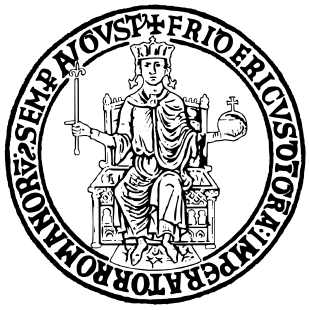
Università degli Studi di Napoli Federico II



Scuola Politecnica e delle Scienze di Base

Dipartimento di Ingegneria Elettrica e Tecnologie dell'Informazione

Robotics Lab

Homework 4: Control a mobile robot to follow a trajectory

Components
William Notaro
Chiara Panagrosso
Salvatore Piccolo
Roberto Rocco

Anno Accademico 2024–2025

Contents

The aim of the project is to develop an autonomous navigation software framework to control a mobile robot. The project involves creating a simulation environment in Gazebo, implementing trajectory-following tasks using the Nav2 framework, mapping the environment by tuning navigation parameters via SLAM algorithm, and enabling vision-based navigation through the use of ArUco markers and the robot's camera system. The goal is to demonstrate the robot's ability to navigate autonomously, detect specific visual markers, and interact effectively with its surroundings.

It is possible to view the video of the comparison of the configurations used in the third chapter on YouTube at the following link:

- Video comparison: `https://youtu.be/ieOGv2oU5Bo`

Chapter 1

World settings

1.1 Spawning fra2mo and obstacle

To begin, the project required spawning the robot at the assigned position with a specific orientation. Below is a code snippet that demonstrates how this task was accomplished:

```
position = [-3.0, 3.5, 0.100]
yaw = -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(yaw)]
)
```

Listing 1.1: `gazebo_fra2mo.launch.py`

Secondly, an obstacle called "obstacle 9" was placed in the following pose: `x = -3 m, y = -3.3 m, z = 0.1 m, Y = 90 deg` by appropriately modifying the world file, as shown below.

```
<include>
  <name>obstacle_09_wArUco</name>
  <pose> -3 -3.3 0.1 0 0 1.57</pose>
  <uri>model://obstacle_09_wArUco</uri>
</include>
```

Listing 1.2: `leonardo_race_field.sdf`

1.2 Aruco marker and Camera

On the obstacle 9 an ArUco 115 tag was placed by editing the model `.sdf` of the obstacle_09 by adding the following code thus putting the ArUco marker `.png` on a link ArUco attached to a joint whose parent link is the link of the obstacle considered:

```
<joint name="joint" type="fixed">
  <parent>link_obstacle</parent>
  <child>link_aruco</child>
</joint>

<link name="link_aruco">
  <pose>-0.001 0.05 0.22 -1.57 3.14 0</pose>
  <visual name='aruco_marker'>
    <geometry>
      <plane>
        <normal>1 0 0</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>
          <!--<uri>~/ros2_ws/src/rl_fra2mo_description/models/arucotag/
    aruco-201.png</uri> -->
        </metal>
      </pbr>
    </material>
  </visual>
</link>
```

Listing 1.3: `obstacle_09.sdf, in obstacle_09_wArUco folder`

Finally a camera was added on top of the fra2mo robot: in the `URDF` file of the fra2mo robot is called a macro, that is attached to a an empty link called `camera_link` which implementation is shown below.

```
<xacro:macro name="camera_gazebo_sensor" params="parent">

  <link name="tool_camera">
    <inertial>
      <origin xyz="0 0 0"/>
      <mass value="0.1"/>
      <inertia ixx="0.0001" ixy="0" ixz="0" iyy="0.0001" iyz="0" izz="
    0.0001"/>
    </inertial>
    <visual>
      <geometry>
        <box size="0.022 0.022 0.055"/>
      </geometry>
      <origin xyz="0 0 0" rpy="0 0 0"/>
      <material name="green"/>
    </visual>
  </link>

  <joint name="joint_camera" type="fixed">
    <origin xyz="0.0 0.0 0.0" rpy="0 0 0"/>
```

```
    <parent link="${parent}" />
    <child link="tool_camera"/>
  </joint>

  <gazebo reference="tool_camera">
    <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>30.0</update_rate>
      <visualize>true</visualize>
      <topic>camera</topic>
    </sensor>
    <plugin filename="gz-sim-sensors-system" name="
   gz::sim::systems::Sensors">
        <render_engine>ogre2</render_engine>
    </plugin>
</gazebo>

</xacro:macro>
```

Listing 1.4: camera_gazebo_macro.xacro

Chapter 2

Navigation towards desired goals

The aim of the second point of the project is to enable autonomous navigation for the mobile robot using the `Nav2 Simple Commander API`. This involves defining a sequence of navigation goals in a YAML file, implementing a waypoint-following behavior that progresses through the specified goals in a defined order, and recording the executed trajectory for analysis. The goal is to showcase the robot's ability to autonomously navigate through a predefined path in the simulation environment while adhering to specified waypoints.

2.1 Map creation

To enable autonomous navigation, the first step was to create a detailed map of the `world` in which the robot operates. The exploration was initiated from the center of the Gazebo map, which will henceforth be considered as the coordinates `[0, 0, 0]` (corresponding to `[x, y, yaw]`). These coordinates also coincide with the position of the `map` frame. Every subsequent position will thus be referenced relative to this specifically defined frame.

The exploration process, executed using `fra2mo_explore.launch.py`, resulted in the map shown in Fig. 2.1. Such exploration made use of the `explore_lite_launch` file, which included an algorithm for generating particular points to explore in such a way to obtain a complete and detailed mapping of the `world`.

Finally, the map was converted into a serialized format using `slam_toolbox`, which enables saving the pose-graph and metadata of the map for reloading at a later time, either with the same robot or a different one, to continue mapping the environment. This process utilizes serialization and deserialization techniques to store and reload map data, allowing for the loading of existing maps, setting the robot's pose within the map (not necessarily at the center), and performing localization or updating the map as needed. The creation of the serialized map was facilitated by the `SlamToolboxPlugin`, an `Rviz` plugin designed for this purpose, which GUI is shown in Fir. 2.2.

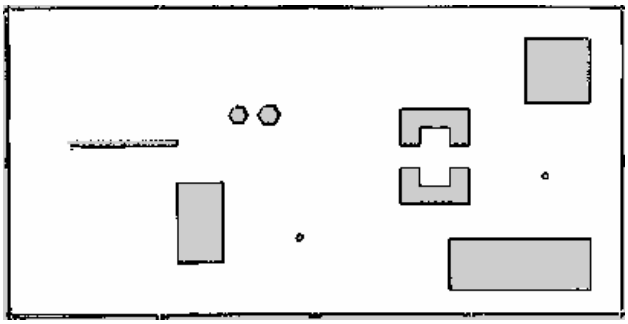


Figure 2.1: Map obtained with exploration

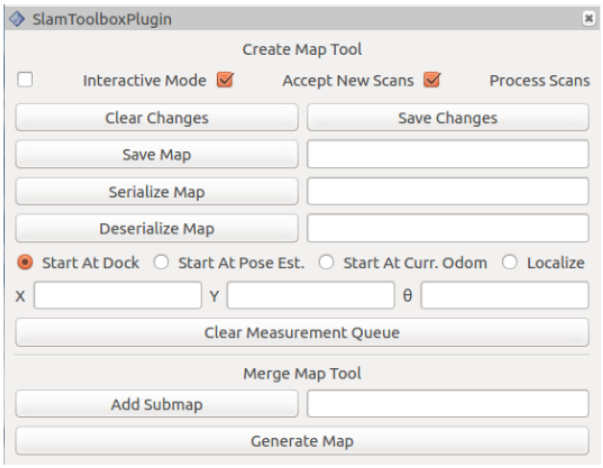


Figure 2.2: SlamToolboxPlugin GUI

2.1.1 Goal poses

The next step is to import the four goals assigned by the homework into an appropriate file, named `waypoints.yaml`. The goal points are:

- **Goal 1:** $x = 0\,\text{m}$, $y = 3\,\text{m}$, $Y = 0°$
- **Goal 2:** $x = 6\,\text{m}$, $y = 4\,\text{m}$, $Y = 30°$
- **Goal 3:** $x = 7.0\,\text{m}$, $y = -1.4\,\text{m}$, $Y = 180°$
- **Goal 4:** $x = -1.6\,\text{m}$, $y = -2.5\,\text{m}$, $Y = 75°$

It has been assumed that the coordinates of these waypoints are expressed with respect to the `map` frame, positioned at the center of the map as explained previously. In Fig. 2.3 the position of these waypoints in the map is shown.

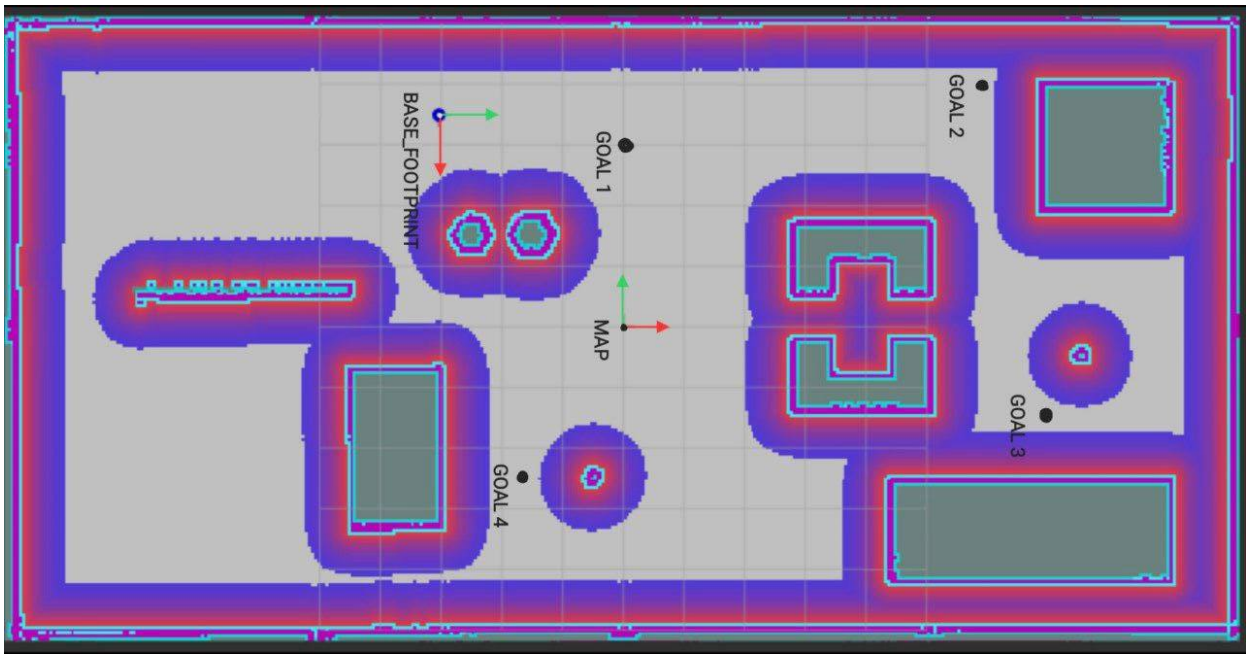


Figure 2.3: Position of the goal waypoints in the map

2.2 Autonomous Waypoint Traversal

The order of the explored goals must be Goal 3 → Goal 4 → Goal 2 → Goal 1. In order to get this kind of behaviour we had to specifiy inside our config file `waypoint.yaml` a name for each goal point. This enable us to create a dictionary of goal poses, each of team is acceded using as key their name, that is 'Goal_X'. In order to give goal poses in a preferred order we used a string vector `waypoint_order` that is used to acced to the waypoints dictionary to create the goal poses following the desired order. Once we've created the goal poses, we wait for `navigator` to start-up, and sent to it our goal poses via `followWaypoints()` function

```
def main():
    rclpy.init()

    global navigator
    navigator = BasicNavigator()
    # Load waypoints from the YAML file
    yaml_file_name = 'waypoints.yaml'
    yaml_file = os.path.join(get_package_share_directory('
   rl_fra2mo_description'), "config",yaml_file_name)
    # Load waypoints from the YAML file
    waypoints = load_waypoints(yaml_file)
    # Define the desired order of waypoint execution
    waypoint_order = ['Goal_3', 'Goal_4', 'Goal_2', 'Goal_1']
    # Create the goal poses in the specified order
    goal_poses = [create_pose(waypoints[name]) for name in waypoint_order]
    print(goal_poses)
    # 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)
```

```
24
25      i = 0
26      while not navigator.isTaskComplete():
27          i = i + 1
28          feedback = navigator.getFeedback()
29
30          if feedback and i % 5 == 0:
31              print('Executing current waypoint: ' +
32                    str(feedback.current_waypoint + 1) + '/' + str(len(
    goal_poses)))
33              now = navigator.get_clock().now()
34
35              # Some navigation timeout to demo cancellation
36              if now - nav_start > Duration(seconds=600):
37                  navigator.cancelTask()
38                  print('Cancelled current waypoint')
39
40      result = navigator.getResult()
41      if result == TaskResult.SUCCEEDED:
42          print('Goal succeeded!')
43      elif result == TaskResult.CANCELED:
44          print('Goal was canceled!')
45      elif result == TaskResult.FAILED:
46          print('Goal failed!')
47      else:
48          print('Goal has an invalid return status!')
49      exit(0)
```

Listing 2.1: `follow_waypoints.py`

2.3 Robot's performed trajectory in the XY-plane

The analysis and visualization of the robot's performed path, as planned by the global planner in Nav2, began with the establishment of a dynamic transform publisher between the `map` frame and the `base_footprint` frame, which was essential for providing the real-time spatial relationship between the robot's base and the map frame. The dynamic transform publisher has been implemented as a node in the `fra2mo_pos_publisher.cpp`.

Using `ros2bag`, we recorded two key topics: `/fra2mo_position` and `/plan`.
The `/fra2mo_position` topic provided the robot's actual positions over time, computed relative to the `map` frame. Meanwhile, the `/plan` topic captured the trajectory computed by the global planner, which represented the robot's intended path through the environment. These recordings encapsulated both the true motion of the robot and the planner's output, forming the basis for comparison and performance assessment.

Subsequently, the recorded data were processed using a MATLAB script. This script read the bagfiles, extracted the positional data, and plotted the robot's executed trajectory. To enhance the visualization, the script overlaid the trajectory on a background map, as shown in Fig. 2.4, providing a spatially contextual representation of the robot's navigation within the environment. The resulting plot not only has enabled a detailed examination of how closely the robot followed the planned path, but has also allowed us to notice how the Nav2 global planner can change its desired path over time.

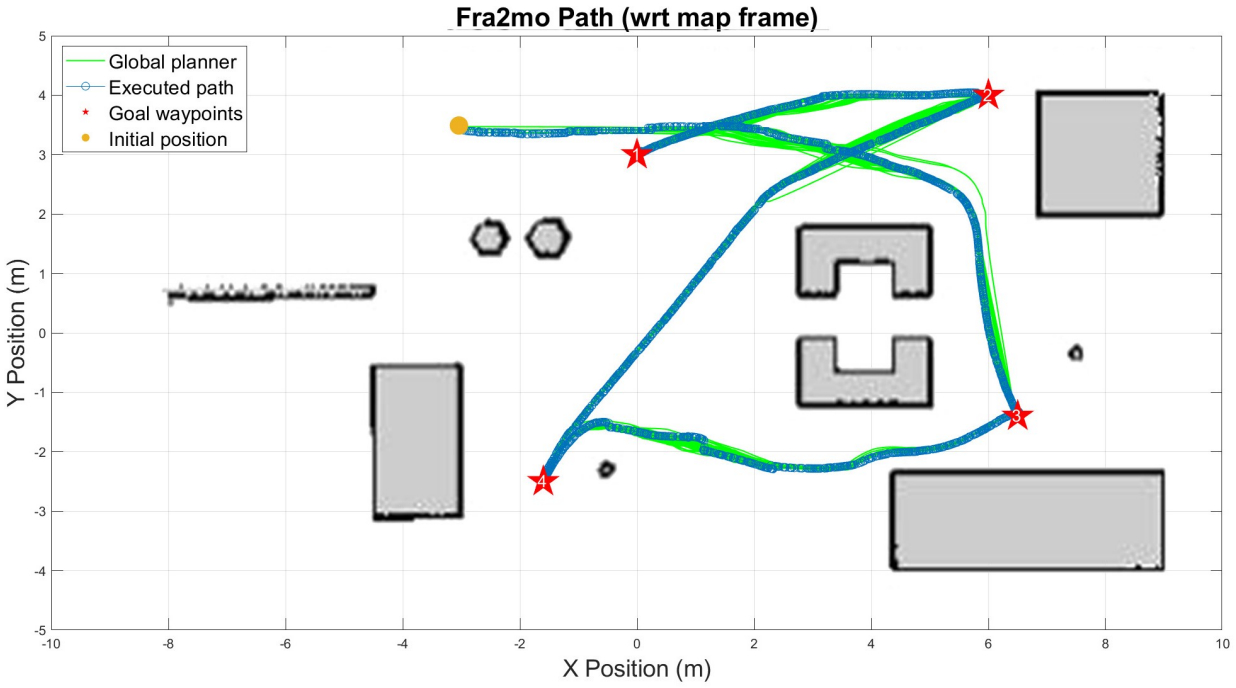


Figure 2.4: Executed robot trajectory in the XY-plane

Chapter 3

Mapping of the environment

In this chapter we will get a complete map of the environment by using different goal poses with respect to the starting pose of fra2mo.

3.1 Points of interest

In order to map the entire environment some goal poses were chosen with respect to the base_footprint frame of our mobile robot. Initially, for testing purposes, 15 goal points have been designed. However, with the default LIDAR specifications, it turned out that only 6 of them were sufficient to map the entire environment.

```
waypoints:
  - name: Goal_2
    position:
      x: 4.485
      y: -5.46
      z: 0.0
    orientation:
      w: 0.933
      x: 0.0
      y: 0.0
      z: 0.359
  - name: Goal_3
    position:
      x: 7.7443
      y: -1.2044
      z: 0.0
    orientation:
      w: 0.6966
      x: 0.0
      y: 0.0
      z: 0.7175
  - name: Goal_9
    position:
      x: -1.0747
      y: 10.139
      z: 0.0
    orientation:
      w: 0.70716
      x: 0.0
      y: 0.0
      z: -0.70705
  - name: Goal_10
```

```
    position:
      x: 0.64859
      y: 6.7699
      z: 0.0
    orientation:
      w: 0.82454
      x: 0.0
      y: 0.0
      z: 0.5658
  - name: Goal_12
    position:
      x: 6.7059
      y: 12.491
      z: 0.0
    orientation:
      w: 1.0
      x: 0.0
      y: 0.0
      z: 0.0
  - name: Goal_13
    position:
      x: 8.2134
      y: 7.258
      z: 0.0
    orientation:
      w: 0.65202
      x: 0.0
      y: 0.0
      z: -0.7581
```

Listing 3.1: An extract of `mapping.yaml`

Of this points only a few essential ones were used to map the environment.

The python script that sends commands to the mobile robot to make it reach the goal poses is the following

```
#! /usr/bin/env python3
import os
from ament_index_python.packages import get_package_share_directory
from geometry_msgs.msg import PoseStamped
from nav2_simple_commander.robot_navigator import BasicNavigator,
    TaskResult
import rclpy
from rclpy.duration import Duration
import yaml

def load_waypoints(file_path):
    """Load waypoints from a YAML file into a dictionary."""
    with open(file_path, 'r') as f:
        data = yaml.safe_load(f)

    waypoints = {}
    for waypoint in data['waypoints']:
        name = waypoint['name']
        waypoints[name] = waypoint
    return waypoints

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

def main():
    rclpy.init()

    global navigator
    navigator = BasicNavigator()

    # Load waypoints from the YAML file

    yaml_file_name = 'mapping.yaml'
    yaml_file = os.path.join(get_package_share_directory('
   rl_fra2mo_description'), "config",yaml_file_name)

    # Load waypoints from the YAML file
    waypoints = load_waypoints(yaml_file)

    # Define the desired order of waypoint execution
    waypoint_order = ['Goal_10','Goal_9','Goal_12','Goal_13','Goal_3','
   Goal_2']


    # Create the goal poses in the specified order
    goal_poses = [create_pose(waypoints[name]) for name in waypoint_order]

    print(goal_poses)


    # 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
```

```
80              if now - nav_start > Duration(seconds=600):
81                  navigator.cancelTask()
82
83      # Do something depending on the return code
84      result = navigator.getResult()
85      if result == TaskResult.SUCCEEDED:
86          print('Goal succeeded!')
87      elif result == TaskResult.CANCELED:
88          print('Goal was canceled!')
89      elif result == TaskResult.FAILED:
90          print('Goal failed!')
91      else:
92          print('Goal has an invalid return status!')
93
94      # navigator.lifecycleShutdown()
95
96      exit(0)
97
98
99  if __name__ == '__main__':
100     main()
```

Listing 3.2: `map_everything.py`

3.2 Behavior with Parameter Variations

This subsection documents the results of tuning parameters for SLAM and exploration tasks in a simulated environment. The study explores the impact of different configurations on robot performance, specifically focusing on trajectory behavior, execution timing, and map accuracy.

Parameter Definitions

The following parameters were modified to analyze their impact on SLAM and exploration:

SLAM Parameters

- **minimum_travel_distance**: The minimum distance the robot must travel before updating its position in the map. Lower values increase mapping detail but incur higher computation costs.
- **minimum_travel_heading2**: The minimum change in the robot's heading required to trigger a map update. Smaller values result in more precise updates.
- **resolution**: Defines the size of each grid cell in the costmap. Lower values provide higher detail but require greater computational resources.
- **transform_publish_period**: The time interval at which transforms are published. Lower values enhance real-time responsiveness.

Exploration Parameters

- **inflation_radius**: The radius around obstacles where navigation costs are inflated to discourage close proximity.

- **cost_scaling_factor**: Determines how rapidly the navigation cost increases near obstacles. Higher values lead to more cautious behavior.

Configurations and Expectations

Four configurations were tested to analyze performance under different conditions:

Configuration 1: High Precision

- **Execution Time: 4.16s**
- **SLAM Parameters:**
  - `minimum_travel_distance = 0.01 m`
  - `minimum_travel_heading = 0.005 rad`
  - `resolution = 0.05 m`
  - `transform_publish_period = 0.02 s`
- **Exploration Parameters:**
  - `inflation_radius = 0.2 m`
  - `cost_scaling_factor = 3.0`
- **Results:** Detailed mapping, cautious navigation, slower execution times.

Configuration 2: Fast Exploration

- **Execution Time: 2.20s**
- **SLAM Parameters:**
  - `minimum_travel_distance = 0.05 m`
  - `minimum_travel_heading = 0.01 rad`
  - `resolution = 0.1 m`
  - `transform_publish_period = 0.1 s`
- **Exploration Parameters:**
  - `inflation_radius = 0.1 m`
  - `cost_scaling_factor = 2.0`
- **Results:** Fast exploration with reduced map detail and precision.

Configuration 3:

- **Execution Time: Failed**
- **SLAM Parameters:**
  - `minimum_travel_distance = 0.3 m`
  - `minimum_travel_heading = 0.1 rad`
  - `resolution (local) = 0.2 m`
  - `resolution (global) = 0.05 m`
  - `transform_publish_period = 0.2s`
- **Exploration Parameters:**
  - `inflation_radius = 0.05 m`
  - `cost_scaling_factor = 1.5`
- **Results: Incomplete Map Generation**

Configuration 4:

- **Execution Time: Failed**
- **SLAM Parameters:**
  - `minimum_travel_distance = 0.2 m`
  - `minimum_travel_heading = 0.2 rad`
  - `resolution = 0.2 m`
  - `transform_publish_period = 0.2 s`
- **Exploration Parameters:**
  - `inflation_radius = 0.08 m`
  - `cost_scaling_factor = 1.8`
- **Results: Incomplete Map Generation**

Observations

Both Configuration 3 and 4 failed to generate a complete map of the environment due to the fact that during the exploration robot's controller is unable to find a clear path due to an obstacle, and after waiting too long, it exceeds its "patience" limit. This could be because `minimum_travel_distance` coulb be set too high: this param ensure that the robot moves a certain distance before considering a change in its path. Another reason could be the setting higher `transform_publish_period` parameter to control the frequency robot's position and map data are updated, allowing for more responsive obstacle detection. and path adjustments.

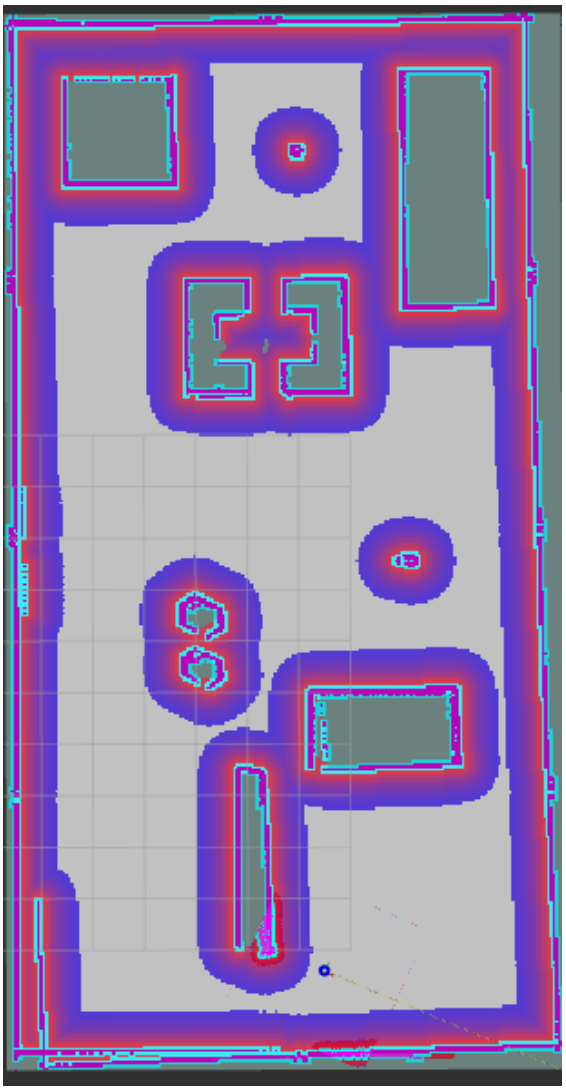


Figure 3.1: High Precision (Configuration 1)

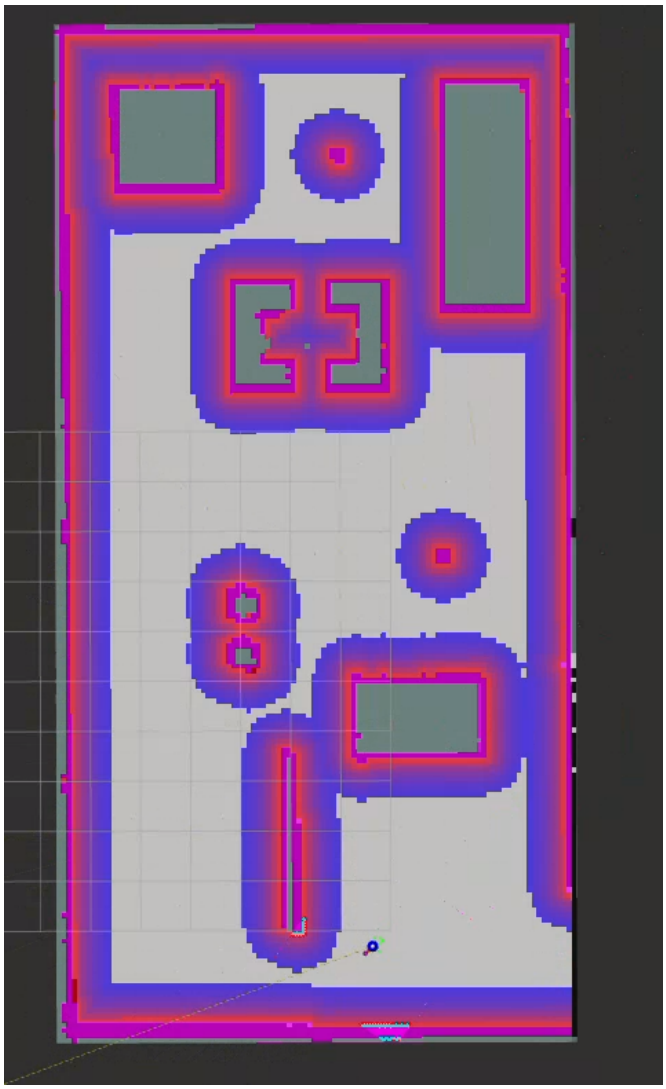


Figure 3.2: Fast Exploration (Configuration 2)

General Observations We have an obvious trade-offs between speed of the exploration and accuracy of the map: high-precision configuration resulted in superior map quality but slower navigation, while fast exploration prioritized speed at the expense of detail. Both Configuration 3 and 4 failed to generate a complete map of the environment due to the fact that during the exploration robot's controller is unable to find a clear path due to an obstacle, and after waiting too long, it exceeds its "patience" limit. This could be because `minimum_travel_distance` coulb be set too high: this param ensure that the robot moves a certain distance before considering a change in its path. Another reason could be the setting higher `transform_publish_period` parameter to control the frequency robot's position and map data are updated, allowing for more responsive obstacle detection. and path adjustments.

Chapter 4

Vision-based navigation

In this chapter we will merge vision detection tasks with autonomous navigation tasks, using the package `ros2_vision`, ad nin particular the `aruco_ros` package already implemented in the previous homework. The aim is very simple: the robot is sent to a position nearby an ArUco Marker, and oriented in such way that the marker is seen by the camera on the robot; once the marker is detected, we retrieve its position and then send it back to its initial position.

4.1 2D Navigation Vision Based

To start the vision-based navigation, the `fra2mo_run.launch.py` launch file, implemented inside the `rl_fra2mo_description` package, was modified by adding the condition `aruco_retrieve`: setting it to true file starts running both the Nav2 `navigation_launch` launch file and the `simple_single` node. This additional node, from the `aruco_ros` package, is responsible for detecting the ArUco Marker, and publishes its position with respect to a specified frame; in our case we set as default for the reference frame our `map` frame. Inside the launch file will also be launched another node, called `ArUco_Tf` whose functionality will be explained in the following section. Since we specify our SLAM launch with our configuration file `slam.yaml`, where we put full path for the serialized map, this means that we load the pre-existing map and in this way the map frame matches which the center of the environment, so we're able to check if the ArUco position we retrieve is equal to the position of its model inside the Gazebo world. Once we set up our robot, we need to give it some goal poses in order to send it nearby the marker, and with the right orientation to make it look directly at it: while it reaches the Obstacle 9, where the ArUco is attached, the `simple_single` node detects its position and we retrieve it from the `/aruco_single/pose` topic where its being published. The pose we sent to the robot to execute this task are fund inside the config `ArucoGoals.yaml`, that is passed to the `ArucoRetrieve.py`: this script gets the points to reach from the config file and sends them as goal poses via the command `followWaypoints()`. It also create a Node that subscribes to the `/aruco_single/pose` topic, to let us check if the position of the ArUco is actually being correctly detected.

```
#!/usr/bin/env python3
import os
import yaml
from ament_index_python.packages import get_package_share_directory
from geometry_msgs.msg import PoseStamped
from nav2_simple_commander.robot_navigator import BasicNavigator,
    TaskResult
import rclpy
```

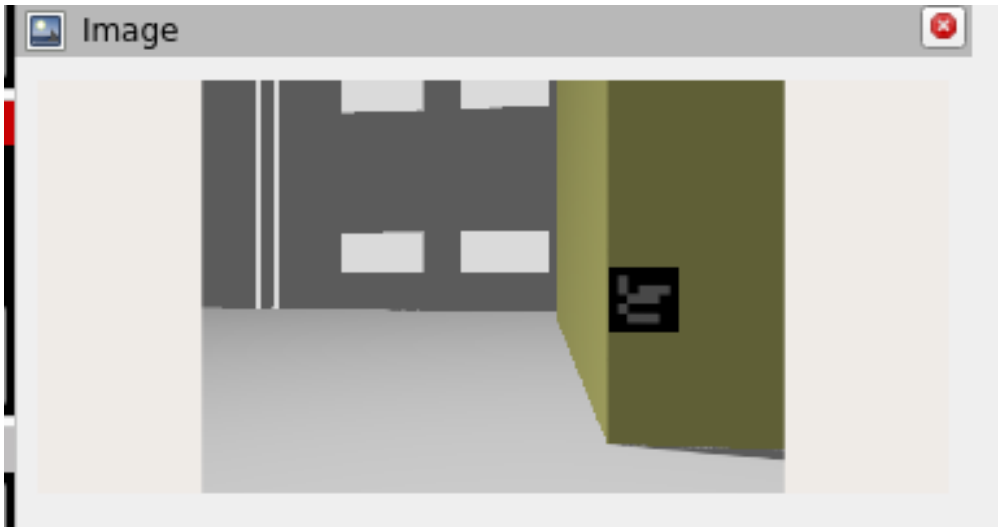


Figure 4.1: ArUco Marker seen by fra2mo

```
from rclpy.duration import Duration
from rclpy.node import Node


def load_waypoints(file_path):
    """Load waypoints from a YAML file into a dictionary."""
    with open(file_path, 'r') as f:
        data = yaml.safe_load(f)

    waypoints = {}
    for waypoint in data['waypoints']:
        name = waypoint['name']
        waypoints[name] = waypoint
    return waypoints


def create_pose(transform):
    """Create a PoseStamped message from a dictionary."""
    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


class PoseSubscriber(Node):
    """Node to subscribe to PoseStamped messages."""
    def __init__(self):
        super().__init__('ArUcoPose_subscriber')
        self.subscription = self.create_subscription(
            PoseStamped,
            '/aruco_single/pose',  # Topic name
            self.listener_callback,
```

```
            10
        )
        self.subscription  # Prevent unused variable warning

    def listener_callback(self, msg):
        """Callback to process incoming PoseStamped messages."""
        position = msg.pose.position
        orientation = msg.pose.orientation
        self.get_logger().info(
            f"Received Pose: Position: ({position.x}, {position.y}, {position.z}), "
            f"Orientation: ({orientation.x}, {orientation.y}, {orientation.z}, {orientation.w})"
        )


def main():
    rclpy.init()

    # Create a BasicNavigator instance
    global navigator
    navigator = BasicNavigator()

    # Load waypoints from the YAML file
    yaml_file_name = 'ArucoGoals.yaml'
    yaml_file = os.path.join(get_package_share_directory('rl_fra2mo_description'), "config", yaml_file_name)
    waypoints = load_waypoints(yaml_file)

    # Define the desired order of waypoint execution
    waypoint_order = ['Goal_4','Goal_5'] #'Goal_1','Goal_2','Goal_3',

    # Create the goal poses in the specified order
    goal_poses = [create_pose(waypoints[name]) for name in waypoint_order]
    print(goal_poses)

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

    # Start following the waypoints
    nav_start = navigator.get_clock().now()
    navigator.followWaypoints(goal_poses)

    # Start the ArUco Pose subscriber
    aruco_pose_subscriber = PoseSubscriber()

    # This will keep the node spinning and processing messages as they arrive
    rclpy.spin(aruco_pose_subscriber)

    # Monitor navigation task completion
    i = 0
    while not navigator.isTaskComplete():



        i += 1
        feedback = navigator.getFeedback()

```

```
102         if feedback and i % 5 == 0:
103             print('Executing current waypoint: ' +
104                   str(feedback.current_waypoint + 1) + '/' + str(len(
    goal_poses)))
105             now = navigator.get_clock().now()
106 
107             # Handle navigation timeout for demonstration
108             if now - nav_start > Duration(seconds=600):
109                 navigator.cancelTask()
110 
111 
112 
113     # Handle the result of the navigation
114     result = navigator.getResult()
115     if result == TaskResult.SUCCEEDED:
116         print('Goal succeeded!')
117     elif result == TaskResult.CANCELED:
118         print('Goal was canceled!')
119     elif result == TaskResult.FAILED:
120         print('Goal failed!')
121     else:
122         print('Goal has an invalid return status!')
123 
124 
125     aruco_pose_subscriber.destroy_node()
126 
127     exit(0)
128 
129 
130 if __name__ == '__main__':
131     main()
```

Listing 4.1: `ArucoRetrieve.py`

4.2 Publishing ArUco Pose as TF

In this final paragraph, we will explain the functionality of the last node launched in `fra2mo_run.launch.py` under the `true` condition of `aruco_retrieve`, namely the `Aruco_Tf` node. This node is responsible for publishing on the `/tf` topic a static tf: the position of the ArUco relative to the `map` frame. To achieve this, a package named `aruco_transform` was created, which contains the source code that implements the executable launched by the `Aruco_Tf`. Within the code file `static_aruco_tf2_broadcaster.cpp`, we find the definition of a class, `StaticFramePublisher`, which contains two private methods: `aruco_pose` and `make_transform`. During construction, a subscription is made to the `/aruco_single/pose` topic, from which we retrieve the position of the ArUco relative to the `map` frame, and a pointer to an object of type `StaticTransformBroadcaster` is initialized. The method `sendTransform` is then called on this object, which publishes a message of type

`geometry_msgs::msg::TransformStamped` on the `/tf` topic. Our goal now is to convert the message from `/aruco_single/pose` into a `geometry_msgs::msg::TransformStamped`, using the two methods implemented in the class. The first method, `aruco_pose`, is the callback of the subscription and is therefore called every time a message is available on `/aruco_single/pose`. This method stores the values from the message into two private fields of the class: a vector for the position and a vector for the quaternions. Within this method, the second method, `make_transform()`, is invoked. This method is responsible for

actually creating a `TransformStamped` message. First, it retrieves the current time, sets the `frame_id` to `map`, and the `child_frame_id` to `aruco_marker_frame`, which is the frame of the ArUco. It then populates the translation and rotation fields with the values stored in the member variables. Once the message is set, the `sendTransform` method is called on the `tf_static_broadcaster_` pointer, passing the `TransformStamped` message to be published.

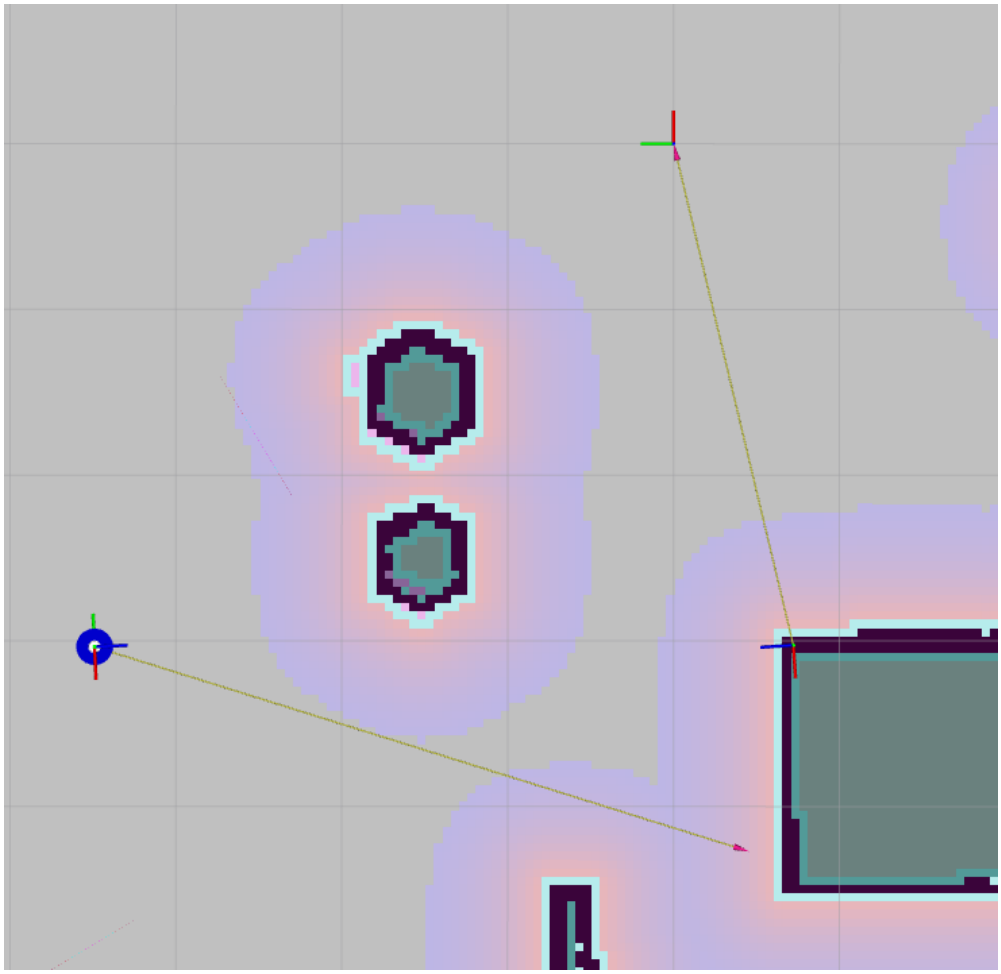


Figure 4.2: TF from map frame to ArUco Marker frame

```
#include <stdio.h>
#include <iostream>
#include <memory>
#include <cstdlib>
#include "geometry_msgs/msg/transform_stamped.hpp"
#include <geometry_msgs/msg/pose_stamped.hpp>
#include "rclcpp/rclcpp.hpp"
#include "rclcpp/wait_for_message.hpp"
#include "tf2/LinearMath/Quaternion.h"
#include "tf2_ros/static_transform_broadcaster.h"

using namespace std;

class StaticFramePublisher : public rclcpp::Node
{
public:
  explicit StaticFramePublisher()
  : Node("static_aruco_tf2_broadcaster")
  {
```

```
20      aruco_pose_available_ = false;
21      tf_static_broadcaster_ = std::make_shared<tf2_ros::
        StaticTransformBroadcaster>(this);
22      ArucoPos_ = this->create_subscription<geometry_msgs::msg::PoseStamped
        >(
23                      "/aruco_single/pose", 10, std::bind(&
        StaticFramePublisher::aruco_pose, this, std::placeholders::_1));
24      //  Waiting for the /aruco_single/pose topic
25    RCLCPP_INFO(this->get_logger(), "Aruco Pose not available ...");
26    }
27
28  private:
29    void make_transforms()
30    {
31      geometry_msgs::msg::TransformStamped t;
32
33      t.header.stamp = this->get_clock()->now();
34      t.header.frame_id = "map";
35      t.child_frame_id = "aruco_marker_frame";
36
37      t.transform.translation.x = vec[0];
38      t.transform.translation.y = vec[1];
39      t.transform.translation.z = vec[2];
40
41      t.transform.rotation.x = quat[0];
42      t.transform.rotation.y = quat[1];
43      t.transform.rotation.z = quat[2];
44      t.transform.rotation.w = quat[3];
45
46      tf_static_broadcaster_->sendTransform(t);
47    }
48
49    void aruco_pose(const geometry_msgs::msg::PoseStamped::SharedPtr msg){
50      //position
51      vec[0] = msg->pose.position.x;
52      vec[1] = msg->pose.position.y;
53      vec[2] = msg->pose.position.z;
54      //orientation
55      quat[0] =msg->pose.orientation.x;
56      quat[1] =msg->pose.orientation.y;
57      quat[2] =msg->pose.orientation.z;
58      quat[3] =msg->pose.orientation.w;
59      this->make_transforms();
60      aruco_pose_available_=true;
61      std::cout <<"Aruco Pose Published as TF!" << std::endl;
62    }
63
64    std::shared_ptr<tf2_ros::StaticTransformBroadcaster>
       tf_static_broadcaster_;
65    rclcpp::Subscription<geometry_msgs::msg::PoseStamped>::SharedPtr
       ArucoPos_;
66    bool aruco_pose_available_;
67    double vec[3];
68    double quat[4];
69
70  };
71
72  int main(int argc, char * argv[])
73  {
```

```
74   // Pass parameters and initialize node
75   rclcpp::init(argc, argv);
76   rclcpp::spin(std::make_shared<StaticFramePublisher>(/*argv*/));
77   rclcpp::shutdown();
78   return 0;
79 }
```

Listing 4.2: `static_aruco_tf2_broadcaster.cpp`