UNIVERSITÀ DEGLI STUDI DI
NAPOLI FEDERICO II

Scuola Politecnica e delle Scienze di Base
Corso di Laurea Magistrale in Ingegneria dell'Automazione e Robotica

Report

HOMEWORK 4

Academic Year 2024/25

Professor
Mario Selvaggio

Candidates
Pasquale Farese matr. **P38000285**
Nello Di Chiaro matr. **P38000286**
Gianmarco Corrado matr. **P38000287**
Andrea Colapinto matr. **P38000309**

Abstract

This report will show how we succeeded in completing each task of the Homework 4.

The goal of this homework is to implement an autonomous navigation software framework to control a mobile robot. Starting from the provided package **rl_fra2mo_description** (`https://github.com/RoboticsLab2024/rl_fra2mo_description`), the homework is divided into four macro points:

- Change the pose of the mobile robot, of an obstacle and place an ArUco marker on it. Also, add a camera to the robot.
- Create a waypoints file and make the robot follow them.
- Map the environment modifying and tuning navigation parameters.
- Make the robot do a vision-based task.

Repositories

- Pasquale Farese: `https://github.com/PasFar/Homework-4.git`
- Nello Di Chiaro: `https://github.com/Nellodic34/Homework-4.git`
- Gianmarco Corrado: `https://github.com/giancorr/Homework-4.git`
- Andrea Colapinto: `https://github.com/colandrea02/Homework-4.git`

Contents

Chapter 1

Construction of the Gazebo world

The first part of the homework required us to change the initial pose of the mobile robot. In particular, it requires to make it spawn in the pose

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

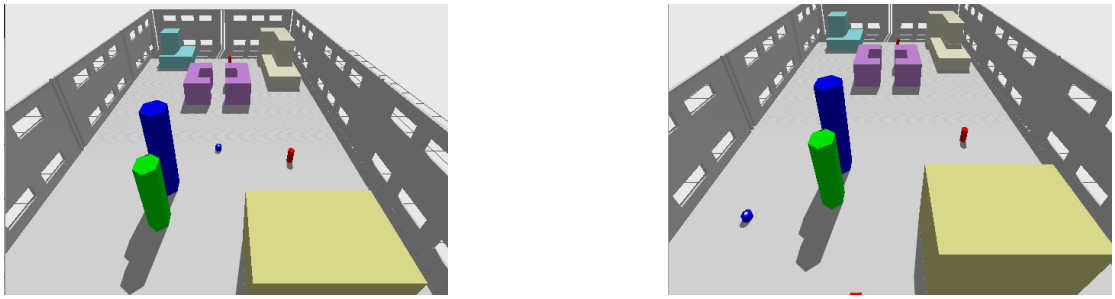
with respect to the map frame. In order to do this, in the Gazebo launcher we set the initial position to the one above specified and passed it to the spawner, as it is possible to see in Fig 1.1.

```
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 1.1: Code snippet of *gazebo_fra2mo.launch.py*.

In Fig 1.2 it is possible to notice the differences applied by this change.



(a) Initial pose. (b) Updated pose.

Figure 1.2: Before and after changing the mobile robot pose.

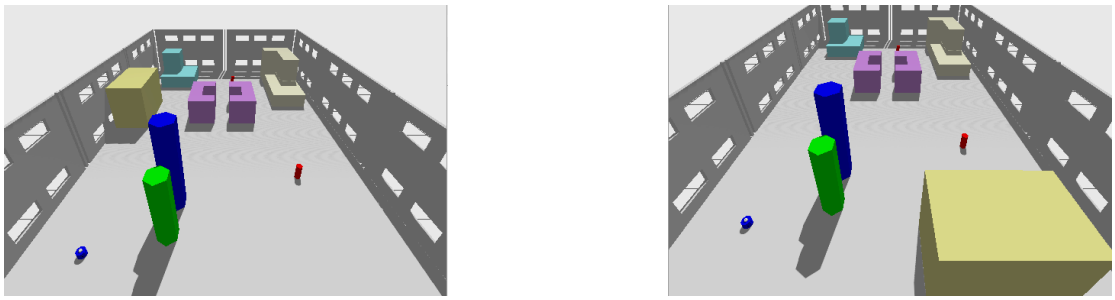
Additionally, we were asked to change the pose of the *obstacle 9*. In detail, we had to make it spawn in the following pose

$$x = -3m, \quad y = -3.3m, \quad Y = 90deg$$

In order to achieve this, we had to change its pose in the *leonardo_race_field.sdf* file, as showed in Fig 1.3.

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

Figure 1.3: Code snippet of *leonardo_race_field.sdf*.



(a) Initial pose.

(b) Updated pose.

Figure 1.4: Before and after changing the obstacle 9 pose.

In Fig 1.4 it is possible to notice the differences applied by this change.

Then, we were required to place an *ArUco* tag (specifically the number 115) on it, in order to be visible by a camera placed on the mobile robot. For this purpose, we first added the ArUco image to the obstacle 09 folder.

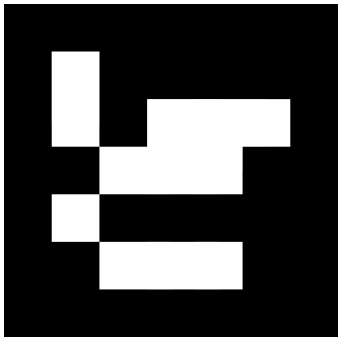


Figure 1.5: ArUco marker number 115.

In order to make it appear on the obstacle, we added a new fixed joint to the obstacle 9 and then a link representing the marker, loading the image on it, as shown in Fig 1.6 and Fig 1.7.

```
<!-- Nuovo link per l'ArUco tag -->
<link name="aruco_link">
  <pose>1 -0.01 0.2 1.57 0 0</pose> <!-- Posizione relativa all'ostacolo -->
  <visual name="aruco_visual">
    <geometry>
      <plane>
        <size>0.1 0.1</size> <!-- Dimensioni del piano del tag -->
      </plane>
    </geometry>
    <material>
      <diffuse>1 1 1 1</diffuse>
      <specular>0.4 0.4 0.4 1</specular>
      <pbr>
        <metal>
          <albedo_map>model://obstacle_09/aruco-115.png</albedo_map>
        </metal>
      </pbr>
    </material>
  </visual>
</link>
```

Figure 1.6: Code snippet of *obstacle_09.sdf*.

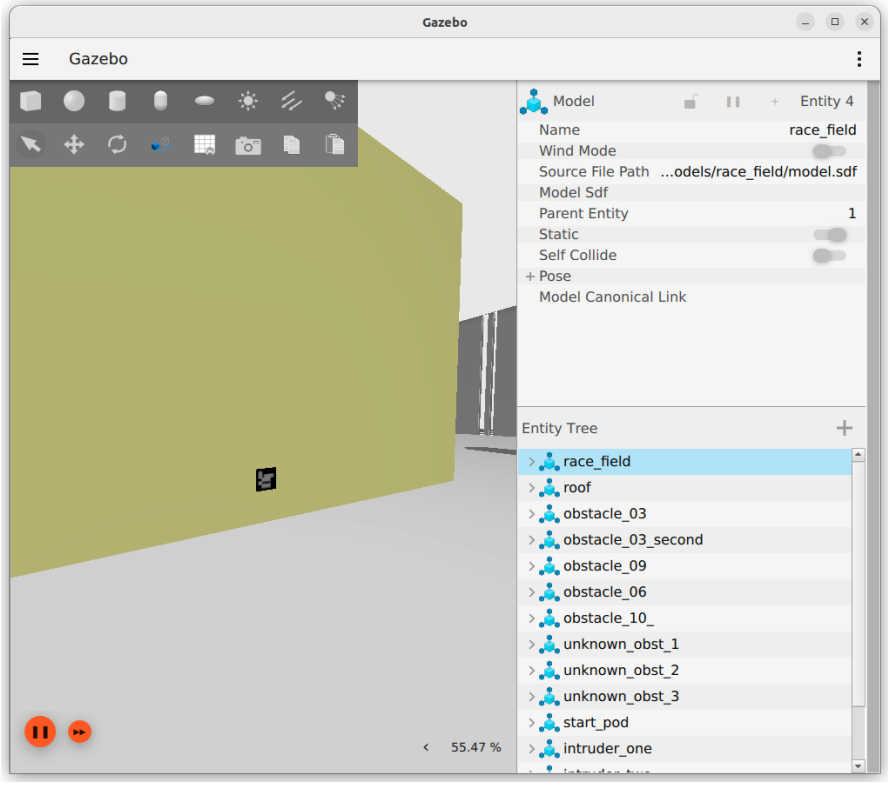


Figure 1.7: ArUco tag attached to obstacle 9.

Moreover, we added a camera to the mobile robot. To this end, we created a new *camera.xacro* file in the *urdf* folder, containing the macro description of the camera sensor that we want to implement on our mobile robot. Then, in the *fra2mo_base_macro.xacro* file we created a new fixed joint for the connection of the camera to the robot and then a link, representing the camera itself, attached to it. In the end, we called the macro in the *fra2mo_urdf.xacro* in order to construct the full robot loaded with the camera.

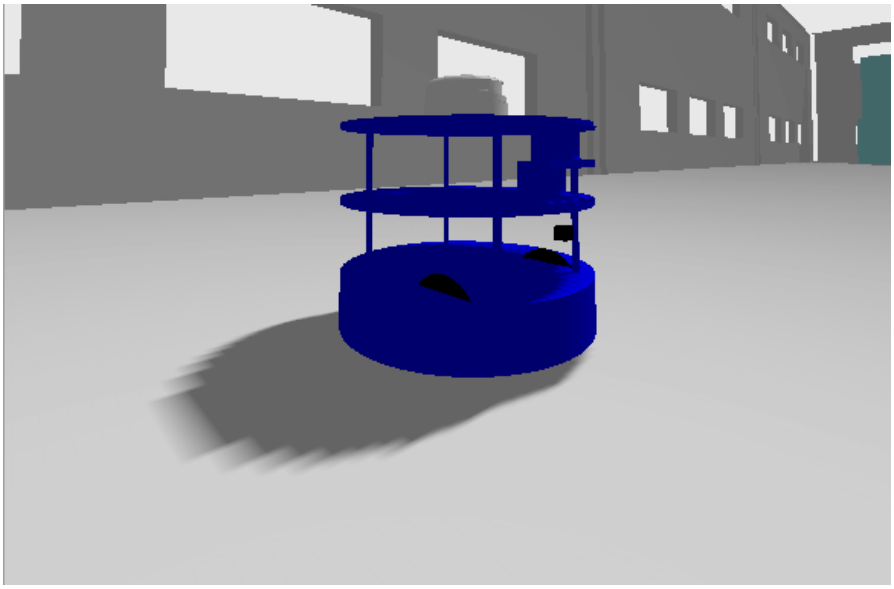


Figure 1.8: Full robot loaded with the camera.

At the end of this first point of the homework, it is possible to see how the world appears in Fig 1.9.

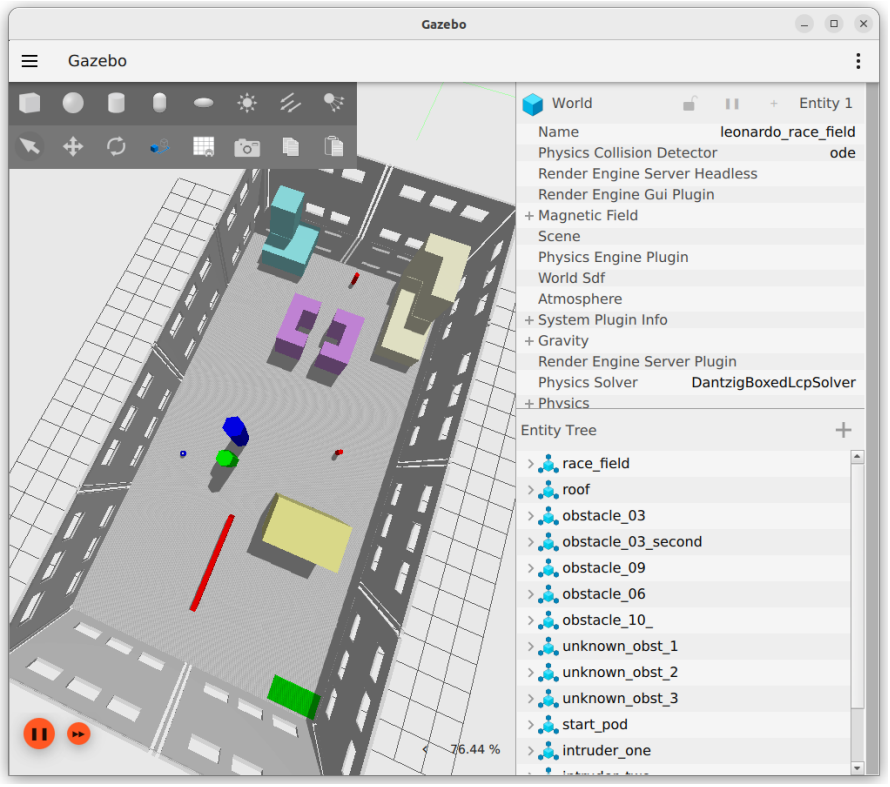


Figure 1.9: Gazebo world at the end of first task.

Chapter 2

Autonomous navigation task

The second part of this homework required us to create an autonomous navigation task using the Nav2 Simple Commander APIs. In order to do that, we created a new file, called *waypoints.yaml* in *config* folder, containing the provided waypoints with respect to the map frame:

- Goal_1: $x = 0m, \quad y = 3m, \quad Y = 0deg$
- Goal_2: $x = 6m, \quad y = 4m, \quad Y = 30deg$
- Goal_3: $x = 6.5m, \quad y = -1.4m, \quad Y = 180deg$
- Goal_4: $x = 1 - 6m, \quad y = -2.5m, \quad Y = 75deg$

The order of execution of these waypoints must be $3 \longrightarrow 4 \longrightarrow 2 \longrightarrow 1$ For this purpose, we modified the *follow_waypoints.py* file. First of

all, we created a function that enables us to transform each waypoint orientation from RPY notation to quaternion, as shown in Fig 2.1.

```
def rpy_to_quaternion(roll, pitch, yaw):

    cy = math.cos(yaw * 0.5)
    sy = math.sin(yaw * 0.5)
    cp = math.cos(pitch * 0.5)
    sp = math.sin(pitch * 0.5)
    cr = math.cos(roll * 0.5)
    sr = math.sin(roll * 0.5)

    qw = cy * cp * cr + sy * sp * sr
    qx = cy * cp * sr - sy * sp * cr
    qy = sy * cp * sr + cy * sp * cr
    qz = sy * cp * cr - cy * sp * sr

    return {
        'x': qx,
        'y': qy,
        'z': qz,
        'w': qw
    }
```

Figure 2.1: Code snippet of *follow_waypoints.py*.

Then we proceeded to load the correct *.yaml* file selected by the user in the command line argument.

```python
def load_waypoints(package_path):

    if sys.argv[1] == 'mapping':
        config_path = '/home/user/ros2_ws/src/rl_fra2mo_description/config/mapping.yaml'
    elif sys.argv[1] == 'waypoints':
        config_path = '/home/user/ros2_ws/src/rl_fra2mo_description/config/waypoints.yaml'
    elif sys.argv[1] == 'aruco':
        config_path = '/home/user/ros2_ws/src/rl_fra2mo_description/config/aruco_waypoint.yaml'
    else:
        print("Action not well-specified")
        return

    try:
        with open(config_path, 'r') as file:
            waypoints_data = yaml.safe_load(file)
        return waypoints_data.get('waypoints', [])
    except FileNotFoundError:
        print(f"Errore: File {config_path} non trovato!")
        return []
    except Exception as e:
        print(f"Errore durante il caricamento dei waypoint: {e}")
        return []
```

Figure 2.2: Code snippet of *follow_ waypoints.py*.

Additionally, because we moved the initial position of the robot, its position does not coincide anymore with the center of the map. In order to change accordingly the waypoints with this movement, we have to adjust their pose. We did this in the *create_ pose* function, where we defined the goal poses starting from the waypoints the program loaded previously, as shown below in Fig 2.3.

```python
def create_pose(transform):
    pose = PoseStamped()
    pose.header.frame_id = 'map'
    pose.header.stamp = navigator.get_clock().now().to_msg()

    #Set the position after the transformation
    pose.pose.position.x = -(transform["position"]["y"]-3.5)
    pose.pose.position.y = (transform["position"]["x"]+3)
    pose.pose.position.z = transform["position"]["z"]

    #Set the quaternion after the transformation
    rpy = transform.get("orientation", {})
    roll = rpy.get("roll", 0)
    pitch = rpy.get("pitch", 0)
    yaw = rpy.get("yaw", 0)+1.57

    quaternion = rpy_to_quaternion(roll, pitch, yaw)

    pose.pose.orientation.x = quaternion['x']
    pose.pose.orientation.y = quaternion['y']
    pose.pose.orientation.z = quaternion['z']
    pose.pose.orientation.w = quaternion['w']

    return pose
```

Figure 2.3: Code snippet of *follow_ waypoints.py*.

Moreover, as done before, we selected the correct waypoint execution order, as shown in Fig 2.4.

```python
if sys.argv[1] == 'mapping':
    waypoint_order = [0, 1, 2, 3, 4, 5]
elif sys.argv[1] == 'waypoints':
    waypoint_order = [2, 3, 1, 0]
elif sys.argv[1] == 'aruco':
    waypoint_order = [0, 1]
else:
    print("Action not well-specified")
    return
```

Figure 2.4: Code snippet of *follow_ waypoints.py*.

In the end, after running this file, we obtained the following trajectory, as shown in Fig 2.5.

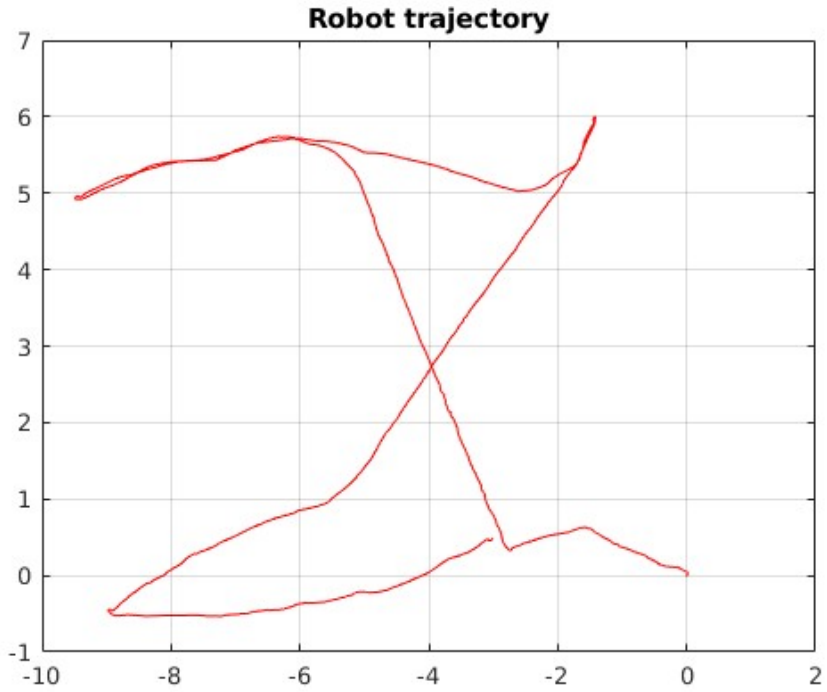


Figure 2.5: Trajectory of the robot in the XY plan.

Where the origin of this graph is the starting point of the trajectory, the upper left point is the first waypoint, the upper right point is the second one, the lower left point is the third one and the lower right point is the fourth and last one.

Chapter 3

Map the environment and tuning of parameters

The third point of the homework required us to map the environment and then repeat the mapping procedure tuning the navigation stack's parameters

3.1 Map of the environment

The first point of this part asked us to define new waypoints in order to obtain a complete map of the environment.
The waypoints we selected are the following, each of them is with respect to map coordinates.

- Goal_0: $x = 1.95m, \quad y = -4.65m, \quad Y = 0deg$
- Goal_1: $x = 4.0m, \quad y = 0.16m, \quad Y = 30$

- Goal_2: $x = 5.9m, \quad y = 4.36m, \quad Y = -90deg$
- Goal_3: $x = 9.3m, \quad y = 0.16m, \quad Y = 0deg$
- Goal_4: $x = -7.38m, \quad y = -1.91m, \quad Y = 0deg$
- Goal_5: $x = 1.75m, \quad y = 0.16m, \quad Y = 0deg$

These waypoints, stored in *mapping.yaml*, are executed in the following order: [0,1,2,3,4,5], as can be seen in Fig 2.5 in Cap 2. Launching again the *follow_ waypoints.py* script and selecting '*mapping*' as waypoints to follow, we ended up by having the map as showed in Fig 3.1.

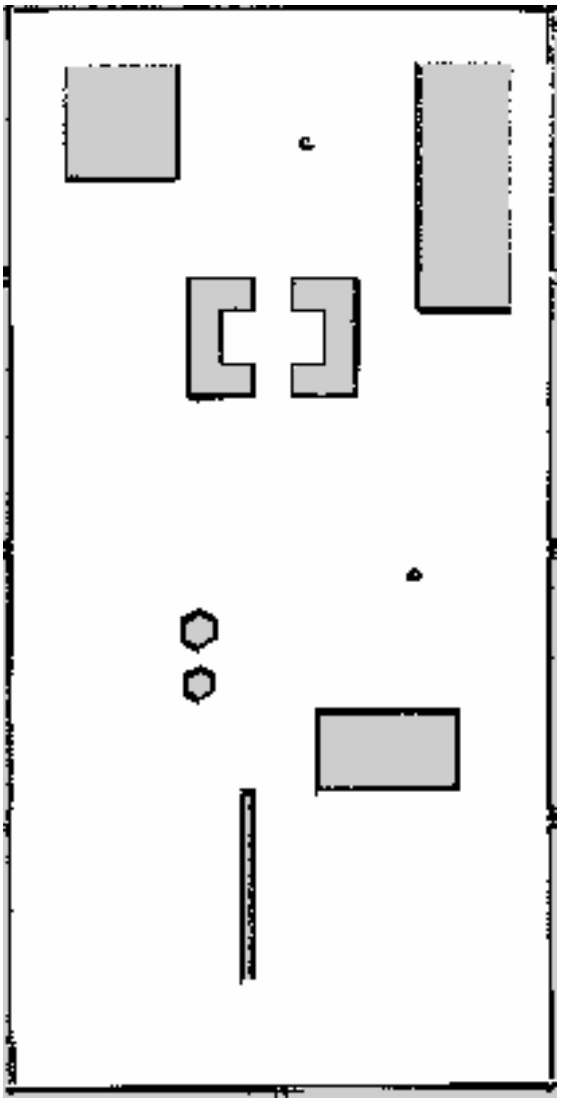


Figure 3.1: Map obtained from mapping waypoints.

3.2 Tuning of navigation stack's parameters

This section is about the tuning of the navigation stack's parameters complemented with some comments about the results. As suggested, the modified parameters are:

- minimum_travel_distance (default: 0.01)
- minimum_travel_heading (default: 0.01)
- resolution (default: 0.05)
- inflation_radius (default: 0.75)
- cost_scaling_factor (default: 3.0)

Moreover, it is right to specify that we set the desired linear velocity to 1.2 in the *controller_plugin* which is in the *explore.yaml* file. In order to appreciate the differences given by changing each parameter, we have run four different tests, each with a different configuration that gives a characteristic name to the robot.

- Scared Robot - it wants to be far from the obstacles;
- Brave Robot - it does not care too much of the obstacles;
- Doctor Robot - it does an accurate scanning of the map;
- Blind Robot - its perception ability is poor;

3.2.1 Test 1 - Scared Robot

For this test we applied the following changes:

- inflation_radius set to 0.9
- cost_scaling_factor 0.5

With this configuration we enhanced the radius of the perceived obstacle and changed the scaling factor in order to take the robot away from the obstacles. The mobile robot took 204 seconds to complete the mapping procedure and the results it produced are shown in Fig 3.2 and Fig 3.3, in terms of map and trajectory.

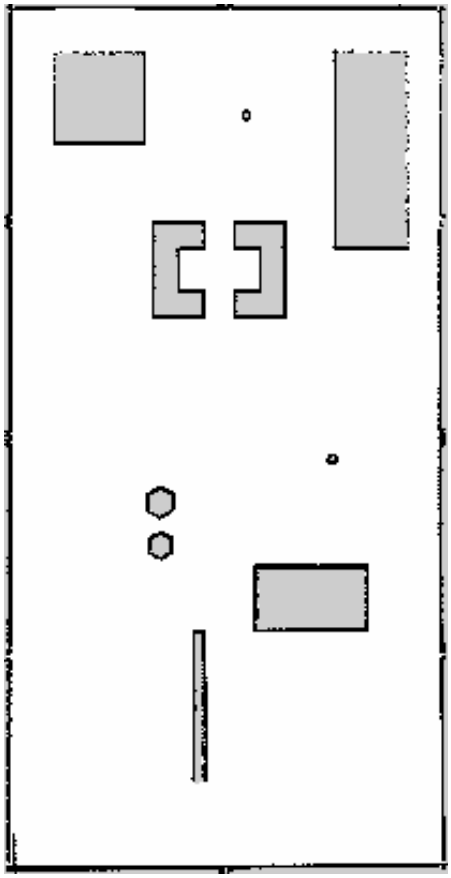


Figure 3.2: Map obtained from first test.

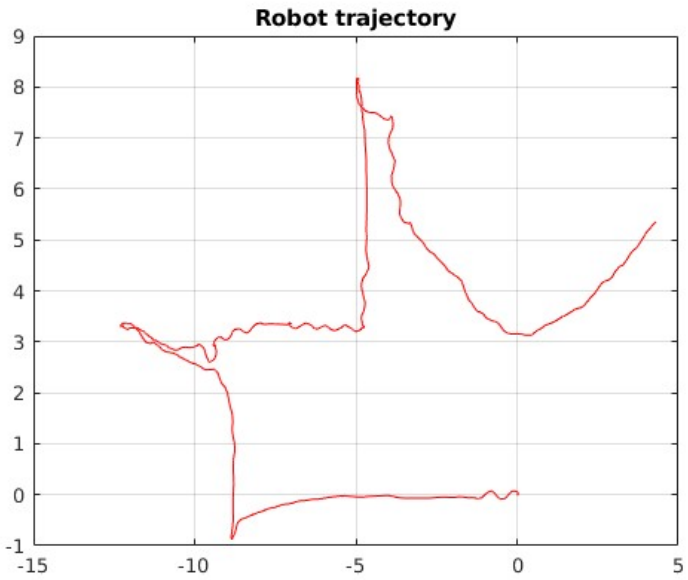


Figure 3.3: Mobile robot trajectory for first test.

As it is possible to see, this configuration made the robot more uncertain near the obstacles and took a more wider trajectory.

3.2.2 Test 2 - Brave Robot

For this test we applied the following changes:

- inflation_radius set to 0.5
- cost_scaling_factor 5.0

Using this configuration, unlike the first test, we want to reduce the radius of the perceived obstacle and changing the scaling factor in order to have a reduced cost near the obstacles. The mobile robot took 147 seconds to complete the mapping procedure and the result it produced is shown in Fig 3.4 and 3.5, in terms of map and trajectory.

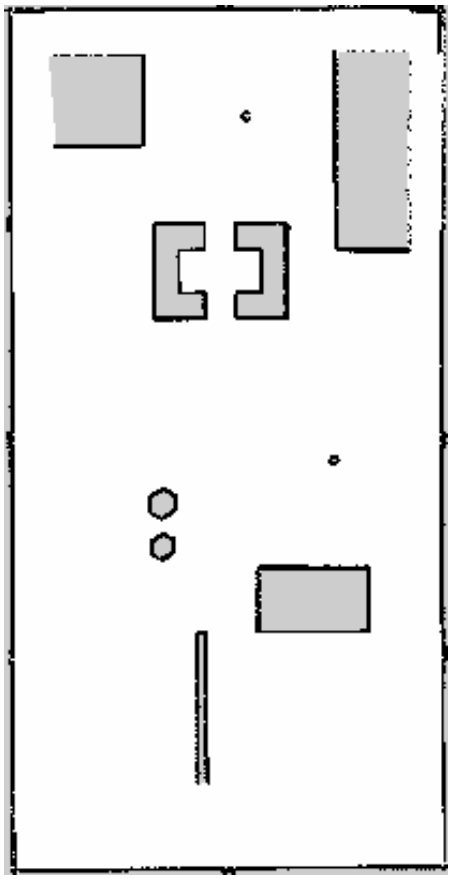


Figure 3.4: Map obtained from second test.

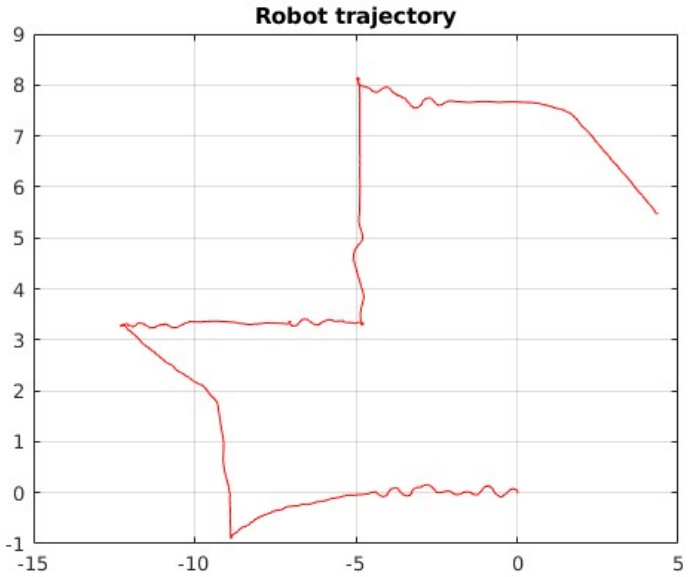


Figure 3.5: Mobile robot trajectory for second test.

Differently as shown for the first test, this configuration made the robot more confident near the obstacles, making smoother trajectories and also took less time to complete.

3.2.3 Test 3 - Doctor Robot

For this test we applied the following changes:

- resolution 0.02
- minimum_travel_distance 0.01
- minimum_travel_heading 0.01

This test, instead, aims to emphasize the robot's perceptive capabilities and how they depend on the modified parameters. We improved the resolution by decreasing its value and configured the robot to update its localization every centimeter, thus achieving a high update rate.

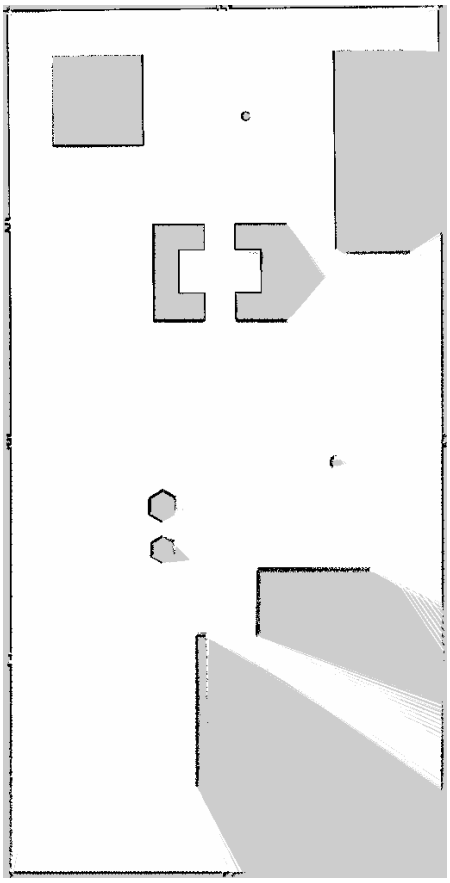


Figure 3.6: Map obtained from third test.

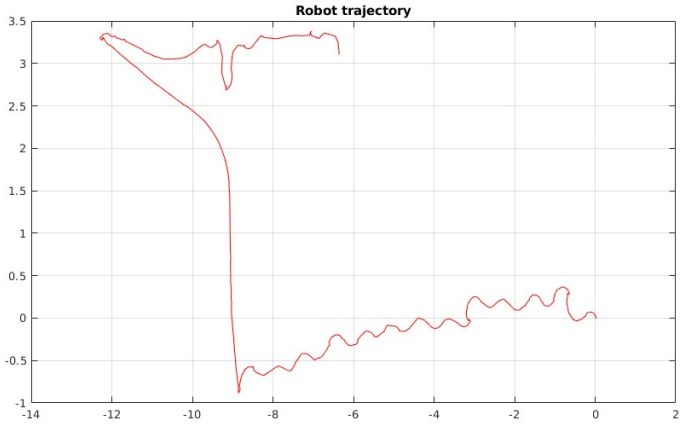


Figure 3.7: Mobile robot trajectory for third test.

As it is possible to see in Fig 3.6 and Fig 3.7 either from the map either from the trajectory, the robot failed its trajectory from the fourth

waypoint, aborting its mission and it took 183 seconds to complete the first three waypoints. On the other hand, the map is way more precise.

3.2.4 Test 4 - Blind Robot

For this test we applied the following changes:

- resolution 0.07
- minimum_travel_distance 0.02
- minimum_travel_heading 0.02

Differently from the previous test, we reduced the resolution by increasing its value and slowed down the localization update rate.

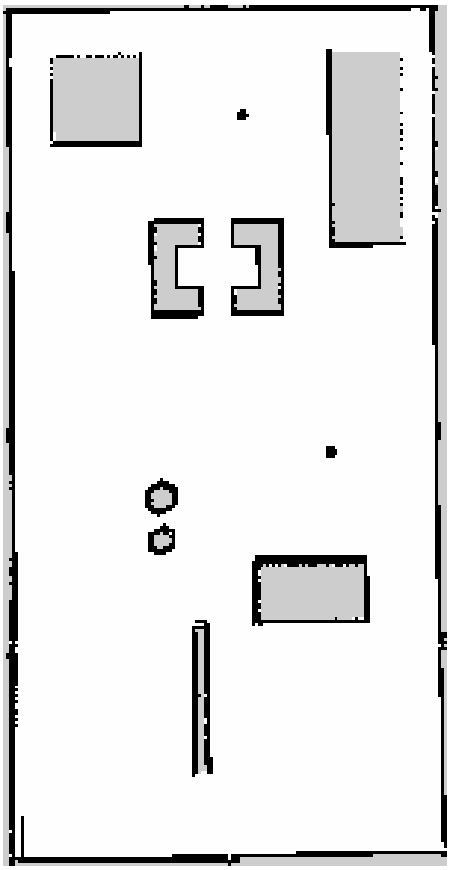


Figure 3.8: Map obtained from third test.

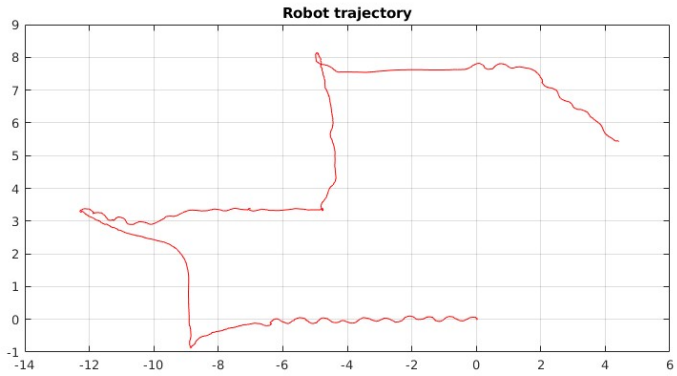


Figure 3.9: Mobile robot trajectory for third test.

In this case, the robot complete successfully the mapping procedure in 148 seconds, being much faster than the previous test. However, as it could be deducible, the map is not very well detailed.

Chapter 4

Vision-based navigation task

The fourth and last part of the homework required us to make the robot perform a vision-based navigation task following these guidelines:

- Send the robot in proximity of the obstacle 9, where it has been previously placed the ArUco tag;
- Make the robot look for the marker with the camera and then, once detected, publish its pose with respect to the map frame;
- In the end, the mobile robot has to return to its initial position.

In order to complete this task, we first created a new launch file that runs the ArUco's *single.launch.py* with the parameters of the marker number 115, the *fra2mo_ explore.launch.py* launch file and the node we

created for publishing the static transform of the marker with respect to the map frame, as you can see in Fig 4.1.

```python
def generate_launch_description():
    # Percorsi dei file
    explore_path = os.path.join(
        get_package_share_directory('rl_fra2mo_description'),
        'launch',
        'fra2mo_explore.launch.py'
    )

    aruco_path = os.path.join(
        get_package_share_directory('aruco_ros'),
        'launch',
        'single.launch.py'
    )

    # Argomenti di configurazione
    use_sim_time = LaunchConfiguration('use_sim_time', default='true')

    # Include del nodo esplorazione
    explore_node = IncludeLaunchDescription(
        PythonLaunchDescriptionSource(explore_path)
    )

    # Include del nodo ArUco
    aruco_node = IncludeLaunchDescription(
        PythonLaunchDescriptionSource(aruco_path),
        launch_arguments={"marker_id": "115", "marker_size": "0.1"}.items(),
    )

    # Nodo per la trasformazione ArUco
    aruco_tf = Node(
        package='rl_fra2mo_description',
        executable='aruco_tf_publisher',
        name='aruco_tf',
        output='screen',
        parameters=[{'use_sim_time': use_sim_time}]
    )

    # Ritorna la descrizione del lancio
    return LaunchDescription([
        DeclareLaunchArgument('use_sim_time', default_value='true', description='Use simulation clock'),
        explore_node,
        aruco_node,
        aruco_tf,
    ])
```

Figure 4.1: Code snippet of *aruco_fra2mo.launch.py*.

In particular, now we are going to better discuss this last mentioned node. While focusing on the second point of the guidelines, in this node we created a callback for the pose of the marker. In this function, firstly we retrieved the marker's pose and then transformed it from the camera link frame to the map frame, the pose with respect to the map frame is published on the topic */aruco_pose_in_map*. It is possible to appreciate the code in the Fig 4.2.

```cpp
void pose_callback(const geometry_msgs::msg::PoseStamped::SharedPtr pose_msg)
{
    try
    {
        // Trasforma la posa da camera_link a map
        geometry_msgs::msg::TransformStamped transform_stamped =
            tf_buffer_.lookupTransform("map", pose_msg->header.frame_id, rclcpp::Time(0), rclcpp::Duration::from_seconds(1.0));

        geometry_msgs::msg::PoseStamped pose_in_map;
        tf2::doTransform(*pose_msg, pose_in_map, transform_stamped);

        RCLCPP_INFO(this->get_logger(), "ArUco Pose in map frame: x: %.2f, y: %.2f, z: %.2f",
                    pose_in_map.pose.position.x,
                    pose_in_map.pose.position.y,
                    pose_in_map.pose.position.z);

        // Pubblica la posa trasformata
        pose_in_map.header.frame_id = "map";
        pose_in_map.header.stamp = this->now();
        publisher_->publish(pose_in_map);

        // Pubblica una TF statica se non è già stata pubblicata
        if (!aruco_static_tf_published_)
        {
            publish_static_transform(pose_in_map);
            aruco_static_tf_published_ = true;
            RCLCPP_INFO(this->get_logger(), "Static TF for ArUco marker published.");
        }
    }
    catch (const tf2::TransformException &ex)
    {
        RCLCPP_ERROR(this->get_logger(), "Could not transform pose: %s", ex.what());
    }
}
```

Figure 4.2: Code snippet of *aruco_tf_publisher.cpp*.

Furthermore, we created a new function that publishes the static transform of the marker, as it is possible to see in Fig 4.3.

```cpp
void publish_static_transform(const geometry_msgs::msg::PoseStamped &pose)
{
    geometry_msgs::msg::TransformStamped static_transform_stamped;

    static_transform_stamped.header.stamp = this->now();
    static_transform_stamped.header.frame_id = "map";  // Riferimento fisso
    static_transform_stamped.child_frame_id = "aruco_marker_static";  // Nome del frame statico

    // Copia posizione
    static_transform_stamped.transform.translation.x = pose.pose.position.x;
    static_transform_stamped.transform.translation.y = pose.pose.position.y;
    static_transform_stamped.transform.translation.z = pose.pose.position.z;

    // Copia orientamento
    static_transform_stamped.transform.rotation = pose.pose.orientation;

    // Pubblica la trasformazione statica
    static_broadcaster_->sendTransform(static_transform_stamped);
}
```

Figure 4.3: Code snippet of *aruco_tf_publisher.cpp*.

In the end, launching the previous mentioned launch file along with *follow_waypoints.py* selecting '*aruco*' as waypoints to follow, we are able to send the robot near the obstacle 9, where the ArUco marker has been placed, and once it has recognized it, it goes back to the

original position. This has been accomplished by adding some new features to the *follow_waypoints.py* file. In particular, we added an aruco callback (Fig 4.4) that tells the node if the marker has been detected. If the marker has been detected (FIg 4.5), the node can go over and make the robot return to its spawn point.

```
if sys.argv[1] == 'aruco':
        node.aruco_detected = False  # Flag per indicare se il marker è stato rilevato
        def aruco_pose_callback(msg):
            # Callback per gestire la pose dell'ArUco
            if not node.aruco_detected:
                node.get_logger().info("ArUco marker detected!")
                node.aruco_detected = True

        aruco_pose_sub = node.create_subscription(
            PoseStamped,
            '/aruco_single/pose',
            aruco_pose_callback,
            10
        )
```

Figure 4.4: Code snippet of *follow_waypoints.py*.

```
if result == TaskResult.SUCCEEDED:
    print(f'Goal {waypoint_order[i] + 1} succeeded!')

    # Se l'azione è "aruco", aspetta il rilevamento del marker
    if sys.argv[1] == 'aruco' and waypoint_order[i] == 0:
        node.aruco_detected = False
        while not node.aruco_detected:
            rclpy.spin_once(node)

        node.get_logger().warn("ARUCO")
```

Figure 4.5: Code snippet of *follow_waypoints.py*.

Moreover, as soon as the robot sees the marker, it publishes the static transform of the marker, making it visible even if the robot does not look anymore at the target.

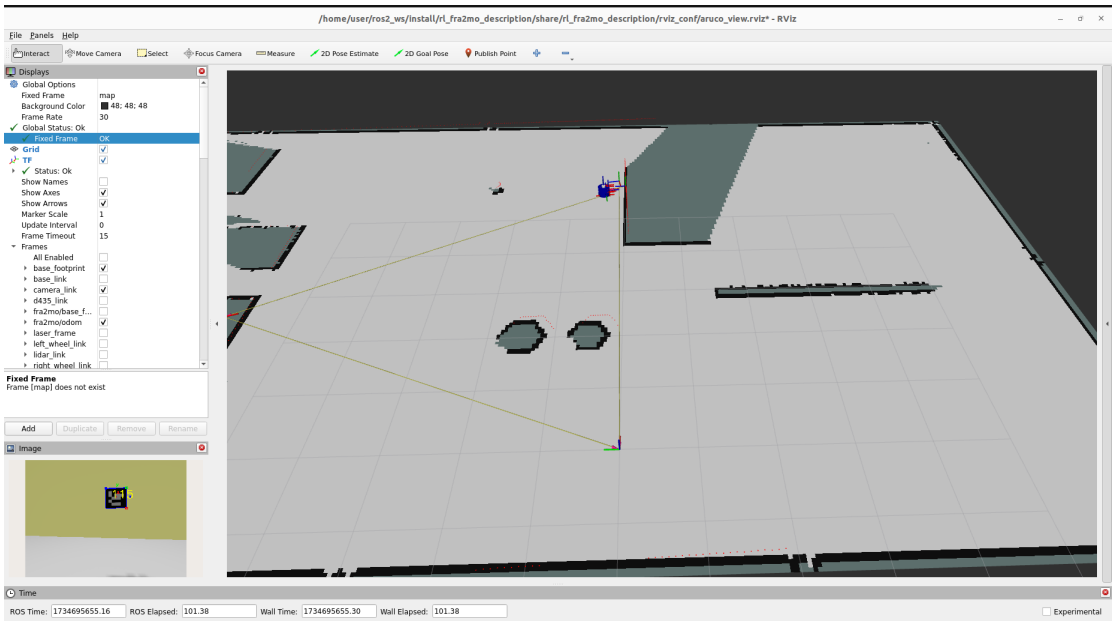


Figure 4.6: Robot after completing the first waypoint and detecttion of the marker.

In Fig 4.6 is shown the robot near the obstacle 9. In the lower left of the figure it is possible to see the ArUco detection and in the center the transformation of the marker with respect to the map frame. Additionally, in Fig 4.7 is shown the robot returned in its initial pose with the marker transform still being published as static.

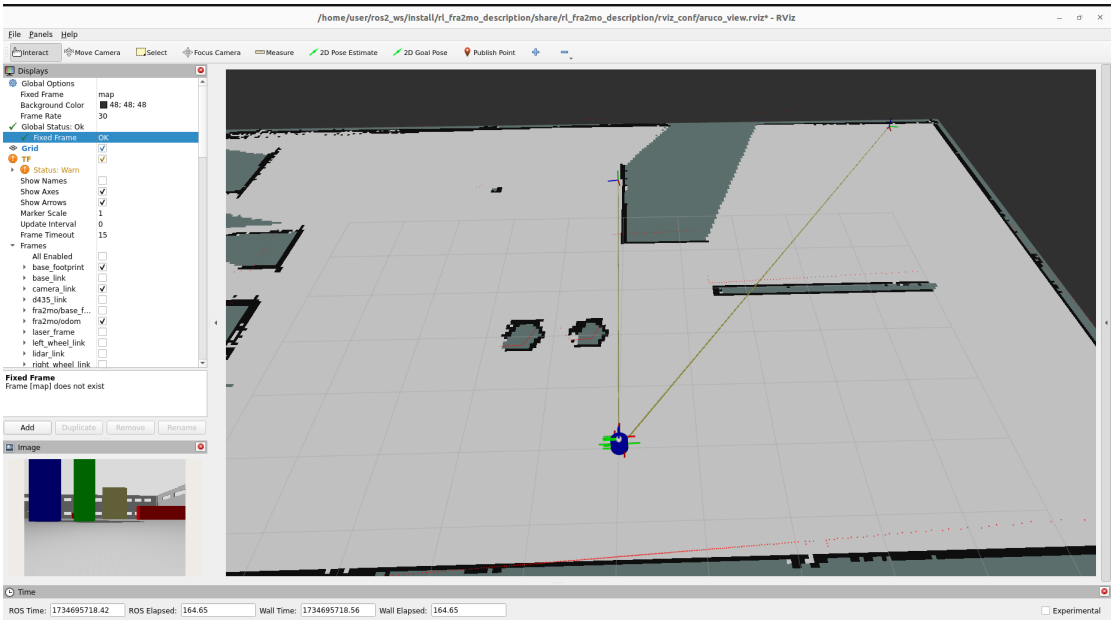


Figure 4.7: Robot after turning back to its initial position.