

Robotics Lab: Homework 4 Report

a.y. 2024/2025

Students

Annese Antonio	P38000296
Bosco Stefano	P38000245
Ercolanese Luciana	P38000197
Varone Emanuela	P38000284

Contents

1	Construct a gazebo world and spawn the mobile robot in a given pose	3
1.0.1	1.a Spawn the mobile robot in the given pose	3
1.0.2	1.b leonardo_race_field.sdf	3
1.0.3	1.c ArUco marker	4
2	Using the Nav2 Simple Commander API enable an autonomous navigation task	7
2.0.1	2.a Goals definition	7
2.0.2	2.b Sending the defined goals to the mobile platform in a given order	8
2.0.3	2.c Record a bagfile and plot the executed robot trajectory in the XY plane . . .	10
3	Map the environment tuning the navigation stack's parameters	13
3.0.1	3.a Getting a complete map of the environment	13
3.0.2	3.b Changing the parameters of the navigation config	14
3.0.3	3.c Comment on the results	17
4	Vision-based navigation of the mobile platform	19
4.0.1	4.a Running both the navigation and the aruco_ros node	19
4.0.2	4.b 2D navigation task	19
4.0.3	4.c Aruco pose as TF	23
5	GitHub repositories links:	26

Goal: control a mobile robot to follow a trajectory

This document contains a report of Homework 4 of the Robotics Lab class. The goal of this homework is to implement an autonomous navigation software framework to control a mobile robot.

1 Construct a gazebo world and spawn the mobile robot in a given pose

In this section, is displayed how construct a gazebo world and spawn the mobile robot in a given pose.

1.0.1 1.a Spawn the mobile robot in the given pose

To spawn the mobile robot in the world leonardo_race_field in the pose:

$$x = -3 \text{ m}, \quad y = 3.5 \text{ m}, \quad Y = -90 \text{ deg},$$

with respect to the map frame, we modified gazebo_fra2mo.launch.py file, as seen in Fig.1:

```
#1a
position = [-3.0, 3.5, 0.100]
orientation = [0.0, 0.0, -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]),
                "-R", str(orientation[0]),
                "-P", str(orientation[1]),
                "-Y", str(orientation[2]), ]
)
```

Figure 1: mobile robot pose

1.0.2 1.b leonardo_race_field.sdf

Into the leonardo_race_field.sdf file we moved the obstacle 9 in position:

$$x = -3 \text{ m}, \quad y = -3.3 \text{ m}, \quad z = 0.1 \text{ m}, \quad Y = 90 \text{ deg}.$$

as follows (Fig.2):

```

<!-- 1b.-->
<include>
  <name>obstacle_09</name>
  <pose> -3 -3.3 0.1 0 0 1.57</pose>
  <uri>model://obstacle_09</uri>
</include>

```

Figure 2: obstacle 9

1.0.3 1.c ArUco marker

The objective of this subsection is to 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.

First of all, we generated the ArUco marker number 115 (here <https://chev.me/arucogen/>), which appears like this (Fig.3):

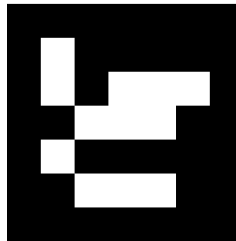


Figure 3: aruco 115

Then, inside the rl_fra2mo_description/model/marker_155 folder we created model.sdf file, which defines a model to simulate the ArUco tag in Gazebo (Fig.4):

```

<?xml version="1.0" encoding="UTF-8"?>
<sdf version='1.9'>
  <model name='arucotag'>
    <static>true</static>
    <pose>0 0 0 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</diffuse>
          <specular>0.4 0.4 0.4 1</specular>
          <pbr>
            <metal>
              <albedo_map>model://marker_115/aruco_115.png</albedo_map>
            </metal>
          </pbr>
        </material>
      </visual>
    </link>
  </model>
</sdf>

```

Figure 4: model.sdf

To ensure that the robot can see the ArUco, we added a camera to the robot's lidar_link (Fig.5-Fig.6):

```

<?xml version="1.0"?>
<robot xmlns:xacro="http://www.ros.org/wiki/xacro">

  <xacro:macro name="my_camera" params="parent">

    <joint name="camera_joint" type="fixed">
      <parent link="${parent}"/>
      <child link="camera_link"/>
      <origin xyz="0.0 0.0 0.05" rpy="0 0 0"/>
    </joint>

    <link name="camera_link">
      <visual>
        <geometry>
          <box size="0.01 0.01 0.01"/>
        </geometry>
        <material name="blue">
          <color rgba="0 0 1 1"/>
        </material>
      </visual>
    </link>

    <joint name="camera_optical_joint" type="fixed">
      <parent link="camera_link"/>
      <child link="camera_link_optical"/>
      <origin xyz="0.0 0.0 0.0" rpy="${-pi/2} 0 ${-pi/2}"/>
    </joint>

    <link name="camera_link_optical"></link>

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

    <gazebo reference="camera_link">
      <sensor name="camera" type="camera">
        <pose> 0 0 0 0 0 0 </pose>
        <camera>
          <horizontal_fov>1.047</horizontal_fov>
          <image>
            <width>320</width>
            <height>240</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>
  </xacro:macro>
</robot>

```

Figure 5: camera.xacro

```

<?xml version="1.0"?>

<robot xmlns:xacro="http://www.ros.org/wiki/xacro" name="fra2mo" >
  <xacro:include filename="$(find rl_fra2mo_description)/urdf/fra2mo_base_macro.xacro"/>
  <xacro:include filename="$(find rl_fra2mo_description)/urdf/lidar_gazebo_macro.xacro"/>
  <xacro:include filename="$(find rl_fra2mo_description)/urdf/camera.xacro"/>

  <xacro:property name="LIDAR" value="True"/>

  <!-- Diff Drive Robot Base -->
  <xacro:fra2mo_base/>

  <!-- LIDAR Sensor -->
  <xacro:if value="$(LIDAR)">
    <xacro:lidar_gazebo_sensor parent="lidar_link"/>
  </xacro:if>

  <xacro:my_camera parent="lidar_link"/>

</robot>

```

Figure 6: fra2mo.urdf.xacro

Next, we included the ArUco model within the leonardo_race_field.sdf file (Fig.7):

```

<!-- lc -->
<include>
  <uri>
    | model://marker_115
  </uri>
  <name>arucotag</name>
  <pose>-3.60 -0.69 0.36 2.00 1.53 -2.57</pose>
</include>

```

Figure 7: leonardo_race_field.sdf

The pose of the Aruco, specified by the `<pose>` tag, was accurately chosen to ensure that the marker:

- appears on obstacle 9;
- is visible by the mobile robot's camera when it comes in the proximity of the object (Fig.8).

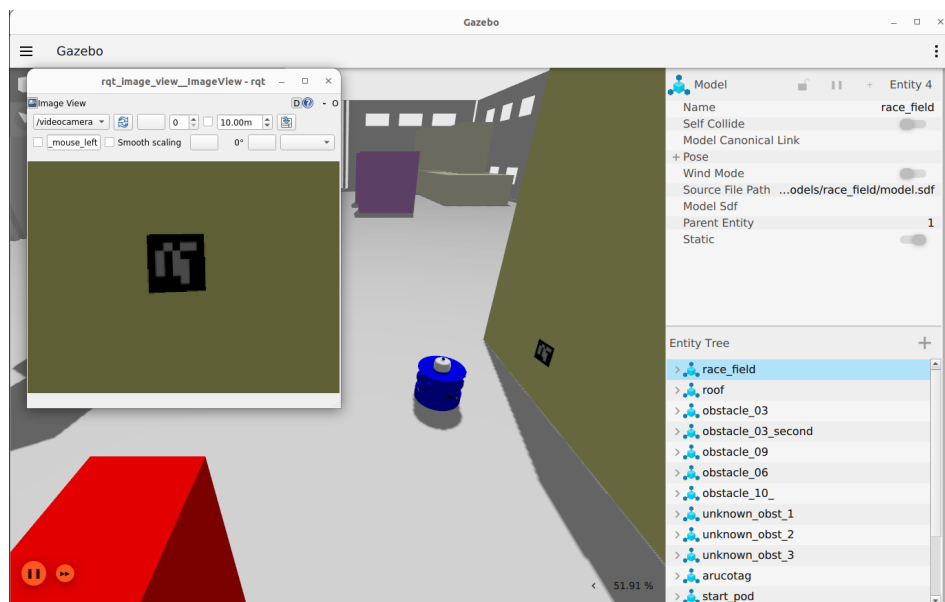


Figure 8: Gazebo simulation

2 Using the Nav2 Simple Commander API enable an autonomous navigation task

The objective of this chapter is to define 4 goals, send them to the mobile platform, record a bagfile of the executed robot trajectory and plot it in the XY plane.

2.0.1 2.a Goals definition

First of all, we created a goal.yaml file in which are defined the four goals, as seen in Fig.9:

```
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: 0.2588
  w: 0.9659
- 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: 0.382683
  w: 0.923880
```

Figure 9: goal.yaml

2.0.2 2.b Sending the defined goals to the mobile platform in a given order

To send the defined goals to the mobile platform in a given order (Goal3 \rightarrow Goal4 \rightarrow Goal2 \rightarrow Goal1), we modified the follow_waypoint.py file in this way:

- First, we imported the goal.yaml file, in which the four goals are defined, as seen in Fig.10:

```
waypoints = yaml.safe_load(
    open(os.path.join(get_package_share_directory('rl_fra2mo_description'), "config", "goal.yaml"))
)
```

Figure 10: goal.yaml loaded in the follow_waypoint.py file

- Since the poses specified in the four goals are with respect to the map frame, it is necessary to translate them into the odometry frame. To ensure this, we first created an initial_pose.yaml file in which the initial pose of the robot is specified, as seen in Fig.11:

```
rl24fold > rl_fra2mo_description > config > ! initial_pose.yaml
1  initial_pose:
2      position:
3          x: -3.0
4          y: 3.5
5          z: 0.0
6      orientation:
7          x: 0.0
8          y: 0.0
9          z: -0.707
10         w: 0.707
```

Figure 11: initial_pose.yaml

Then, we imported the initial_pose.yaml file in the follow_waypoints.py file, as seen in Fig.12:

```
#import initial pose
initial_pose = yaml.safe_load(
    open(os.path.join(get_package_share_directory('rl_fra2mo_description'), "config", "initial_pose.yaml"))
)
```

Figure 12: initial_pose.yaml loaded in the follow_waypoint.py file

Next the position and orientation were transformed from the global reference frame , map, to the local reference frame, odom, as seen in Fig.13:


```

def create_pose(transform):
    # Parametri della trasformazione (da mappa a odometria)
    initial_translation = {"x": -3, "y": 3.5}
    initial_rotation = -math.pi / 2 # -90° in radianti

    # Estraggo posizione e orientamento dal transform
    map_position = transform["position"]
    map_orientation = transform["orientation"]

    # Trasformo la posizione dal frame mappa al frame odometria
    translated_x = map_position["x"] - initial_translation["x"]
    translated_y = map_position["y"] - initial_translation["y"]

    # Applico la rotazione inversa
    odom_x = math.cos(-initial_rotation) * translated_x - math.sin(-initial_rotation) * translated_y
    odom_y = math.sin(-initial_rotation) * translated_x + math.cos(-initial_rotation) * translated_y

    # L'orientamento (quaternione) deve essere ruotato per tener conto della rotazione iniziale
    # Funzione per ruotare un quaternione di un angolo attorno all'asse Z
    def rotate_quaternion_z(q, theta):
        # Costruisco il quaternione della rotazione
        rot_q = {
            "x": 0,
            "y": 0,
            "z": math.sin(theta / 2),
            "w": math.cos(theta / 2)
        }

        # Moltiplico i quaternioni: rot_q * q
        new_q = {
            "x": rot_q["w"] * q["x"] + rot_q["x"] * q["w"] + rot_q["y"] * q["z"] - rot_q["z"] * q["y"],
            "y": rot_q["w"] * q["y"] - rot_q["x"] * q["z"] + rot_q["y"] * q["w"] + rot_q["z"] * q["x"],
            "z": rot_q["w"] * q["z"] + rot_q["x"] * q["y"] - rot_q["y"] * q["x"] + rot_q["z"] * q["w"],
            "w": rot_q["w"] * q["w"] - rot_q["x"] * q["x"] - rot_q["y"] * q["y"] - rot_q["z"] * q["z"]
        }
        return new_q

    # Ruoto il quaternione dell'orientamento
    odom_orientation = rotate_quaternion_z(map_orientation, -initial_rotation)

    # Creo il PoseStamped nel frame odometria
    pose = PoseStamped()
    pose.header.frame_id = 'map' # Ora nel frame odometria
    pose.header.stamp = navigator.get_clock().now().to_msg()
    pose.pose.position.x = odom_x
    pose.pose.position.y = odom_y
    pose.pose.position.z = map_position["z"] # La quota rimane invariata
    pose.pose.orientation.x = odom_orientation["x"]
    pose.pose.orientation.y = odom_orientation["y"]
    pose.pose.orientation.z = odom_orientation["z"]
    pose.pose.orientation.w = odom_orientation["w"]

    return pose

```

Figure 13: create_pose function

The `create_pose(transform)` function (Fig.12) performs a transformation of the coordinates and orientation of an object (goal) from the global reference frame `map` to the local reference frame `odom`. First the transformation parameters are initialized (`initial_translation` and `initial_rotation`). Next, the position and orientation of a point in the frame `map` is extracted (`map_position` and `map_orientation`). The initial offset `(-3,3.5)` is then subtracted from the global position `(x,y)` in the frame `map`. This shifts the coordinates so that the `odom` frame has correct relative origin with respect to the `map` frame. An inverse rotation `(-90°)` is then applied to the translated position, to correctly position the point in the `odom` frame. The `rotate_quaternion_z` function creates a new quaternion representing a rotation about the `z` axis by an angle `theta`. The resulting new quaternion is calculated by multiplying `rot_q` by the original quaternion `q`. Multiplication of quaternions allows the rotations to be combined. The orientation of the point is rotated by `-90°` to align it with the `odom` frame. Finally, a `PoseStamped` object, that contains the transformed coordinates (position and orientation) in the `odom` frame, is created.

At the end we implemented the code (Fig.14) that guides the robot through the sequence of reordered waypoints by following these steps:

- Transforms the waypoints from the YAML file into coordinates readable by the robot;
- Reorders the waypoints according to a specific sequence (Goal3 → Goal4 → Goal2 → Goal1);
- Sends the list to the Nav2 navigation system;
- Monitors progress and provides feedback;

-Verifies the result at the end of navigation.

```

main()
all_goal_poses = list(map(create_pose, waypoints["waypoints"]))

# Reorder the goals: Goal 3 → Goal 4 → Goal 2 → Goal 1
reordered_goal_indices = [2, 3, 1, 0] # Python indexing starts at 0
goal_poses = [all_goal_poses[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():

    # 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)

```

Figure 14: code that guides the robot through the sequence of reordered waypoints

The attached video ("follow_waypoints.webm") shows the executed robot trajectory.

2.0.3 2.c Record a bagfile and plot the executed robot trajectory in the XY plane

To record the bagfile we use the command: "ros2 bag record -o my_recording /pose". This command records pose data in a .bag file once the robot starts its autonomous movement towards the 4 goals. The method pursued to extrapolate the plot consisted of using MATLAB (Fig.15):

```

% Definizione delle variabili
fileName = "pose_plot"; % Nome per l'immagine salvata
recName = "my_recording_0.db3"; % Nome del file .db3

% Percorsi delle cartelle
imgFolderPath = "C:\Users\anton\OneDrive\Magistrale\Robotic Lab\immagini4";
recFolderPath = "C:\Users\anton\OneDrive\Magistrale\Robotic Lab\my_recording4";

% Caricamento del file bag
% Utilizzo di ros2bagreader per leggere il file .db3
bag = ros2bagreader(fullfile(recFolderPath, recName));

% Visualizza i topic disponibili nel file bag
disp('Topic disponibili nel file bag:');
disp(bag.AvailableTopics);

% Seleziona il topic '/pose'
msgs = readMessages(select(bag, 'Topic', '/pose'));

% Copia dei messaggi
% Numero di messaggi
n = numel(msgs);

% Pre-alloca matrici per posizione e orientamento
positions = zeros(n, 3); % x, y, z
orientations = zeros(n, 4); % quaternion: x, y, z, w

% Copia i valori di posizione e orientamento dai dati dei messaggi
for i = 1:n
    % Estraggo la posizione
    positions(i, :) = [msgs{i}.pose.pose.position.x, ...
                      msgs{i}.pose.pose.position.y, ...
                      msgs{i}.pose.pose.position.z];

    % Estraggo l'orientamento (quaternion)
    orientations(i, :) = [msgs{i}.pose.pose.orientation.x, ...
                          msgs{i}.pose.pose.orientation.y, ...
                          msgs{i}.pose.pose.orientation.z, ...
                          msgs{i}.pose.pose.orientation.w];
end

% Definizione del tempo
loop_rate = 50; % Frequenza di campionamento ipotetica (regola se necessario)
n_of_point = 1:n;
time = n_of_point / loop_rate;

% Plot della posizione
figure;
hold on;

plot(time, positions(:, 1), 'DisplayName', 'Position X', 'LineWidth', 2);
plot(time, positions(:, 2), 'DisplayName', 'Position Y', 'LineWidth', 2);
plot(time, positions(:, 3), 'DisplayName', 'Position Z', 'LineWidth', 2);

xlabel('Time (s)');
ylabel('Position');
title('Position Over Time');
legend show;
grid on;
hold off;

% Salva il grafico della posizione
saveas(gcf, fullfile(imgFolderPath, fileName + "_position.png"));

% Plot dell'orientamento (quaternion)
figure;
hold on;

plot(time, orientations(:, 1), 'DisplayName', 'Quaternion X', 'LineWidth', 2);
plot(time, orientations(:, 2), 'DisplayName', 'Quaternion Y', 'LineWidth', 2);
plot(time, orientations(:, 3), 'DisplayName', 'Quaternion Z', 'LineWidth', 2);
plot(time, orientations(:, 4), 'DisplayName', 'Quaternion W', 'LineWidth', 2);

xlabel('Time (s)');
ylabel('Orientation (Quaternion)');
title('Orientation Over Time');
legend show;
grid on;
hold off;

% Salva il grafico dell'orientamento
saveas(gcf, fullfile(imgFolderPath, fileName + "_orientation.png"));

```

Figure 15: Script MATLAB

The figures (Fig.16 - Fig.17) provided below show the plots obtained:

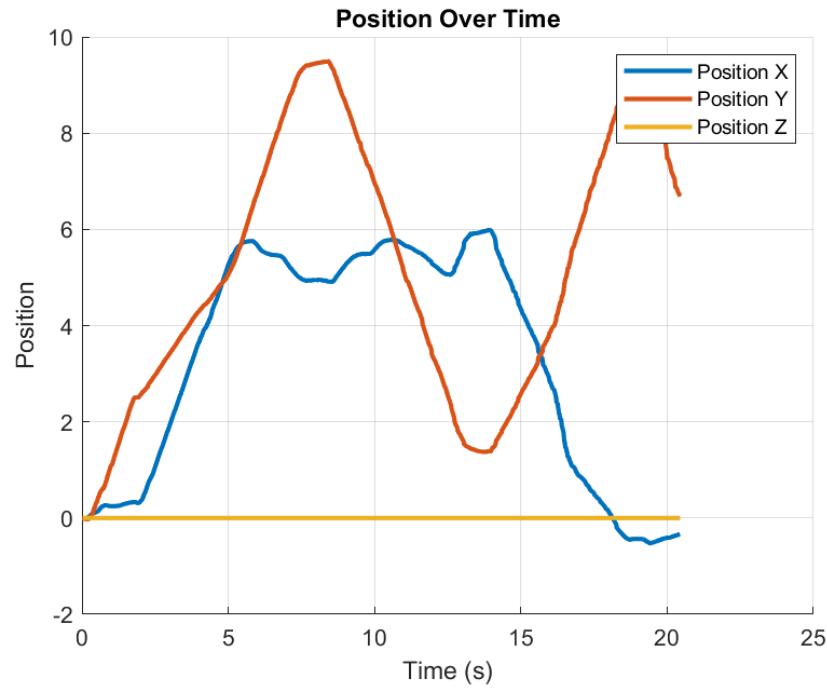


Figure 16: Position Over Time

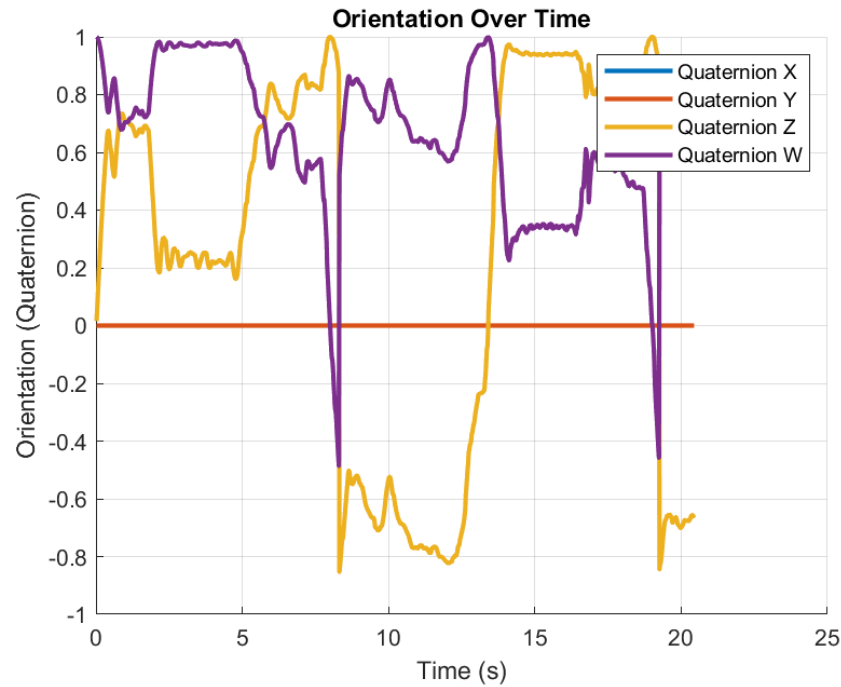


Figure 17: Orientation Over Time

3 Map the environment tuning the navigation stack's parameters

The objective of this chapter is to get a complete map of the environment, change the parameters of the navigation config and comment the results obtained.

3.0.1 3.a Getting a complete map of the environment

To get a complete map of the environment, we proceeded as described below:

- first, we started the simulation in Gazebo (command: `ros2 launch rl_fra2mo_description gazebo_fra2mo.launch.py`);
- next, we run Rviz and selected `slam_view.rviz` as configuration;
- finally, we mapped the environment by using the 2D Goal Pose interface provided by Rviz and saved the map obtained by running the `ros2 run nav2_map_server map_saver_cli -f map_3a` command.

In this way, two files are automatically generated in the map folder:

- `map_3a.pgm`: image file representing the map (Fig.18);

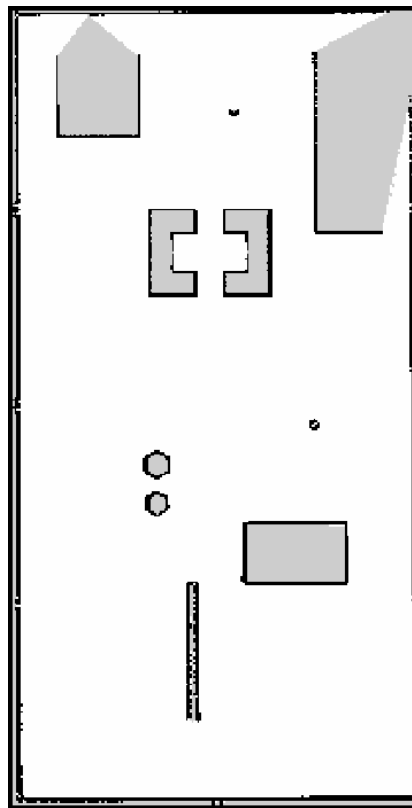


Figure 18: map_3a

- `map_3a.yaml`: map configuration file to define its parameters, which is essential to ensure that the robot correctly interprets the environment during navigation (Fig.19).

```

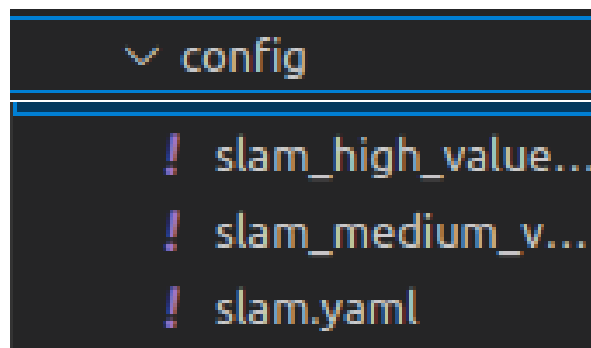
rl24fold > rl_fra2mo_description > maps > ! map_3a.yaml
1  image: map_3a.pgm
2  mode: trinary
3  resolution: 0.05
4  origin: [-1.63, -7.16, 0]
5  negate: 0
6  occupied_thresh: 0.65
7  free_thresh: 0.25

```

Figure 19: map_3a.yaml

3.0.2 3.b Changing the parameters of the navigation config

Starting with SLAM, to fulfill the assigned task, we decided to generate in the config folder 3 different YAML files, as seen in Fig.20:



```

v config
! slam_high_value...
! slam_medium_v...
! slam.yaml

```

Figure 20: Slam config

in which the parameters:

- minimum_travel_distance
- minimum_travel_heading2
- resolution
- transform_publish_period

were modified by assigning them:

- small (slam.yaml) values (Fig.21):

```

rl24fold > rl_fra2mo_description > config > ! slam.yaml
minimum_travel_distance: 0.01
minimum_travel_heading: 0.01
resolution: 0.05
transform_publish_period: 0.02 #

```

Figure 21: slam.yaml

- intermediate (slam_medium_values.yaml) values (Fig22):

```
rl24fold > rl_fra2mo_description > config > ! slam_medium_values.yaml
minimum_travel_distance: 0.05 #increased
minimum_travel_heading: 0.05 #increased t
resolution: 0.1 #increased for a more pre
transform_publish_period: 0.1 #increased t
```

Figure 22: slam_medium_values.yaml

- high (slam_high_values.yaml) values (Fig.23):

```
rl24fold > rl_fra2mo_description > config > ! slam_high_values.yaml
minimum_travel_distance: 0.1 #increased to reduce th
minimum_travel_heading: 0.1 #increased to reduce sys
resolution: 0.5 #increased for a more precise map
transform_publish_period: 0.5 #increased to make the
```

Figure 23: slam_high_values.yaml

Turning to exploration, to fulfill the assigned task, we decided to generate in the config folder 3 different YAML files, as seen in Fig.24:

```

v config
! explore_high_va...
! explore_low_val...
! explore.yaml
```

Figure 24: Exploration config

in which the parameters:

-inflation_radius

-cost_scaling_factor

for global and local costmaps, were modified by assigning them:

- small (explore_low_values.yaml) values (Fig.25):

```
global_costmap:
  global_costmap:
    ros__parameters:
      enabled: true
      inflation_radius: 0.5 #reduced to
      cost_scaling_factor: 2.0 #reduced
local_costmap:
  local_costmap:
    ros__parameters:
      enabled: true
      inflation_radius: 0.5 #reduced to
      cost_scaling_factor: 2.0 #reduced
```

Figure 25: explore_low_values.yaml

- intermediate (explore.yaml) values (Fig.26):

```
global_costmap:
  global_costmap:
    ros__parameters:
      enabled: true
      inflation_radius: 0.75
      cost_scaling_factor: 3.0
local_costmap:
  local_costmap:
    ros__parameters:
      enabled: true
      inflation_radius: 0.75
      cost_scaling_factor: 3.0
```

Figure 26: explore.yaml

- high (explore_high_values.yaml) values (Fig.27):


```

global_costmap:
  global_costmap:
    ros__parameters:
      inflation_radius: 1.0 #increased to
      cost_scaling_factor: 4.0 #Increased
local_costmap:
  local_costmap:
    ros__parameters:
      inflation_radius: 1.0 #increased to
      cost_scaling_factor: 4.0 #Increased

```

Figure 27: explore_high_values.yaml

3.0.3 3.c Comment on the results

To analyze the results obtained from the changes made in the previous paragraph, it was necessary to modify the launch files as follows:

- fra2mo_slam.launch.py:

```

slam_params_file_arg = DeclareLaunchArgument([
    'slam_params_file',
    default_value=PathJoinSubstitution(
        [FindPackageShare("rl_fra2mo_description"), 'config', 'slam.yaml']
    ),
    description='Full path to the ROS2 parameters file to use for the slam_toolbox node',
])

return LaunchDescription([use_sim_time_arg, slam_params_file_arg, slam_node])

```

Figure 28: Statement of Launching Arguments – slam.yaml

```

slam_params_file_arg_medium = DeclareLaunchArgument(
    'slam_params_file',
    default_value=PathJoinSubstitution(
        [FindPackageShare("rl_fra2mo_description"), 'config', 'slam_medium_values.yaml']
    ),
    description='Full path to the ROS2 parameters file to use for the slam_toolbox node',
)

return LaunchDescription([use_sim_time_arg, slam_params_file_arg_medium, slam_node])

```

Figure 29: Statement of Launching Arguments – slam_medium_values.yaml

```

slam_params_file_arg_high = DeclareLaunchArgument(
    'slam_params_file',
    default_value=PathJoinSubstitution(
        [FindPackageShare("rl_fra2mo_description"), 'config', 'slam_high_values.yaml']
    ),
    description='Full path to the ROS2 parameters file to use for the slam_toolbox node',
)

return LaunchDescription([use_sim_time_arg, slam_params_file_arg_high, slam_node])

```

Figure 30: Statement of Launching Arguments – slam_high_values.yaml

- `fra2mo_explore.launch.py`:

```
declare_params_file_cmd_low = DeclareLaunchArgument([
    'params_file',
    default_value=PathJoinSubstitution([fra2mo_dir, 'config', 'explore_low_values.yaml']),
    description='Full path to the ROS2 parameters file to use for all launched nodes',
])

return LaunchDescription(
    [
        declare_params_file_cmd_low,
```

Figure 31: Statement of Launching Arguments – `explore_low_values.yaml`

```
declare_params_file_cmd = DeclareLaunchArgument(
    'params_file',
    default_value=PathJoinSubstitution([fra2mo_dir, 'config', 'explore.yaml']),
    description='Full path to the ROS2 parameters file to use for all launched nodes',
)

return LaunchDescription(
    [
        declare_params_file_cmd,
```

Figure 32: Statement of Launching Arguments – `explore.yaml`

```
declare_params_file_cmd_high = DeclareLaunchArgument(
    'params_file',
    default_value=PathJoinSubstitution([fra2mo_dir, 'config', 'explore_high_values.yaml']),
    description='Full path to the ROS2 parameters file to use for all launched nodes',
)

return LaunchDescription(
    [
        declare_params_file_cmd_high,
```

Figure 33: Statement of Launching Arguments – `explore_high_values.yaml`

From the analysis of the results obtained, we found that:

- A higher value of the `'minimum_travel_distance'` parameter reduces the frequency with which the system updates the position, improving performance but reducing accuracy.
- Increasing the value of `'minimum_travel_heading'` reduces the sensitivity of the system to small changes in orientation, improving efficiency but risking losing details in the map.
- Higher `'resolution'` improved the accuracy and detail of the map, but planning took longer.
- The `'transform_publish_period'` indicates how often the system updates and transmits data, such as transformations (tf), maps, or other useful navigation information. A lower value means that updates occur more frequently, increasing the load on the system, while a higher value reduces that load.
- Reducing `'inflation_radius'` made movement more accurate in tight spaces, but increased the risk of collisions. Conversely, increasing `'inflation_radius'` improved safety, but made navigation in tight spaces more difficult.
- Increasing `'cost_scaling_factor'` improved planning time, but caused slower movements.

4 Vision-based navigation of the mobile platform

The objective of this chapter is to:

- create a launch file running both the navigation and the aruco_ros node;
- implement a 2D navigation task;
- publish the Aruco pose as TF.

4.0.1 4.a Running both the navigation and the aruco_ros node

To run both the navigation and the aruco_ros node, using the robot camera previously added to the robot model, we created in the launch folder a new launch file called `nav_aruco.launch.py` whose content is shown in Fig.34:

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

def generate_launch_description():

    other_launch_file_navigation = PathJoinSubstitution(
        [FindPackageShare("rl_fra2mo_description"), "launch", "fra2mo_explore.launch.py"]
    )

    include_other_launch_navigation = IncludeLaunchDescription(
        PythonLaunchDescriptionSource(other_launch_file_navigation)
    )

    other_launch_file_aruco = PathJoinSubstitution(
        [FindPackageShare("aruco_ros"), "launch", "aruco_cam.launch.py"]
    )

    include_other_launch_aruco = IncludeLaunchDescription(
        PythonLaunchDescriptionSource(other_launch_file_aruco)
    )

    return LaunchDescription([
        include_other_launch_navigation,
        include_other_launch_aruco
    ])

```

Figure 34: `nav_aruco.launch.py`

In Fig.34 the `aruco_cam.launch.py` is the launch file, belonging to the launch folder of the `aruco_ros` package, which configures and starts the ‘single’ node of the `aruco_ros` package, used for ArUco marker detection.

4.0.2 4.b 2D navigation task

To 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;
- return the robot to the initial position

we created a new file, named `vision_based_navigation.py`, inside the script folder of the `rl_fra2mo_description` package, which executes the three required tasks sequentially.

To send the robot in the proximity of obstacle 9, we defined 3 goals in a dedicated `.yaml` file, called `aruco_goal.yaml` (Fig.35):

```
config > f aruco_goal.yaml
1 waypoints:
2   - position:
3     x: 1.01
4     y: 3.50
5     z: 0.0
6   orientation:
7     x: 0.0
8     y: 0.0
9     z: 1.0
10    w: 0.0
11  - position:
12    x: -1.01
13    y: -0.31
14    z: 0.0
15  orientation:
16    x: 0.0
17    y: 0.0
18    z: 1.0
19    w: 0.0
20  - position:
21    x: -3.58
22    y: -0.10
23    z: 0.0
24  orientation:
25    x: 0.0
26    y: 0.0
27    z: -0.71934
28    w: 0.694658
```

Figure 35: `aruco_goal.yaml`

Then, inside the `vision_based_navigation.py`:

- is read the `aruco_goal.yaml` file containing waypoints information. It is converted into a dictionary structure, which is used to determine the positions and orientations of the waypoints to be reached by the robot (Fig.36).

```
# Carica i waypoints da YAML
waypoints = yaml.safe_load(
    open(os.path.join(get_package_share_directory('rl_fra2mo_description'), "config", "aruco_goal.yaml"))
)
```

Figure 36: send the robot in the proximity of obstacle 9

- the navigation system is initialised, specifying the 'smoother_server' locator, which is a component used to calculate smoother routes. Next, navigation to waypoints is started: (Fig.37).

```
def start_navigation(self):
    # Inizializza il navigator
    self.navigator.waitUntilNav2Active(localizer="smoother_server")

    # 1. Invia il robot vicino all'ostacolo 9
    self.navigate_to_waypoints()

def navigate_to_waypoints(self):
    all_goal_poses = list(map(self.create_pose, waypoints["waypoints"]))
    reordered_goal_indices = [0, 1, 2]
    goal_poses = [all_goal_poses[i] for i in reordered_goal_indices]

    # Segui i waypoint
    self.navigator.followWaypoints(goal_poses)

    # Attendi che il robot completi la navigazione
    while not self.navigator.isTaskComplete():
        feedback = self.navigator.getFeedback()
        if feedback:
            print(f'Executing current waypoint: {feedback.current_waypoint + 1}/{len(goal_poses)}')
```

Figure 37: send the robot in the proximity of obstacle 9

Once the robot has detected the ArUco marker, to retrieve its pose with respect to the map frame inside the `vision_based_navigation.py`:

- the `aruco_navigation_node` subscribes to the topic `'aruco_single_pose'` of the ArUco marker (Fig.38)

```
# Sottoscrizione al topic della posa dell'ArUco
self.aruco_pose_sub = self.create_subscription(
    PoseStamped,
    '/aruco_single/pose',
    self.aruco_pose_callback,
    10
)
```

Figure 38

- a function for waiting for marker detection is implemented (Fig.39)

```
def wait_for_marker_detection(self):
    # Aspetta fino a che il marker viene rilevato
    while not self.current_marker_pose:
        rclpy.spin_once(self)

    # Log della posa del marker ArUco nel frame "map"
    self.get_logger().info("ArUco marker position in map frame: ")
    # f"Position -> [x: {self.current_marker_pose.pose.position.x}, "
    # f"y: {self.current_marker_pose.pose.position.y}, z: {self.current_marker_pose.pose.position.z}], "
    # f"Orientation -> [x: {self.current_marker_pose.pose.orientation.x}, "
    # f"y: {self.current_marker_pose.pose.orientation.y}, "
    # f"z: {self.current_marker_pose.pose.orientation.z}, "
    # f"w: {self.current_marker_pose.pose.orientation.w}]"
    # f"Position -> [x: {self.current_marker_pose.transform.translation.x}, "
    # f"y: {self.current_marker_pose.transform.translation.y}, z: {self.current_marker_pose.transform.translation.z}], "
    # f"Orientation -> [x: {self.current_marker_pose.transform.rotation.x}, "
    # f"y: {self.current_marker_pose.transform.rotation.y}, "
    # f"z: {self.current_marker_pose.transform.rotation.z}, "
    # f"w: {self.current_marker_pose.transform.rotation.w}]"
```

Figure 39

- the position of the marker in the global coordinate system is calculated and a static transformation between the marker frame and the map is published (Fig.40).

```
def aruco_pose_callback(self, msg):
    aruco_x = msg.pose.position.x
    aruco_y = msg.pose.position.y
    aruco_z = msg.pose.position.z

    qx = msg.pose.orientation.x
    qy = msg.pose.orientation.y
    qz = msg.pose.orientation.z
    qw = msg.pose.orientation.w

    # Applica l'offset iniziale al frame map
    map_x = aruco_x + self.map_offset_x
    map_y = aruco_y + self.map_offset_y
    map_z = aruco_z

    # Creazione e pubblicazione della trasformazione statica
    transform_stamped = TransformStamped()
    transform_stamped.header.stamp = self.get_clock().now().to_msg()
    transform_stamped.header.frame_id = 'aruco_marker_frame'
    transform_stamped.child_frame_id = 'map'

    transform_stamped.transform.translation.x = map_x
    transform_stamped.transform.translation.y = map_y
    transform_stamped.transform.translation.z = map_z
    transform_stamped.transform.rotation.x = qx
    transform_stamped.transform.rotation.y = qy
    transform_stamped.transform.rotation.z = qz
    transform_stamped.transform.rotation.w = qw

    # Pubblica la trasformazione statica
    self.tf_broadcaster.sendTransform(transform_stamped)
    self.current_marker_pose = transform_stamped
```

Figure 40

In the following image (Fig.41), on the left, there is the output obtained by running the command "ros2 run tf2_ros tf2_echo aruco_marker_frame map", which returns the transformation from the aruco_marker_frame to the map frame, while, on the right, the pose of the aruco with respect to the map frame is printed out when running the aruco_navigation_node.

The image shows two terminal windows side-by-side. The left window displays the output of 'tf2_echo' showing translation, rotation in quaternion, RPY, and degrees, and a transformation matrix. The right window shows the output of 'aruco_navigation_node' displaying waypoint execution status and a detailed pose of the Aruco marker in the map frame, including position (x, y, z) and orientation (w, x, y, z).

Figure 41

Comparing the two outputs, we can see that the node has correctly calculated the transformation between the marker frame and the map.

Finally, to return the robot to the initial position inside the vision_based_navigation.py:

- is read the initial_pose.yaml file, in which is defined the initial_pose of the robot (Fig.42),

```
# Carica la posizione iniziale
initial_pose = yaml.safe_load(
    open(os.path.join(get_package_share_directory('rl_fra2mo_description'), "config", "initial_pose.yaml"))
)
```

Figure 42

- then the function "return_to_initial_position" is invoked, which assigns the initial position, read from the initial_pose.yaml file, as the robot's goal. (Fig.43).

```
class ArucoNavigationNode(Node):
    def return_to_initial_position(self):
        # Recupera la posa iniziale dal file YAML

        initial_position = initial_pose.get("initial_pose", None)
        if not initial_position:

            self.get_logger().error("Initial pose not defined in initial_pose.yaml.")
            return# Crea una singola posa di destinazione

        goal_pose = self.create_pose(initial_position)

        # Naviga verso la posa iniziale

        self.navigator.goToPose(goal_pose)

        # Attendi il completamento della navigazione
        while not self.navigator.isTaskComplete():

            feedback = self.navigator.getFeedback()
            if feedback:
                print(f"Returning to initial position: Distance remaining: {feedback.distance_remaining:.2f} meters.")

            # Verifica il risultato della navigazione

            result = self.navigator.getResult()
            if result == TaskResult.SUCCEEDED:

                self.get_logger().info("Successfully returned to the initial position.")
            else:

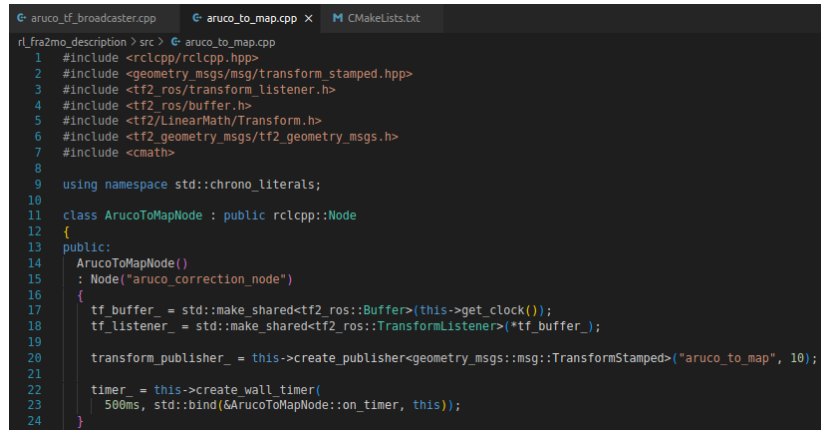
                self.get_logger().error("Failed to return to the initial position.")
```

Figure 43

The attached video ("2D_navigation.webm") shows the 2D navigation task executed by the robot.

4.0.3 4.c Aruco pose as TF

In order to calculate and publish the position of the ArUco marker in a global reference system, called world, we implemented a node named 'ArucoToMapNode' (Fig.44). The node uses TF transformations to calculate this position by combining several reference frames.



```

1  #include <rclcpp/rclcpp.hpp>
2  #include <geometry_msgs/msg/transform_stamped.hpp>
3  #include <tf2_ros/transform_listener.h>
4  #include <tf2_ros/buffer.h>
5  #include <tf2/LinearMath/Transform.h>
6  #include <tf2_geometry_msgs/tf2_geometry_msgs.h>
7  #include <cmath>
8
9  using namespace std::chrono_literals;
10
11 class ArucoToMapNode : public rclcpp::Node
12 {
13 public:
14   ArucoToMapNode()
15     : Node("aruco_correction_node")
16   {
17     tf_buffer_ = std::make_shared<tf2_ros::Buffer>(<this>->get_clock());
18     tf_listener_ = std::make_shared<tf2_ros::TransformListener>(*tf_buffer_);
19
20     transform_publisher_ = this->create_publisher<geometry_msgs::msg::TransformStamped>("aruco_to_map", 10);
21
22     timer_ = this->create_wall_timer(
23       500ms, std::bind(&ArucoToMapNode::on_timer, this));
24   }

```

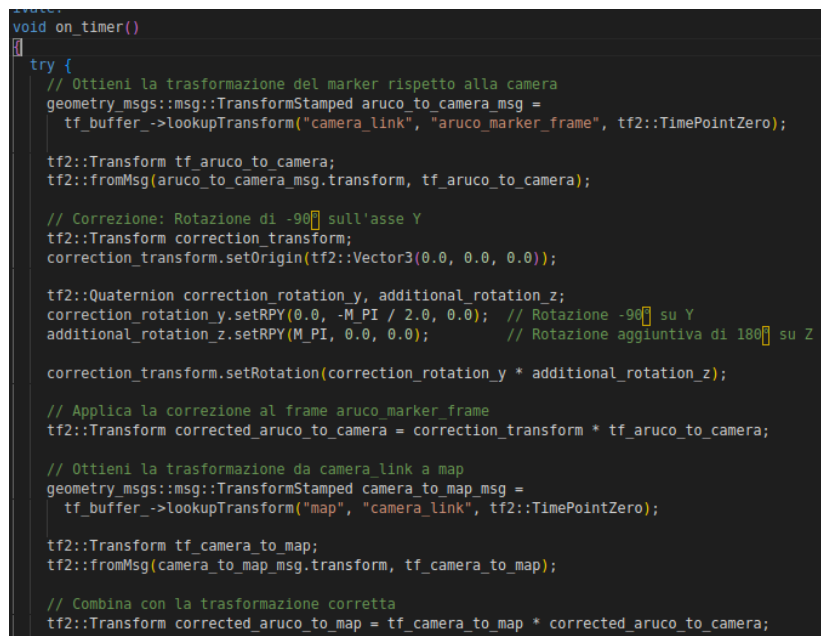
Figure 44: ArucoToMapNode

In Fig.44:

- first, a buffer and a listener are created to capture the TF transformations;
- then, a publisher is generated to publish the calculated transformation;
- finally, is set a timer, that calls the on_timer function every 500 ms.

The node (Fig.45) uses a `tf2_ros::TransformListener` to dynamically obtain transformations between frames. It is particularly interested in the transformations between:

- `camera_link` and `aruco_marker_frame`: position of the marker with respect to the camera,
- `map` and `camera_link`: position of the camera with respect to the global reference system.



```

void on_timer()
{
  try {
    // Ottieni la trasformazione del marker rispetto alla camera
    geometry_msgs::msg::TransformStamped aruco_to_camera_msg =
      tf_buffer_>lookupTransform("camera_link", "aruco_marker_frame", tf2::TimePointZero);

    tf2::Transform tf_aruco_to_camera;
    tf2::fromMsg(aruco_to_camera_msg.transform, tf_aruco_to_camera);

    // Correzione: Rotazione di -90° sull'asse Y
    tf2::Transform correction_transform;
    correction_transform.setOrigin(tf2::Vector3(0.0, 0.0, 0.0));

    tf2::Quaternion correction_rotation_y, additional_rotation_z;
    correction_rotation_y.setRPY(0.0, -M_PI / 2.0, 0.0); // Rotazione -90° su Y
    additional_rotation_z.setRPY(M_PI, 0.0, 0.0); // Rotazione aggiuntiva di 180° su Z

    correction_transform.setRotation(correction_rotation_y * additional_rotation_z);

    // Applica la correzione al frame aruco_marker_frame
    tf2::Transform corrected_aruco_to_camera = correction_transform * tf_aruco_to_camera;

    // Ottieni la trasformazione da camera_link a map
    geometry_msgs::msg::TransformStamped camera_to_map_msg =
      tf_buffer_>lookupTransform("map", "camera_link", tf2::TimePointZero);

    tf2::Transform tf_camera_to_map;
    tf2::fromMsg(camera_to_map_msg.transform, tf_camera_to_map);

    // Combina con la trasformazione corretta
    tf2::Transform corrected_aruco_to_map = tf_camera_to_map * corrected_aruco_to_camera;

```

Figure 45: on_timer

The `on_timer` function (Fig.45) applies a correction to the marker transformation (`aruco_marker_frame`) with respect to the camera (`camera_link`).

The correction consists of:

- a rotation of -90 degrees on the Y-axis,
- an additional rotation of 180 degrees on the Z axis.

This correction aligns the marker frame with the camera frame. The correct marker transformation is combined with the `camera_link` to map transformation (Fig.46).

```
// Applica la correzione al frame aruco_marker_frame
tf2::Transform corrected_aruco_to_camera = correction_transform * tf_aruco_to_camera;

// Ottieni la trasformazione da camera_link a map
geometry_msgs::msg::TransformStamped camera_to_map_msg =
    tf_buffer->lookupTransform("map", "camera_link", tf2::TimePointZero);

tf2::Transform tf_camera_to_map;
tf2::fromMsg(camera_to_map_msg.transform, tf_camera_to_map);

// Combina con la trasformazione corretta
tf2::Transform corrected_aruco_to_map = tf_camera_to_map * corrected_aruco_to_camera;

// Trasformazione da map a world (posizione e orientamento di map rispetto a world)
tf2::Transform map_to_world;
map_to_world.setOrigin(tf2::Vector3(-3.0, 3.5, 0.0)); // Traslazione da map a world
tf2::Quaternion q_map_to_world;
q_map_to_world.setRPY(0.0, 0.0, -M_PI / 2.0); // Rotazione di -90° su Z (yaw)
map_to_world.setRotation(q_map_to_world);

// Combina la trasformazione da map a world con la trasformazione di aruco
tf2::Transform aruco_to_world = map_to_world * corrected_aruco_to_map;

// Pubblica il risultato
geometry_msgs::msg::TransformStamped result;
result.header.stamp = this->get_clock()->now();
result.header.frame_id = "world"; // World come frame di riferimento
result.child_frame_id = "aruco_marker_frame";
result.transform = tf2::toMsg(aruco_to_world);

transform_publisher->publish(result);
```

Figure 46: `on_timer`

A fixed map to world transformation is also added, which includes:

- a translation of (-3.0,3.5,0.0),
- a rotation of -90 degrees in the Z axis.

The resulting transformation from `aruco_marker_frame` to world is published on the topic `aruco_to_map`. The result includes both the position and orientation of the ArUco marker in the global reference system world.

Finally, the `on_timer` function calculates the Euler angles (roll, pitch, yaw) from the final transformation for debugging purposes and logs the position and orientation of the marker in world (Fig.47).

```
// Calcola gli angoli roll, pitch e yaw per il debug
double roll, pitch, yaw;
tf2::Matrix3x3(aruco_to_world.getRotation()).getRPY(roll, pitch, yaw);

// Stampa i risultati
RCLCPP_INFO(this->get_logger(), "Aruco Marker in World: x=%.2f, y=%.2f, z=%.2f, roll=%.2f, pitch=%.2f, yaw=%.2f",
    aruco_to_world.getOrigin().x(),
    aruco_to_world.getOrigin().y(),
    aruco_to_world.getOrigin().z(),
    roll, pitch, yaw);

} catch (const tf2::TransformException &ex) {
    RCLCPP_WARN(this->get_logger(), "Could not get transform: %s", ex.what());
}
```

Figure 47: `on_timer`

After launching the node (with the command `ros2 run rl_fra2mo_description aruco_to_map`), by doing an echo on the topic `/aruco_to_map` (command: `ros2 topic echo /aruco_to_map`) it is possible to see the transformation (Fig.48-Fig.49-Fig.50):

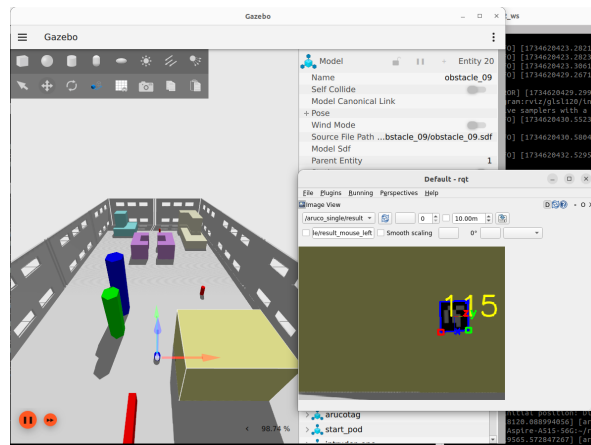


Figure 48: Aruco marker detected

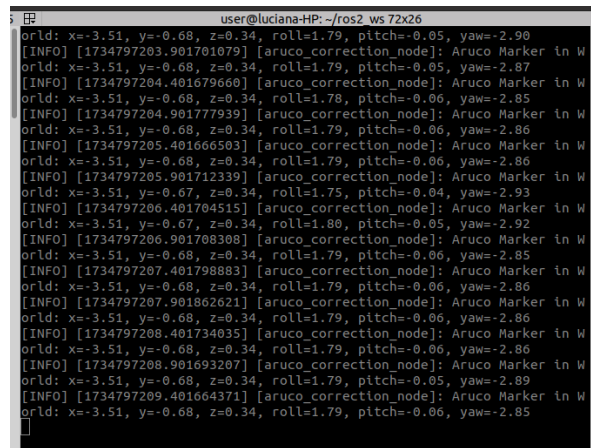


Figure 49: ros2 run rl_fra2mo_description aruco_to_map

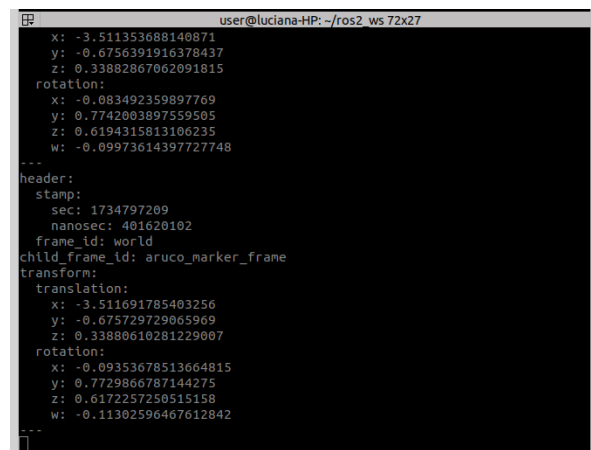


Figure 50: ros2 topic echo /aruco_to_map

NOTE: We have modified the launchfile "display_fra2mo.launch.py" such that Rviz automatically opens with the proper configuration.

5 GitHub repositories links:

Students

Annese Antonio	https://github.com/antann1/HomeworkRL_4.git
Bosco Stefano	https://github.com/SteBosco/HMW4_RL_2024.git
Ercolanese Luciana	https://github.com/LErcolanese/RL_Homework4.git
Varone Emanuela	https://github.com/Emanuela-var/Homework4_RL.git