LaunchConfiguration,
    PathJoinSubstitution
from launch_ros.substitutions import FindPackageShare

def generate_launch_description():
    fra2mo_dir = FindPackageShare('rl_fra2mo_description')
    aruco_dir = FindPackageShare('aruco_ros')

    explore_launch = IncludeLaunchDescription(
        PythonLaunchDescriptionSource(
            PathJoinSubstitution([fra2mo_dir, 'launch', '
                fra2mo_explore.launch.py'])))

    aruco_launch = IncludeLaunchDescription(
        PythonLaunchDescriptionSource(
            PathJoinSubstitution([aruco_dir, 'launch', 'single.launch
                .py'])))

    return LaunchDescription(
        [
            explore_launch,
```

```
        aruco_launch,  
    ]  
)
```

The **2D Navigation Task** to implement is the following:

1. Send the Robot in the proximity of **Obstacle 9** (where the ArUco Tag was placed)
2. Make the Robot look for the ArUco Marker. Once detected, it is necessary to retrieve its pose with respect to the **Map Frame**
3. Return the Robot to the **Initial Position**

So an **aruco\_goals.yaml** file in **config** directory has been created in order to define the position goals useful to do the required task:

```
waypoints:  
- position:  
  x: -3.7  
  y: 0.0  
  z: 0.0  
  orientation:  
    x: 0.0  
    y: 0.0  
    z: -0.7068  
    w: 0.7074  
- position:  
  x: 0.0  
  y: 0.0  
  z: 0.0  
  orientation:  
    x: 0.0  
    y: 0.0  
    z: -0.7068  
    w: 0.7074  
- position:  
  x: -3.0  
  y: 3.5  
  z: 0.0  
  orientation:  
    x: 0.0  
    y: 0.0  
    z: 0.0  
    w: 0.1
```

The first one allows the robot to approach Obstacle 9 and to orient itself toward the ArUco Tag. Instead, the second and the third one allow the robot to come

back to their respective Initial Positions. At this point, the two files

**follow\_\_aruco\_\_waypoints.py** and

**follow\_\_aruco\_\_waypoints\_\_no\_\_origin.py** have been created in order to implement the task written before. Now the code for both files.

**follow\_\_aruco\_\_waypoints.py:**

```
#!/usr/bin/env python3

from geometry_msgs.msg import PoseStamped, TransformStamped
from tf2_geometry_msgs import tf2_geometry_msgs
from nav2_simple_commander.robot_navigator import BasicNavigator,
    TaskResult
import rclpy
from rclpy.node import Node
import yaml, os, math
from ament_index_python.packages import get_package_share_directory
from tf2_ros import Buffer, TransformListener,
    StaticTransformBroadcaster, TransformBroadcaster
from tf2_ros import TransformException
from rclpy.duration import Duration
from rclpy.time import Time
import time

goals_yaml_path=os.path.join(get_package_share_directory('
    rl_fra2mo_description'), "config", "aruco_goals.yaml")
with open(goals_yaml_path, 'r') as yaml_file:
    yaml_content = yaml.safe_load(yaml_file)

class ArucoNode(Node):
    def __init__(self):
        super().__init__('aruco_node')

        self.navigator = BasicNavigator()
        self.tf_buffer = Buffer()

        self.aruco_sub = self.create_subscription(
            PoseStamped,
            '/aruco_single/pose',
            self.aruco_callback,
            10
        )

        self.aruco_pose = None

        self.offset_x = 0.0
        self.offset_y = 0.0

        #self.offset_x = -3.0
        #self.offset_y = 3.5

    def aruco_navigation(self):
```

---

```

        # Wait for navigation to fully activate, since autostarting
        # nav2
        self.navigator.waitUntilNav2Active(localizer="smoother_server")

        # Near Obstacle 9
        self.go_near_obstacle()

        # ArUco Detection
        self.aruco_detection()

        # Back
        self.return_to_initial_position()

def go_near_obstacle(self):

    only_one_print = False
    goals = list(map(self.create_pose, yaml_content["waypoints"]))

    # Near Obstacle 9
    reordered_goal_indices = [0]
    goal_poses = [goals[i] for i in reordered_goal_indices]

    self.navigator.followWaypoints(goal_poses)

    # Wait Until Navigation is Completed
    while not self.navigator.isTaskComplete():
        feedback = self.navigator.getFeedback()
        if feedback and not only_one_print:
            print(f'Go_Near_Obstacle_9')
            only_one_print = True

def aruco_detection(self):

    # Wait until Detection
    while not self.aruco_pose:
        rclpy.spin_once(self)

    self.get_logger().info(f"ArUco_Tag_Position_in_Map_Frame:_"
                          f"Position->[x:{self.aruco_pose.pose.position.x},_"
                          f"y:{self.aruco_pose.pose.position.y},_"
                          f"z:{self.aruco_pose.pose.position.z}],_"
                          f"Orientation->[x:{self.aruco_pose.pose.orientation.x},_"
                          f"y:{self.aruco_pose.pose.orientation.y},_"
                          f"z:{self.aruco_pose.pose.orientation.z},_"
                          f"w:{self.aruco_pose.pose.orientation.w}]]")

```

```
def aruco_callback(self, msg):

    try:
        # Camera Frame
        source_frame = msg.header.frame_id

        # Message Time
        timestamp = msg.header.stamp

        # Transformation from Camera Frame and Map Frame
        transform = self.tf_buffer.lookup_transform(
            "map", source_frame, timestamp, timeout=Duration(
                seconds=0.0))

        # ArUco Pose with respect Camera Frame
        aruco_position = msg.pose.position
        aruco_orientation = msg.pose.orientation

        # URDF Offsets
        base_offset = [0.0825, 0.0, 0.04]
        camera_offset = [0.1, 0.0, 0.072]

        # ArUco Pose with respect Map Frame
        aruco_map_pose = PoseStamped()
        aruco_map_pose.header.frame_id = "map"
        aruco_map_pose.header.stamp = self.get_clock().now().
            to_msg()
        aruco_map_pose.pose.position.x = transform.transform.
            translation.x - aruco_position.x - camera_offset[0] +
            base_offset[0]
        aruco_map_pose.pose.position.y = transform.transform.
            translation.y - aruco_position.z + camera_offset[1] +
            base_offset[1]
        aruco_map_pose.pose.position.z = transform.transform.
            translation.z - aruco_position.y + camera_offset[2] +
            base_offset[2]
        aruco_map_pose.pose.orientation = aruco_orientation

        # ArUco Pose for function aruco_detection
        self.aruco_pose = aruco_map_pose

    except TransformException as e:
        self.get_logger().error(f"Error:_{e}")

def return_to_initial_position(self):

    only_one_print = False
    goals = list(map(self.create_pose, yaml_content["waypoints"]))
    )

    # Initial Position
    reordered_goal_indices = [1]
```

---

```

goal_poses = [goals[i] for i in reordered_goal_indices]

self.navigators.followWaypoints(goal_poses)

# Wait Until Navigation is Completed
while not self.navigators.isTaskComplete():
    feedback = self.navigators.getFeedback()
    if feedback and not only_one_print:
        print(f'Back_To_Initial_Position')
        only_one_print = True

def create_pose(self, transform):

    pose = PoseStamped()
    pose.header.frame_id = 'map'
    pose.header.stamp = self.navigators.get_clock().now().to_msg()
    pose.pose.position.x = transform["position"]["x"]
    pose.pose.position.y = transform["position"]["y"]
    pose.pose.position.z = transform["position"]["z"]
    pose.pose.orientation.x = transform["orientation"]["x"]
    pose.pose.orientation.y = transform["orientation"]["y"]
    pose.pose.orientation.z = transform["orientation"]["z"]
    pose.pose.orientation.w = transform["orientation"]["w"]
    return pose

def main():
    rclpy.init()
    node = ArucoNode()
    node.aruco_navigation()
    rclpy.shutdown()

if __name__ == '__main__':
    main()

```

### follow\_aruco\_waypoints\_no\_origin.py:

```

#!/usr/bin/env python3

from geometry_msgs.msg import PoseStamped, TransformStamped
from tf2_geometry_msgs import tf2_geometry_msgs
from nav2_simple_commander.robot_navigator import BasicNavigator,
    TaskResult
import rclpy
from rclpy.node import Node
import yaml, os, math
from ament_index_python.packages import get_package_share_directory
from tf2_ros import Buffer, TransformListener,
    StaticTransformBroadcaster, TransformBroadcaster
from tf2_ros import TransformException
from rclpy.duration import Duration
from rclpy.time import Time
import time

```

```
goals_yaml_path=os.path.join(get_package_share_directory('
    rl_fra2mo_description'), "config", "aruco_goals.yaml")
with open(goals_yaml_path, 'r') as yaml_file:
    yaml_content = yaml.safe_load(yaml_file)

class ArucoNode(Node):
    def __init__(self):
        super().__init__('aruco_node')

        self.navigators = BasicNavigator()
        self.tf_buffer = Buffer()

        self.aruco_sub = self.create_subscription(
            PoseStamped,
            '/aruco_single/pose',
            self.aruco_callback,
            10
        )

        self.aruco_pose = None

        #self.offset_x = 0.0
        #self.offset_y = 0.0

        self.offset_x = -3.0
        self.offset_y = 3.5

    def aruco_navigation(self):

        # Wait for navigation to fully activate, since autostarting
        # nav2
        self.navigators.waitUntilNav2Active(localizer="smoother_server")

        # Near Obstacle 9
        self.go_near_obstacle()

        # ArUco Detection
        self.aruco_detection()

        # Back
        self.return_to_initial_position()

    def go_near_obstacle(self):

        only_one_print = False
        goals = list(map(self.create_pose, yaml_content["waypoints"]))

        # Near Obstacle 9
        reordered_goal_indices = [0]
        goal_poses = [goals[i] for i in reordered_goal_indices]
```



```
self.navigators.followWaypoints(goal_poses)

# Wait Until Navigation is Completed
while not self.navigators.isTaskComplete():
    feedback = self.navigators.getFeedback()
    if feedback and not only_one_print:
        print(f'Go_Near_Obstacle_9')
        only_one_print = True

def aruco_detection(self):

    # Wait until Detection
    while not self.aruco_pose:
        rclpy.spin_once(self)

    self.get_logger().info(f"ArUco_Tag_Position_in_Map_Frame:_
        f"Position->[_x:_{self.aruco_pose.
            pose.position.x},_"
        f"y:_{self.aruco_pose.pose.position.y
            },_z:_{self.aruco_pose.pose.
            position.z}],_"
        f"Orientation->[_x:_{self.aruco_pose.
            pose.orientation.x},_"
        f"y:_{self.aruco_pose.pose.orientation
            .y},_"
        f"z:_{self.aruco_pose.pose.orientation
            .z},_"
        f"w:_{self.aruco_pose.pose.orientation
            .w}]]")

def aruco_callback(self, msg):

    try:
        # Camera Frame
        source_frame = msg.header.frame_id

        # Message Time
        timestamp = msg.header.stamp

        # Transformation from Camera Frame and Map Frame
        transform = self.tf_buffer.lookup_transform(
            "map", source_frame, timestamp, timeout=Duration(
                seconds=0.0))

        # ArUco Pose with respect Camera Frame
        aruco_position = msg.pose.position
        aruco_orientation = msg.pose.orientation

        # URDF Offsets
        base_offset = [0.0825, 0.0, 0.04]
        camera_offset = [0.1, 0.0, 0.072]

        # ArUco Pose with respect Map Frame
```

---

```

        aruco_map_pose = PoseStamped()
        aruco_map_pose.header.frame_id = "map"
        aruco_map_pose.header.stamp = self.get_clock().now().
            to_msg()
        aruco_map_pose.pose.position.x = transform.transform.
            translation.y - aruco_position.x - camera_offset[0] +
            base_offset[0] + self.offset_x
        aruco_map_pose.pose.position.y = transform.transform.
            translation.x - aruco_position.z + camera_offset[1] +
            base_offset[1] - self.offset_y
        aruco_map_pose.pose.position.z = transform.transform.
            translation.z - aruco_position.y + camera_offset[2] +
            base_offset[2]
        aruco_map_pose.pose.orientation = aruco_orientation

        # ArUco Pose for function aruco_detection
        self.aruco_pose = aruco_map_pose

    except TransformException as e:
        self.get_logger().error(f"Error:_{e}")

def return_to_initial_position(self):

    only_one_print = False
    goals = list(map(self.create_pose, yaml_content["waypoints"]))
    )

    # Initial Position
    reordered_goal_indices = [2]
    goal_poses = [goals[i] for i in reordered_goal_indices]

    self.navigators.followWaypoints(goal_poses)

    # Wait Until Navigation is Completed
    while not self.navigators.isTaskComplete():
        feedback = self.navigators.getFeedback()
        if feedback and not only_one_print:
            print(f'Back_To_Initial_Position')
            only_one_print = True

def create_pose(self, transform):

    translation = {"x": self.offset_x, "y": self.offset_y}
    rotation = -math.pi / 2

    map_position = transform["position"]
    map_orientation = transform["orientation"]

    x_dash = map_position["x"] - translation["x"]
    y_dash = map_position["y"] - translation["y"]

    #Inverse Rotation
    new_x = math.cos(-rotation) * x_dash - math.sin(-rotation) *

```

---

```

        y_dash
    new_y = math.sin(-rotation) * x_dash + math.cos(-rotation) *
        y_dash

    rot_q = {
        "x": 0,
        "y": 0,
        "z": math.sin(-rotation / 2),
        "w": math.cos(-rotation / 2)
    }

    new_q = {
        "x": rot_q["w"] * map_orientation["x"] + rot_q["x"] *
            map_orientation["w"] + rot_q["y"] * map_orientation["z"]
            - rot_q["z"] * map_orientation["y"],
        "y": rot_q["w"] * map_orientation["y"] - rot_q["x"] *
            map_orientation["z"] + rot_q["y"] * map_orientation["w"]
            + rot_q["z"] * map_orientation["x"],
        "z": rot_q["w"] * map_orientation["z"] + rot_q["x"] *
            map_orientation["y"] - rot_q["y"] * map_orientation["x"]
            + rot_q["z"] * map_orientation["w"],
        "w": rot_q["w"] * map_orientation["w"] - rot_q["x"] *
            map_orientation["x"] - rot_q["y"] * map_orientation["y"]
            - rot_q["z"] * map_orientation["z"]
    }

    pose = PoseStamped()
    pose.header.frame_id = 'map'
    pose.header.stamp = self.navigators.get_clock().now().to_msg()
    pose.pose.position.x = new_x
    pose.pose.position.y = new_y
    pose.pose.position.z = map_position["z"]
    pose.pose.orientation.x = new_q["x"]
    pose.pose.orientation.y = new_q["y"]
    pose.pose.orientation.z = new_q["z"]
    pose.pose.orientation.w = new_q["w"]
    return pose

def main():
    rclpy.init()
    node = ArucoNode()
    node.aruco_navigation()
    rclpy.shutdown()

if __name__ == '__main__':
    main()

```

**BE CAREFUL:** when the ArUco Pose with respect the Map Frame is computed, there are some differences in the two files because of the Axis Orientation of the **Map Frame** in the two cases. In fact, as can be seen from the videos shown later, the Red Axis and the Green Axis of the Map Frame are

inverted. Furthermore, in the second case (the one with the Initial Position different from the Origin) the Map Frame does not coincide with that of Gazebo. Therefore, as can be seen from the code written above, two **Offset** (**x** and **y**) were added according to the robot's Initial Position. In both cases, however, two additional **Offsets** were added, referring to the position of the **camera\_link** and the position of the **base\_link** relative to the ground. Below is a figure that clearly illustrates this concept.

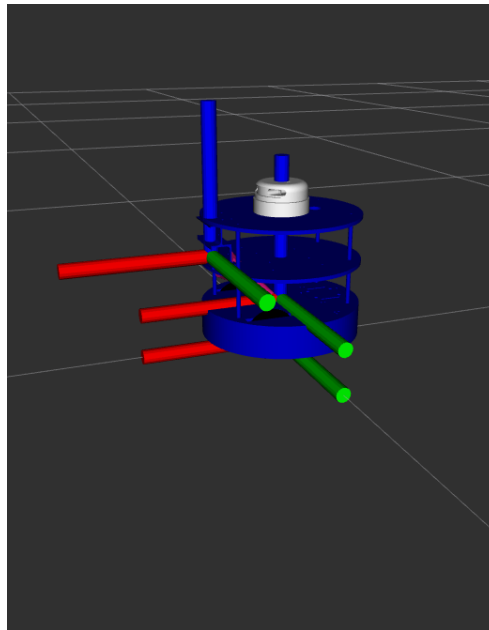


Figure 28: Frames for Offsets

The instructions to run are:

1. For Initial Position in the Origin:

```
ros2 launch rl_fra2mo_description gazebo_fra2mo.launch.py
ros2 launch rl_fra2mo_description fra2mo_aruco.launch.py
ros2 run rl_fra2mo_description follow_aruco_waypoints.py
```

to start the simulation. Additionally, it is possible to give the instruction

```
ros2 run rqt_image_view rqt_image_view
```

to visualize the detection of the ArUco Tag by selecting  
`/aruco_single/result`.

2. For Initial Position in the one introduced in **Chapter 1**: the same instructions of the previous case but, instead of the third one:

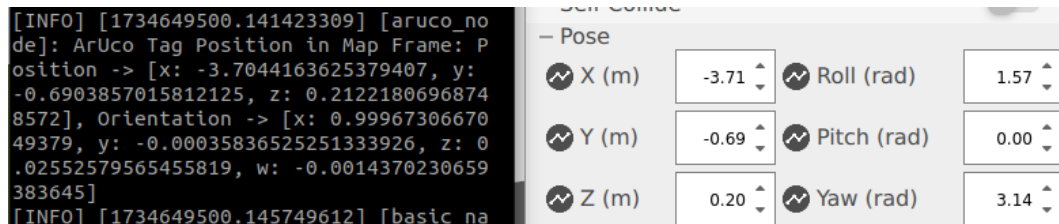
```
ros2 run rl_fra2mo_description
follow_aruco_waypoints_no_origin.py
```

The videos for these two implementation will be shown in the next section with the **TF Publishing** of the same ArUco Pose.

Now some Screenshots showing the results.

```
[INFO] [1734649500.141423309] [aruco_node]: ArUco Tag Position in Map Frame
: Position -> [x: -3.7044163625379407, y: -0.6903857015812125, z: 0.2122180
6968748572], Orientation -> [x: 0.9996730667049379, y: -0.00035836525251333
926, z: 0.02552579565455819, w: -0.0014370230659383645]
```

Figure 29: ArUco Pose with respect the Map Frame



```
[INFO] [1734649500.141423309] [aruco_node]: ArUco Tag Position in Map Frame: P
osition -> [x: -3.7044163625379407, y:
-0.6903857015812125, z: 0.2122180696874
8572], Orientation -> [x: 0.99967306670
49379, y: -0.00035836525251333926, z: 0
.02552579565455819, w: -0.0014370230659
383645]
[INFO] [1734649500.145749612] [basic_na
```

Pose			
X (m)	-3.71	Roll (rad)	1.57
Y (m)	-0.69	Pitch (rad)	0.00
Z (m)	0.20	Yaw (rad)	3.14

Figure 30: ArUco Pose Comparison with Gazebo

## ArUco Pose and TF

The last point of the Homework requires publishing the **ArUco Pose** as **TF**. To do this a little modification of the previous files has been made in order to fulfill this request. In fact, this part of the code was added to both files:

```
...
...
self.tf_broadcaster = StaticTransformBroadcaster(self)
```

```

...
...
def aruco_callback(self, msg):
...
...
    # ArUco Pose as TF
    t = TransformStamped()

    # Timestamp
    t.header.stamp = self.get_clock().now().to_msg()

    t.header.frame_id = "map"
    t.child_frame_id = "aruco_pose_tf"

    # TF Position
    t.transform.translation.x = aruco_map_pose.pose.position.
        x
    t.transform.translation.y = aruco_map_pose.pose.position.
        y
    t.transform.translation.z = aruco_map_pose.pose.position.
        z

    # TF Orientation
    t.transform.rotation.x = aruco_map_pose.pose.orientation.
        x
    t.transform.rotation.y = aruco_map_pose.pose.orientation.
        y
    t.transform.rotation.z = aruco_map_pose.pose.orientation.
        z
    t.transform.rotation.w = aruco_map_pose.pose.orientation.
        w

    # Send TF
    self.tf_broadcaster.sendTransform(t)

    self.get_logger().info("\nArUco_TF_Published_Successfully
        \n")
...
...

```

In this way, a new **TF** will be published once the function **aruco\_detection()**, in the Node, starts.

The instructions to run are the same as before for both cases. Additionally, it is possible to execute:

```
rviz2
```

adding **TF** and **RobotModel** to see the new **aruco\_pose\_tf** once created.

```
ros2 topic echo tf_static
```

to stamp the results of the new **TF**.

The results are the same as before.

```
transforms:
- header:
  stamp:
    sec: 1734738261
    nanosec: 184995349
  frame_id: map
  child_frame_id: aruco_pose_tf
  transform:
    translation:
      x: -3.71247995568209
      y: -0.7045559090682934
      z: 0.21219233910739418
    rotation:
      x: 0.9993211301048587
      y: 0.0007144701522775747
      z: 0.03644350171670283
      w: 0.0053516017204392275
---
```

Figure 31: aruco\_pose\_tf

Giving the instruction:

```
ros2 run tf2_tools view_frames
```

it is possible to obtain a general overview of the existing TFs.

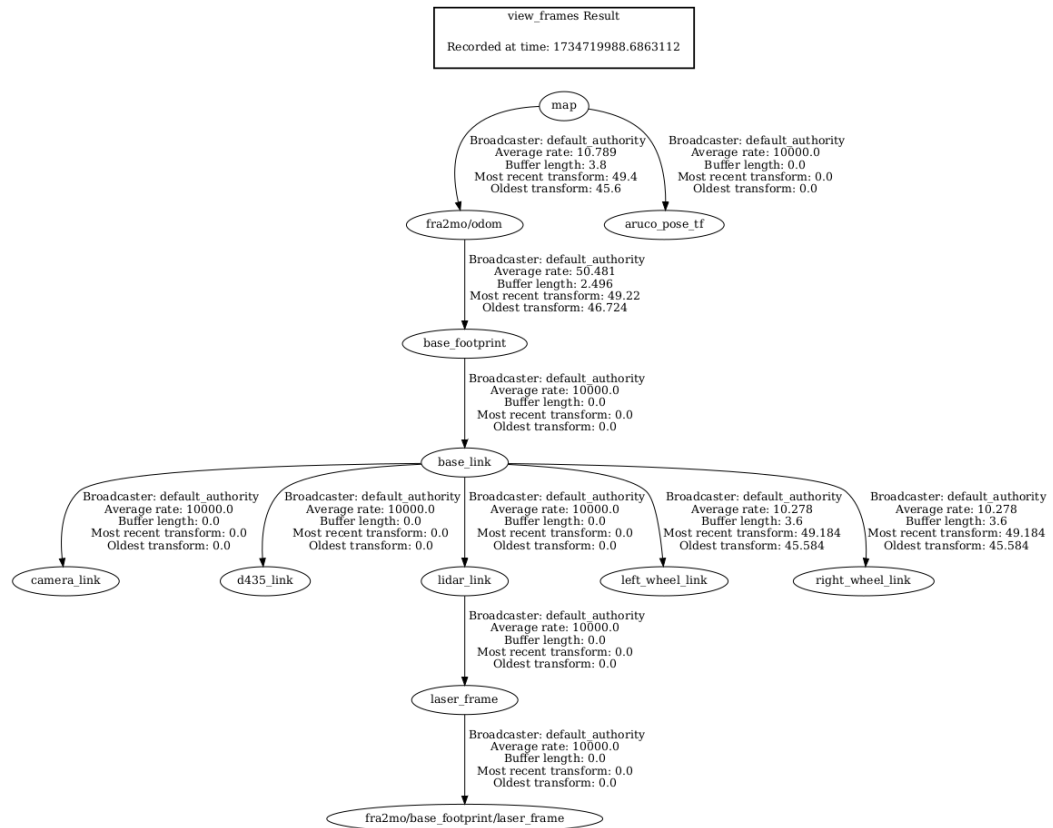


Figure 32: Frames

To conclude, here the link to some videos:

- Navigation and ArUco Pose TF from the Origin  
<https://youtu.be/DCAEMj6-fvY>
- Navigation and ArUco Pose TF from the given Initial Position  
<https://youtu.be/LvG0jWif7cI>