Robotics Lab

Report Homework 4

Federico Trenti P38000263
Matteo Russo P38000247
Alessandra Del Sorbo P38000289
Giulio Acampora P38000258

2024/2025

GitHub Links

- Link Trenti
- Link Russo
- Link Del Sorbo
- Link Acampora

Contents

Chapter 1

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

1.1 Launching the Gazebo simulation and spawning the mobile robot

The first step involves setting up a Gazebo simulation environment and spawning the mobile robot in the specified pose.
The launch file `/launch/gazebo_fra2mo.launch.py` was appropriately modified to achieve the required functionality. The necessary changes ensured that the mobile robot was spawned in the `leonardo_race_field` world at the specified pose $x = -3\,\text{m}$, $y = 3.5\,\text{m}$, and yaw angle $Y = -90°$. The updated launch file is shown below:

```
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)]
```

1.2 Modification of the World File

The world file `leonardo_race_field.sdf` was modified to relocate obstacle 9 to the specified position. The obstacle was moved to the coordinates $x = -3\,\mathrm{m}$, $y = -3.3\,\mathrm{m}$, $z = 0.1\,\mathrm{m}$, with a yaw angle of $Y = 90°$.
The modifications made to the `leonardo_race_field.sdf` file are detailed below:

```xml
<!-- caption=Modified section of leonardo_race_field.
    sdf  -->
<include>
  <name>obstacle_09</name>
  <!-- <pose> 5 5 0.1 0 0 3.14159</pose> -->
  <pose> -3 -3.3 0.1 0 0 1.57 </pose>
  <uri>model://obstacle_09</uri>
</include>
<!-- Other link properties remain unchanged -->
```

1.3 Placing the Aruco marker

An Aruco marker was placed on the obstacle 9 at the following coordinates:

- **x**: $x = -3.72\,\mathrm{m}$
- **y**: $y = -0.68\,\mathrm{m}$
- **z**: $z = 0.31\,\mathrm{m}$

with orientation angles of

- **roll**: $3.04°$
- **pitch**: $1.57°$
- **yaw**: $-1.7°$

These values were carefully chosen to ensure that the marker is clearly visible to the robot's camera when the robot is in the vicinity of the obstacle. The marker used was number 115, which was generated using the online tool available at `https://chev.me/arucogen/`. The generated marker image was then converted to the `.png` format for use in the simulation environment.
The Aruco marker was placed on obstacle 9 by appropriately modifying the `.sdf` file of the `leonardo_race_field` world. The updated code snippet is shown below:

```xml
<!-- Placement of ArUco Marker in leonardo_race_field.sdf -->
<include>
  <name>arucotag</name>
  <pose> -3.72 -0.68 0.31 3.04 1.57 -1.70 </pose>
  <uri>model://arucotag</uri>
</include>
```

To ensure that the Aruco marker could be detected, a camera sensor was added to the mobile robot. The integration of the camera was carried out as described in the previous homework assignment. The camera is mounted on the robot in a way that allows it to capture a clear view of the marker when the robot is near obstacle 9.

Chapter 2

Using the Nav2 Simple Commander API enable an autonomous navigation task

2.1 Definition of goals in a dedicated .yaml file

Four goals were defined in a dedicated `goals.yaml` file. The goals were originally specified with respect to the map frame, with the following poses:

- Goal 1: $x = 0\,\mathrm{m}, y = 3\,\mathrm{m}, Y = 0°$
- Goal 2: $x = 6\,\mathrm{m}, y = 4\,\mathrm{m}, Y = 30°$
- Goal 3: $x = 6.5\,\mathrm{m}, y = -1.4\,\mathrm{m}, Y = 180°$
- Goal 4: $x = -1.6\,\mathrm{m}, y = -2.5\,\mathrm{m}, Y = 75°$

However, since the robot does not spawn at the origin of the map, and the odometry frame is based on the robot's spawn location, the poses need to be transformed from the map frame to the robot's local frame. This requires applying both translations and rotations to the goal positions.
In our `goals.yaml` file, we defined the goals with the appropriate transformations, and the transformed coordinates are as follows:

- Goal 1: Transformed pose $x = 0.5\,\mathrm{m}, y = 3\,\mathrm{m}, Y = 90°$
- Goal 2: Transformed pose $x = -0.5\,\mathrm{m}, y = 9.0\,\mathrm{m}, Y = 120°$
- Goal 3: Transformed pose $x = 4.0\,\mathrm{m}, y = -4.4\,\mathrm{m}, Y = -90°$
- Goal 4: Transformed pose $x = -4.6\,\mathrm{m}, y = -5.5\,\mathrm{m}, Y = 165°$

Since the Nav2 Simple Commander API was used for autonomous navigation, all transformations were further converted to quaternions. This is necessary because the API operates with quaternions for orientation. Therefore, the goal poses in the `goals.yaml` file include quaternion representations and are reported below:

```yaml
goals:
  - name: "goal_1"
    position:
      x: 0.5
      y: 3.0
      z: 0.0
    orientation:
      x: 0.0
      y: 0.0
      z: 0.707107
      w: 0.707107

  - name: "goal_2"
    position:
      x: -0.5
      y: 9.0
      z: 0.0
    orientation:
      x: 0.0
      y: 0.0
      z: 0.866025
      w: 0.5

  - name: "goal_3"
    position:
      x: 4.9
      y: 9.5
      z: 0.0
    orientation:
      x: 0.0
      y: 0.0
      z: -1.0
      w: 0.0

  - name: "goal_4"
    position:
      x: 6.0
      y: 1.4
      z: 0.0
    orientation:
```

```
        x: 0.0
        y: 0.0
        z: 0.965926
        w: 0.258819
```

2.1.1 Sending and ordering goals for mobile robot

To ensure that the robot follows the goals in the specified order, we modified the `follow_waypoints.py` script. The goal order was set to follow a specific sequence: Goal 3 → Goal 4 → Goal 2 → Goal 1. This was achieved by rearranging the list of goals in the script.

In the modified `follow_waypoints.py` file, the following code was added to reorder the goals:

```python
# Reorder the goals based on the specified order from the task description
order = ["goal_3", "goal_4", "goal_2", "goal_1"]
ordered_goals = [goal for name in order for goal in waypoints["goals"] if goal["name"] == name]

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

goal_poses = list(map(create_pose, ordered_goals))
```

This is a nested list where the outer loop iterates over the list `order`, which contains the names of the goals in the desired order, while the inner loop iterates over the `waypoints["goals"]` list and selects the goal whose `"name"` matches the current goal name from `order`. This results in a new list `ordered_goals`, where the goals are ordered according to the specified

sequence.
Additionally, in the `fra2mo_explore.launch.py` file, we commented out the `explore_lite_launch` call to ensure that the map exploration follows our custom goal sequence rather than a default exploration strategy. The `explore_lite_launch` node handled automatic map exploration.

2.1.2 Recording the robot trajectory and plotting in the XY plane

To visualize the trajectory followed by the robot, we recorded a bagfile of the executed trajectory during the robot's navigation. We used the `/pose` topic to retrieve information about the robot's position throughout its path. For better visualization, we created a MATLAB script based on the one we had previously developed for Homework 2. This script was used to plot the robot's trajectory in the XY plane. In addition to the trajectory, we also plotted the four goal points to highlight that the robot successfully passed through these points during its navigation. Since the data was recorded in the robot's odometry frame, we transformed the coordinates into the map frame using the following coordinate transformation:

$$X' = Y + \text{offset}_x$$

$$Y' = -X + \text{offset}_y$$

The following MATLAB script was used to plot the trajectory and the goals:

```
% MATLAB script to plot the robot trajectory and goal
    points loaded from the bagfile
% Carico il file .db3 con ros2bagreader
bag = ros2bagreader('C:\Users\HP\Desktop\my_bag2\
    FirstBag_0.db3');  % percorso file
% Visualizzo i topic disponibili nel bag file
topicList = bag.AvailableTopics;
disp(topicList);
% Seleziono il topic che contiene i valori delle
    posizioni
msgs = readMessages(select(bag, 'Topic', '/pose'));
% Numero di messaggi letti
n = numel(msgs);
% Pre-allocazione per i dati
Values = zeros(n, 2);  % Poiche mi interessano solo x e
     y
offset_x = -3; % Traslazione lungo X
offset_y = 3.5; % Traslazione lungo Y
```

```
% Estrazione dei valori
for i = 1:n
    Values(i, 1) = msgs{i,1}.pose.pose.position.y + offset_x;
    Values(i, 2) = -msgs{i,1}.pose.pose.position.x + offset_y;
end
%Punti dei goals traslati (come nel yaml)
goalPoints = [0.5, 3;
              -0.5, 9;
              4.9, 9.5;
              6, 1.4];
a=size(goalPoints,1);
newGoalPoints = zeros(a, 2);
% Applicazione della rotazione e della traslazione
for i = 1:a
    newGoalPoints(i, 1) = goalPoints(i, 2) + offset_x; % X' = Y + offset_x
    newGoalPoints(i, 2) = -goalPoints(i, 1) + offset_y; % Y' = -X + offset_y
end
% Plot dei Goal Points trasformati
plot(newGoalPoints(:,1), newGoalPoints(:,2), 'ro', 'MarkerSize', 8, 'MarkerFaceColor', 'none', 'DisplayName', 'Goal Points');
hold on;
grid on;
% Plot della traiettoria
plot(Values(:, 1), Values(:, 2), 'b-', 'DisplayName', 'Trajectory');
hold on;
% Impostazione dei limiti degli assi con un margine
xlim([min(Values(:,1))-1, max(Values(:,1))+1]);
ylim([min(Values(:,2))-1, max(Values(:,2))+1]);
% Legenda e grafico
xlabel('X');
ylabel('Y');
title('Plot of the trajectory in XY plane');
legend show;
axis equal
grid on;
hold off;
```

The resulting plot shows the trajectory in blue, with the goal points marked in red. This visualization confirms that the robot correctly passed through the four defined goal points during its navigation.

Below is the generated plot:

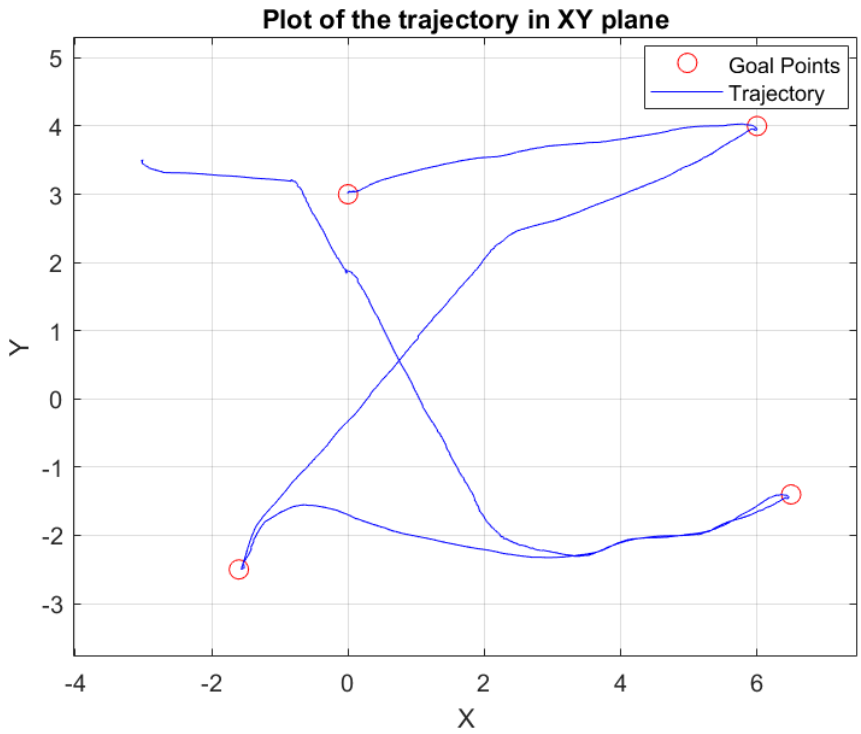

Figure 2.1: Robot trajectory with goal points

Chapter 3

Map the environment tuning the navigation stack's parameters

3.1 Modifying goals for complete Map mapping

To complete the mapping of the environment, we utilized six newly added goals along with **Goal 4**, excluding the previous three goals from the navigation sequence. The new goals were strategically placed at key locations within the map to ensure the robot covered all critical areas for a full exploration of the environment.
The new goals were defined in the `goals.yaml` file as follows:

- Goal 5: $x = 3.36\,\mathrm{m}, y = 6.94\,\mathrm{m}$
- Goal 6: $x = -0.99\,\mathrm{m}, y = 11.99\,\mathrm{m}$
- Goal 7: $x = 5.47\,\mathrm{m}, y = 12.34\,\mathrm{m}$
- Goal 8: $x = 8.01\,\mathrm{m}, y = 6.68\,\mathrm{m}$
- Goal 9: $x = 6.36\,\mathrm{m}, y = -3.68\,\mathrm{m}$
- Goal 10: $x = 2.36\,\mathrm{m}, y = -5.68\,\mathrm{m}$

In addition to these, **Goal 4** was retained from the original set to assist with mapping a specific section of the environment. The execution of the goals was performed in the following order: Goal 5 → Goal 6 → Goal 7 → Goal 8 → Goal 4 → Goal 9 → Goal 10.
After adding these goals, the robot was guided to each of them, ensuring a complete mapping of the environment. The map was then saved using the following ROS2 command:

```
$ ros2 run nav2_map_server map_saver_cli -f map
```

This command generated the final map of the environment, which was saved within the `maps` directory. The following image of the map is included to illustrate the completed map after exploration:

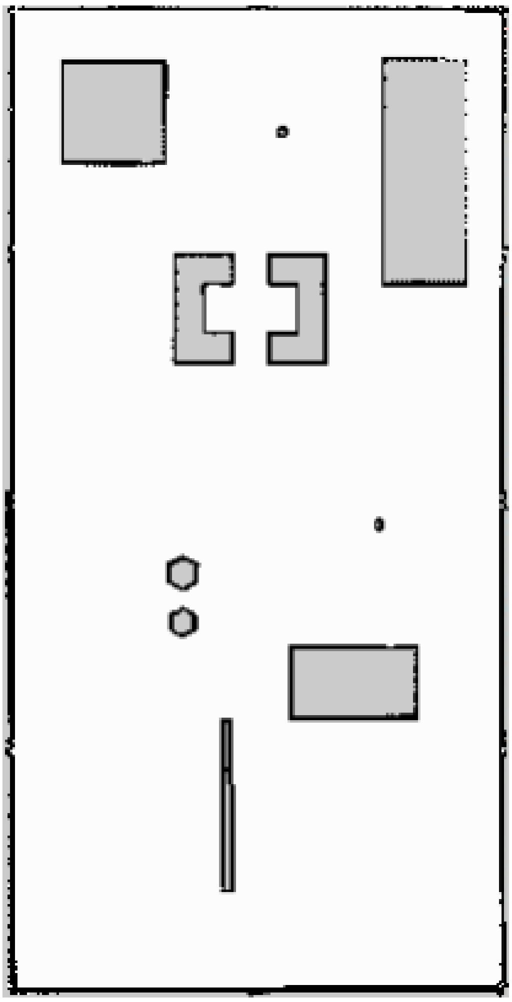

Figure 3.1: Final map of the environment

3.2 Tuning navigation configuration parameters

To optimize the robot's navigation and mapping performance, we modified several parameters in the configuration files. These changes were made to test different behaviors and evaluate their impact on the system's efficiency and accuracy. Specifically, we adjusted parameters in `slam.yaml` and `explore.yaml`.

Parameter Changes in `slam.yaml`

In the `slam.yaml` file, the following parameters were modified:

- **minimum_travel_distance:** This parameter was adjusted to test how far the robot needs to move before updating the map.

- **minimum_travel_heading:** This parameter was tuned to explore its effect on map updates when the robot changes its heading direction.

- **resolution:** The map resolution was varied to balance map detail and computational efficiency.

- **transform_publish_period:** This parameter was adjusted to control how frequently transforms were published.

Parameter Changes in `explore.yaml`

In the `explore.yaml` file, the following parameters were modified:

- **inflation_radius:** This parameter was varied to test the size of the safety buffer around obstacles.

- **cost_scaling_factor:** This parameter, which defines how quickly the cost decreases as the robot moves away from obstacles, was adjusted.

3.2.1 Parameter Configurations

We conducted four tests by modifying key parameters in the `slam.yaml` file.

Tests

Resolution Adjustment We tested the system with different values for the `resolution` parameter:

- **Resolution = 0.05:** The trajectory was completed perfectly in **6.11 minutes**. The generated map was saved as in Fig. 3.2. This resolution provided accurate and reliable navigation.

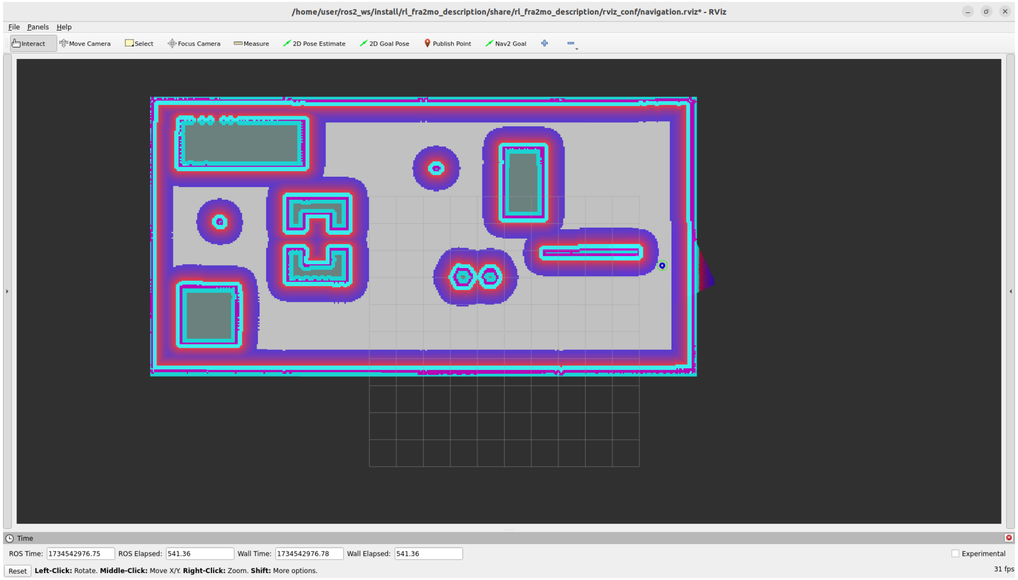

Figure 3.2: Map of the environment with standard parameters

- **Resolution = 0.3:** The robot failed to follow the correct trajectory from the first goal. It stalled in front of the labyrinth for **1.20 minutes** without entering, eventually abandoning the goal and moving to the second goal (`goal 6`). After reaching it, the robot paused again for 3.30 minutes and skipped goals `7` and `8`, attempting to move directly to `goal 4`. This test highlights the limitations of higher resolution in environments requiring precise navigation. The map was saved as in Fig. 3.3.

- **Resolution = 0.005:** The robot remained stationary at the spawn position for **1.30 minutes**, with only orientation changes. The partial map generated was saved as in Fig. 3.4.

- **Resolution = 0.1:** The task was completed successfully in **4.55 minutes**. The generated map was saved as in Fig. 3.5. This value showed good performance with minor improvements in speed compared to the baseline.

- **Resolution = 0.01:** Despite the robot identifying the first goal (entering the labyrinth), it experienced delays (**30 seconds** to start and frequent stops). After **7 minutes**, the simulation was terminated manually without success. The partial map was saved as in Fig. 3.6.

- **Resolution = 0.035:** The robot started faster than in previous tests and completed up to Goal 4 but failed to reach Goals 9 and 10. The task ended after **10.10 minutes** with incomplete results. The map was saved as in Fig. 3.7.

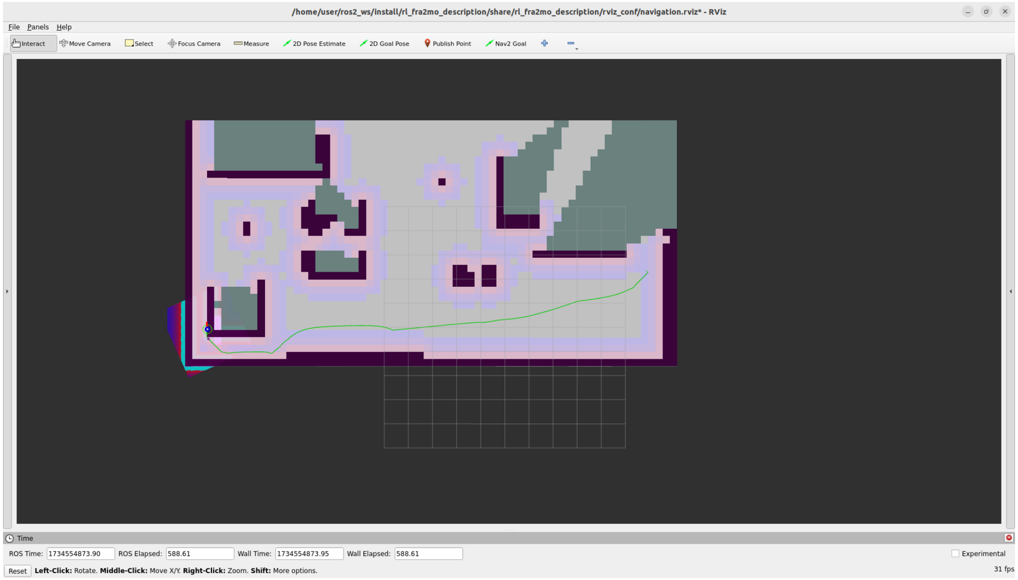

Figure 3.3: Map of the environment with Resolution = 0.3

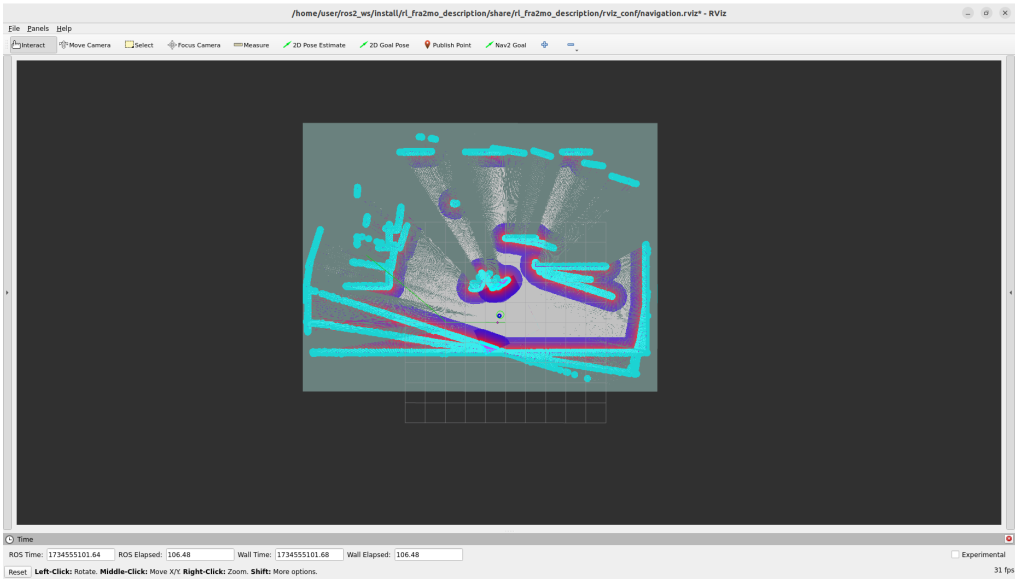

Figure 3.4: Map of the environment with Resolution = 0.005

- **Resolution = 0.065:** The robot performed well, with reduced stalling compared to earlier tests. All goals were reached, and the task was completed in **5.50 minutes**. The map was saved as in Fig. 3.8, making this configuration the best-performing one overall.

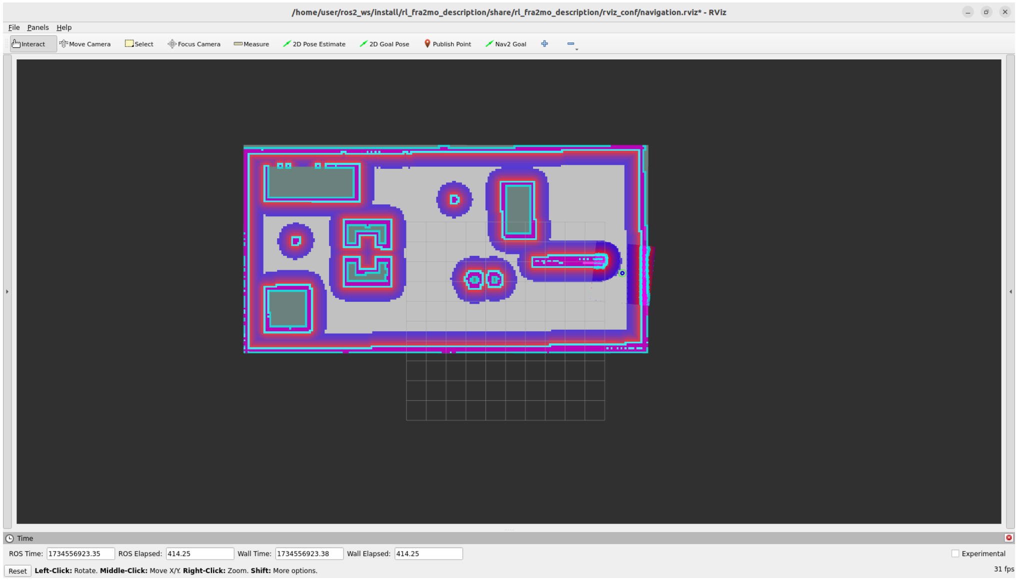

Figure 3.5: Map of the environment with Resolution = 0.1

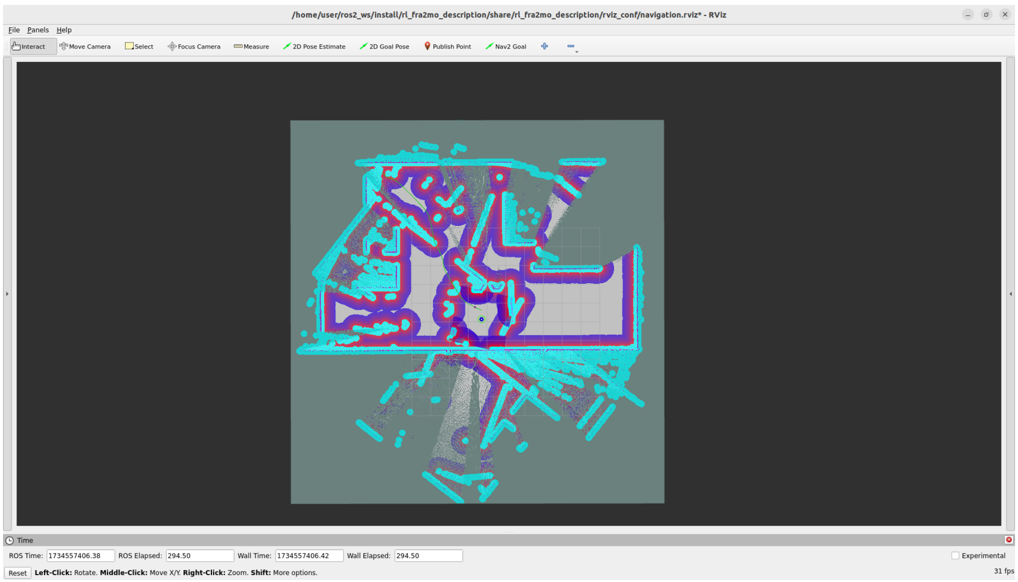

Figure 3.6: Map of the environment with Resolution = 0.01

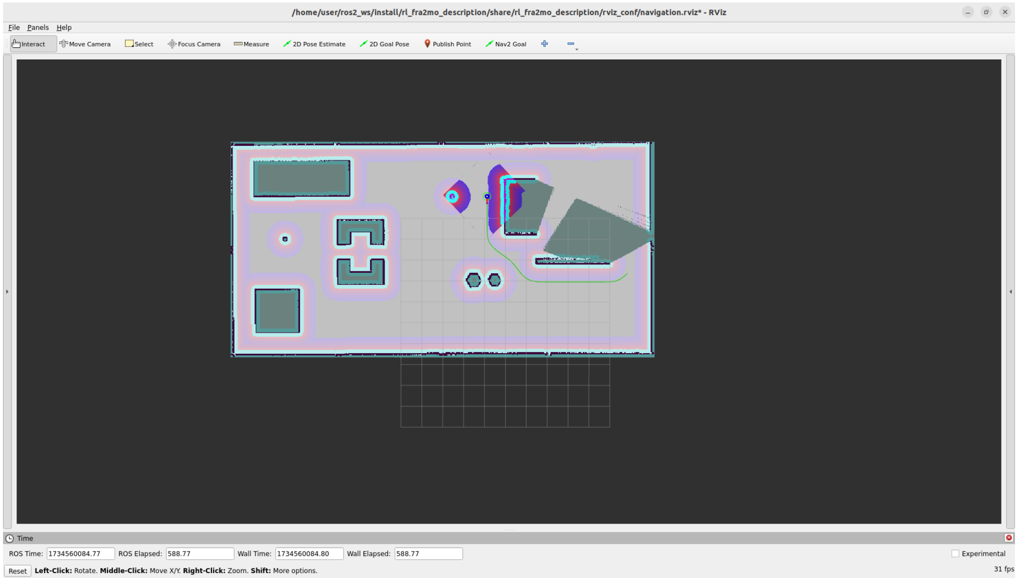

Figure 3.7: Map of the environment with Resolution = 0.035

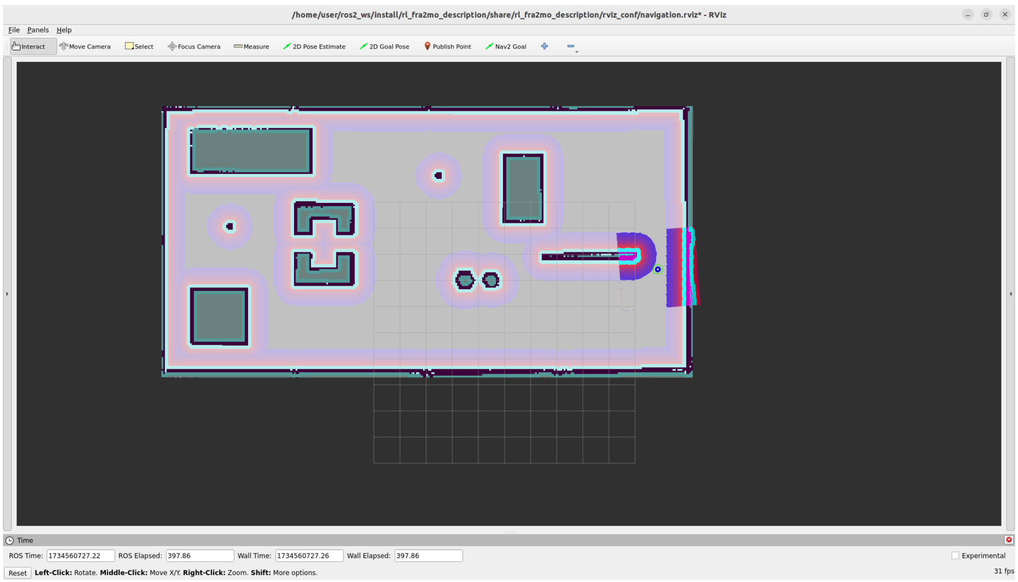

Figure 3.8: Map of the environment with Resolution = 0.065

Minimum Travel Distance The minimum travel distance parameter was evaluated with five different configurations to understand its effect on robot trajectory and mapping performance:

- **M.T.D. = 0.01:** This value corresponds to the original configuration, with a completion time of **6.11 minutes**. The map was saved as in Fig. 3.2.
- **M.T.D. = 0.1:** The robot started immediately and moved smoothly along the trajectory without stalling. The task was completed in **4.15 minutes**, and the map was saved as in Fig. 3.9. This configuration was identified as the **best-performing**.

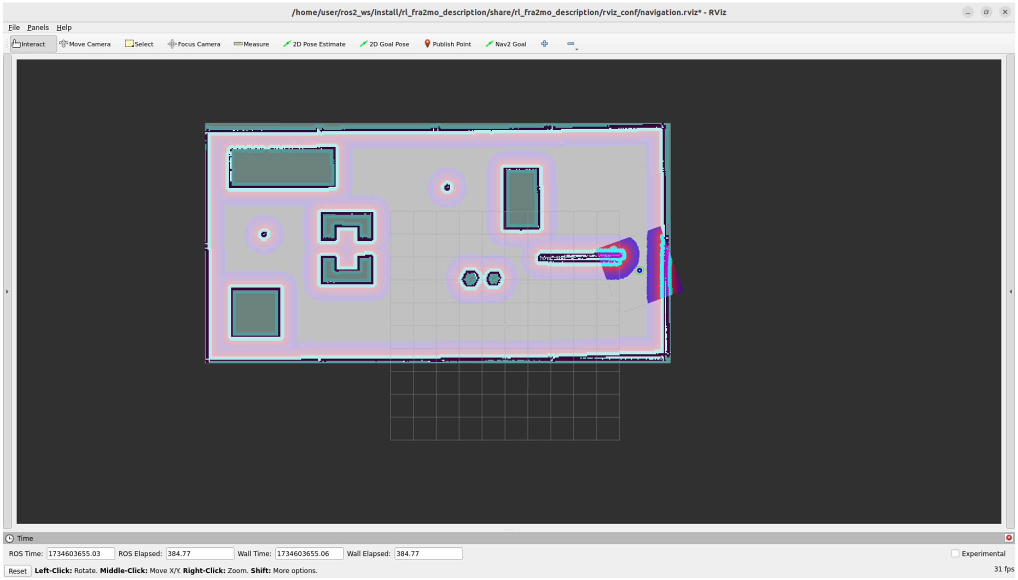

Figure 3.9: Map of the environment with M.T.D. = 0.1

- **M.T.D.= 0.25:** Although the robot started promptly, it experienced minor stalling during the trajectory. The task was completed in **4.50 minutes**, with the map saved as in Fig. 3.10.
- **M.T.D. = 0.06:** The robot exhibited significant stalling and occasional stops during the trajectory. It completed the task in **5.50 minutes**, and the map was saved as in Fig. 3.11.
- **M.T.D. = 0.006:** The robot's performance degraded considerably, skipping Goals 5, 6, and 7. It went directly to Goal 8, followed by Goals 4 and 9, leaving the left side of the map unmapped. The task was completed in **6.30 minutes**, and the map was saved as in Fig. 3.12.

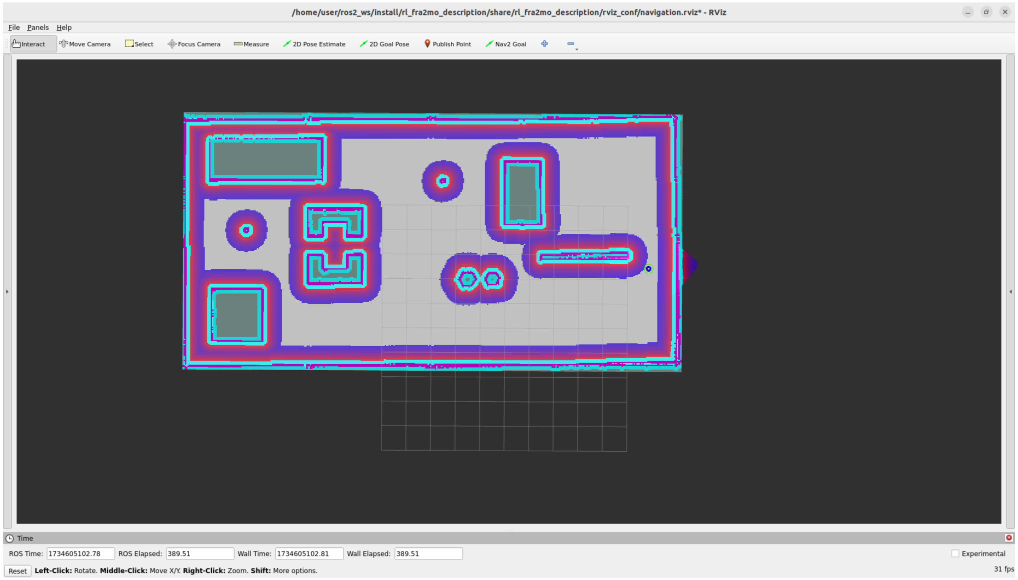

Figure 3.10: Map of the environment with M.T.D. = 0.25

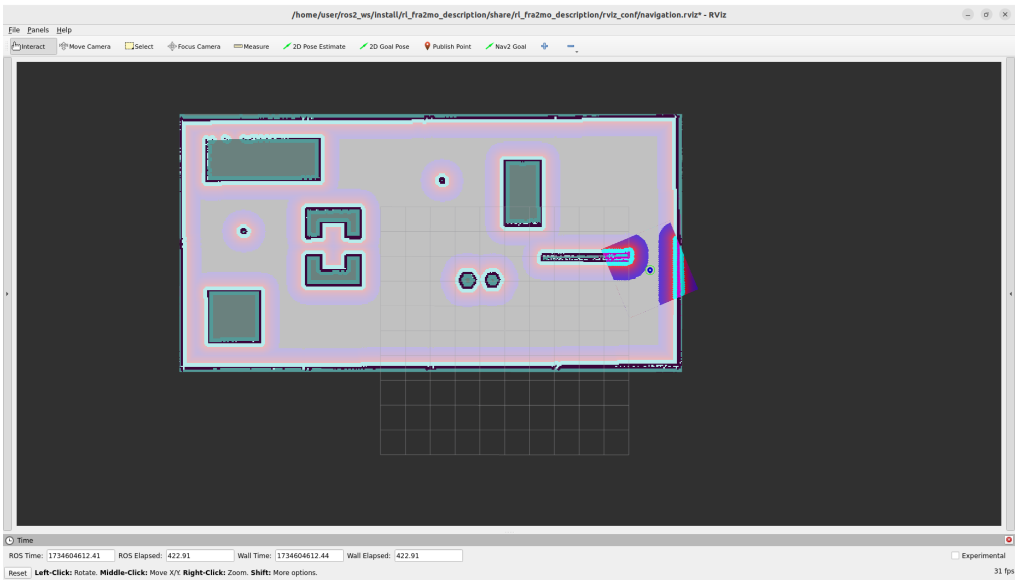

Figure 3.11: Map of the environment with M.T.D. = 0.06

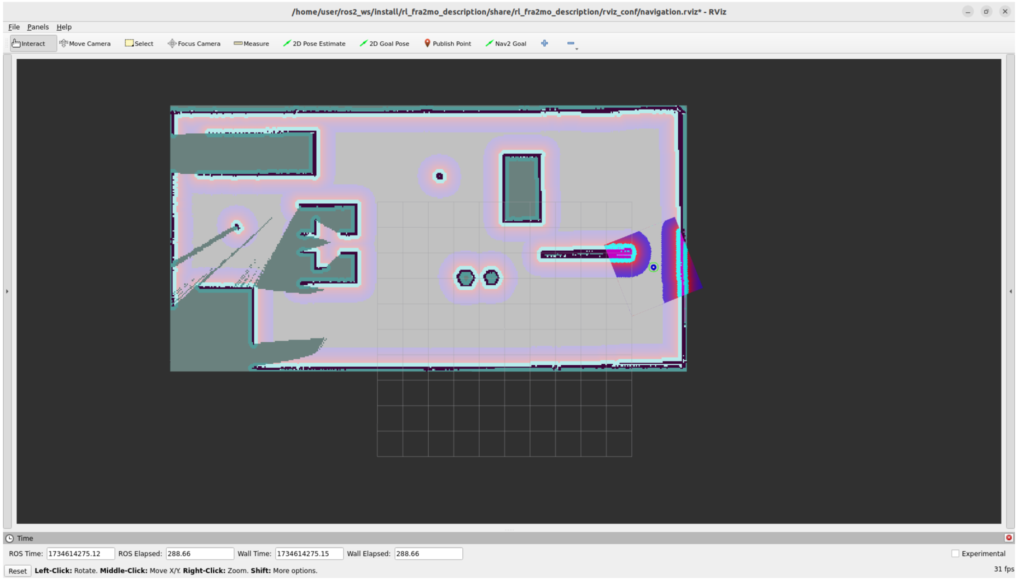

Figure 3.12: Map of the environment with M.T.D. = 0.006

Minimum Travel Heading The minimum travel heading parameter was tested with three configurations, yielding the following results:

- **M.T.H. = 0.01:** The original configuration completed the task in **6.11 minutes**, with the map saved as in Fig. 3.2.
- **M.T.H. = 0.1:** The robot completed the trajectory in **5.40 minutes** but moved with noticeable stalling. The map was saved as in Fig. 3.13.

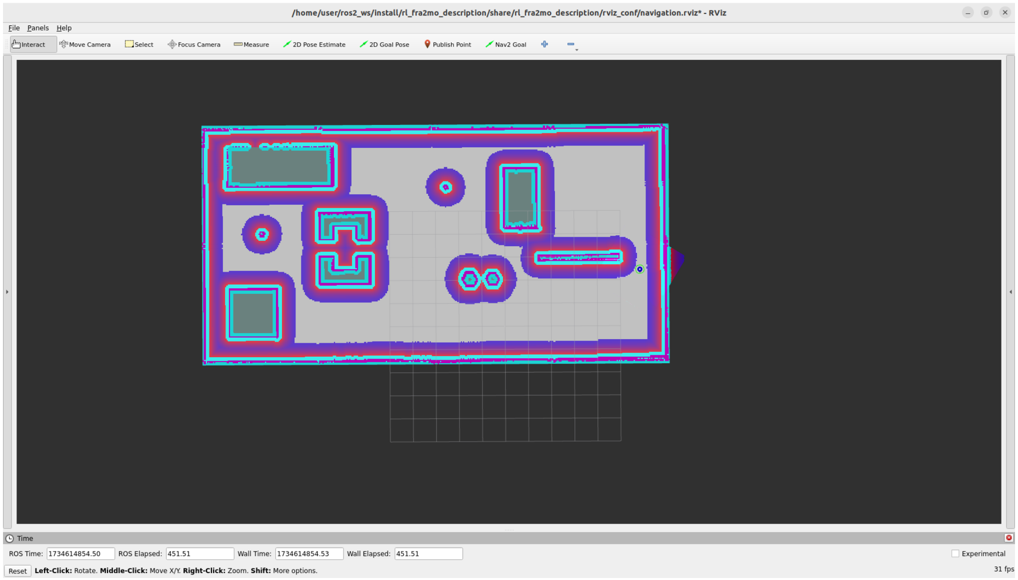

Figure 3.13: Map of the environment with M.T.H. = 0.1

- **M.T.H. = 0.05:** The robot exhibited smoother movements with reduced stalling. The task was completed in **5.20 minutes**, and the map was saved as in Fig. 3.14. The smoother trajectory suggests that higher values reduce computational load by minimizing map updates for small rotations.

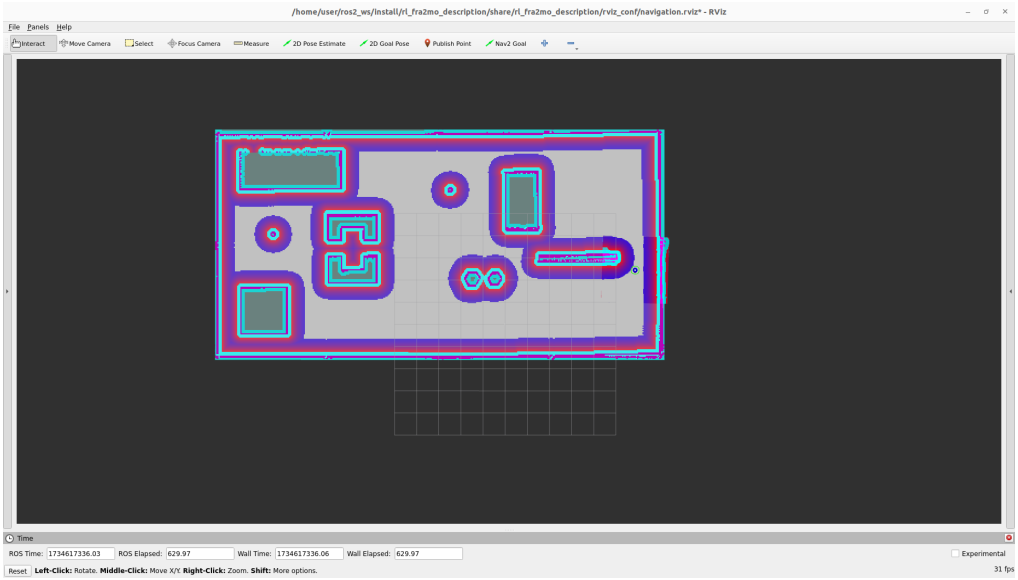

Figure 3.14: Map of the environment with M.T.H. = 0.05

Transform Publish Period The transform publish period parameter was tested with three configurations to evaluate its impact on trajectory and mapping:

- **T.P.P. = 0.02:** The original configuration completed the task in **6.11 minutes**, with the map saved as in Fig. 3.2.
- **T.P.P. = 0.2:** The robot generated the map but skipped Goal 9 after reaching Goal 4. It attempted to reach Goal 10 but failed, resulting in a manually terminated simulation after **8 minutes**. The map lacked the right portion and was saved as in Fig. 3.15.

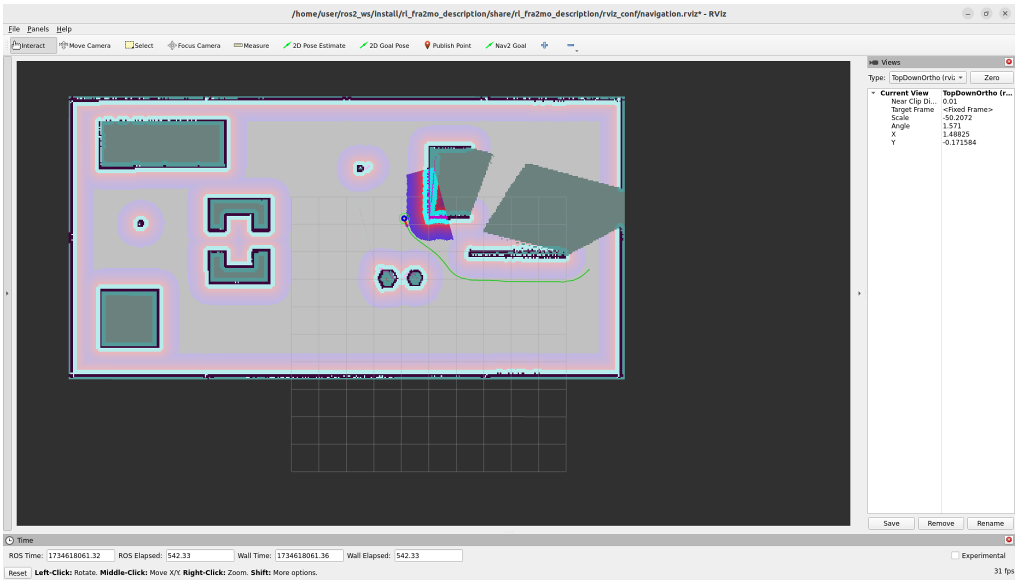

Figure 3.15: Map of the environment with T.P.P. = 0.2

- **T.P.P. = 0.08:** The task was completed successfully in **6.40 minutes**, with the map saved as in Fig. 3.16.

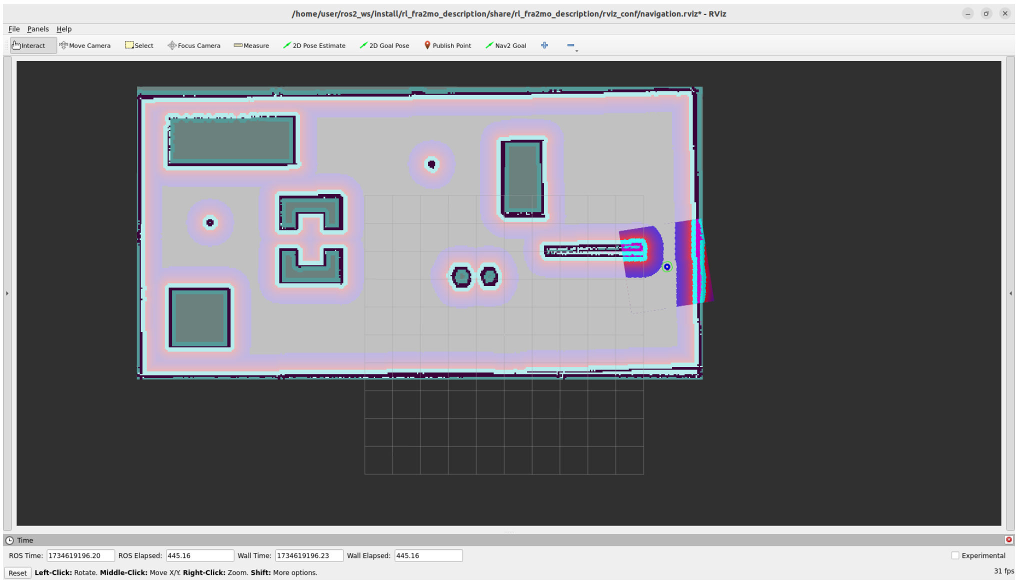

Figure 3.16: Map of the environment with T.P.P. = 0.08

Inflation Radius

- **Inflation Radius = 0.75** The robot exhibited behavior is the same of the default settings from previous tests, completing the trajectory in **6.11 minutes**. The resulting map is in Fig. 3.2.

- **Inflation Radius = 0.1** With a reduced inflation radius, the robot took **7.18 minutes** to complete the map. The resulting map was saved as in Fig. 3.17.

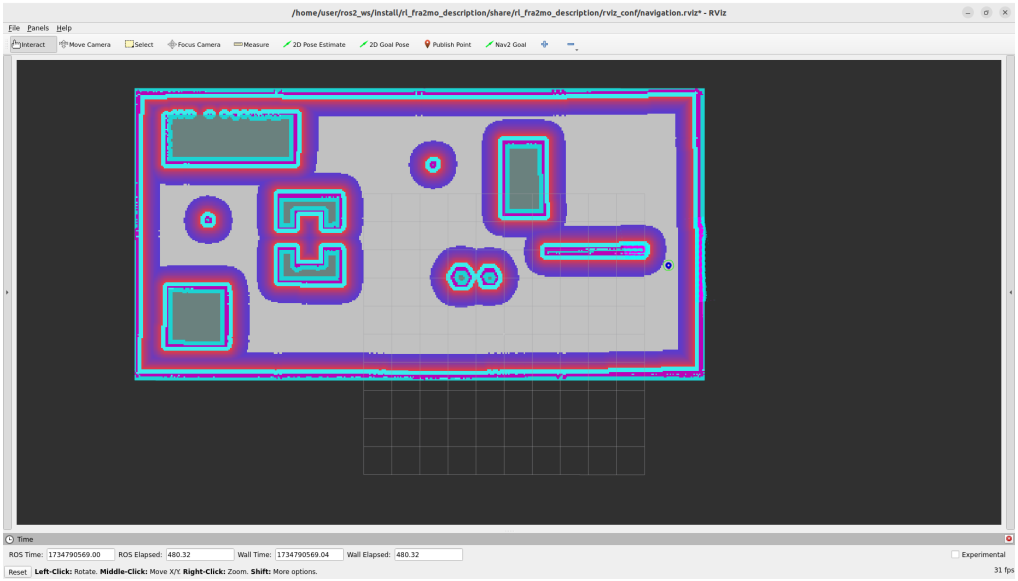

Figure 3.17: Map of the environment with Inflation Radius = 0.1

Cost Scaling Factor

- **Cost Scaling Factor = 3.0** The robot completed the trajectory in **6.11 minutes**, producing the map in Fig. 3.2.

- **Cost Scaling Factor = 6.0** Increasing the cost scaling factor resulted in slightly more cautious navigation, with the robot completing the map in **7.0 minutes**. The saved map was labeled as in Fig. 3.18.

Combined Parameters (`Resolution`, `M.T.D.`, `M.T.H.`)

Using the best values identified in earlier tests:

- **Resolution**: 0.065

- **Minimum Travel Distance** : 0.1

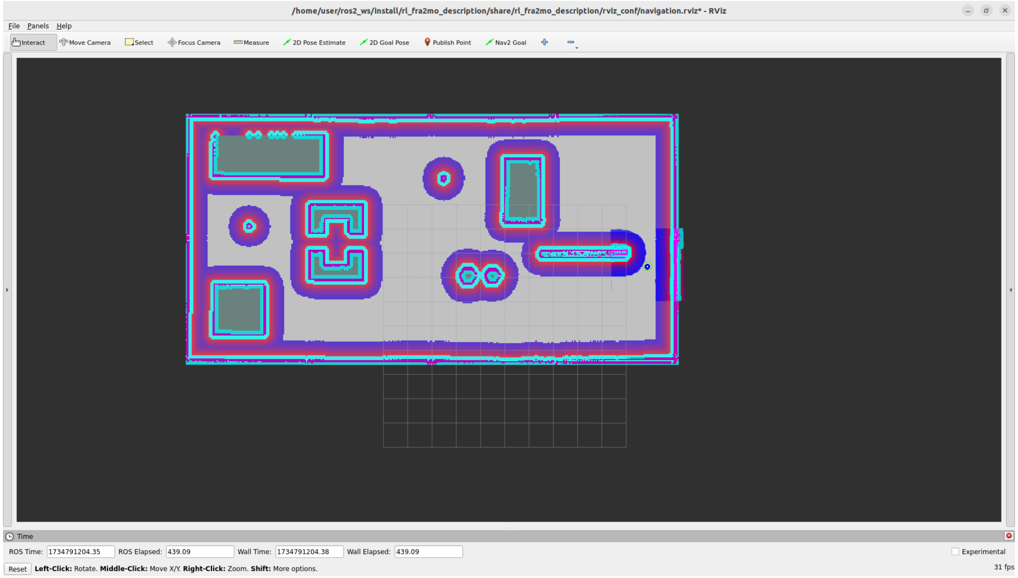

Figure 3.18: Map of the environment with Cost Scaling Factor = 6.0

- **Minimum Travel Heading** : 0.05

The robot achieved the fastest trajectory completion time of **3.90 minutes**. The resulting map was both accurate and efficiently generated. This combination of parameters optimizes the balance between computational load and mapping fidelity.

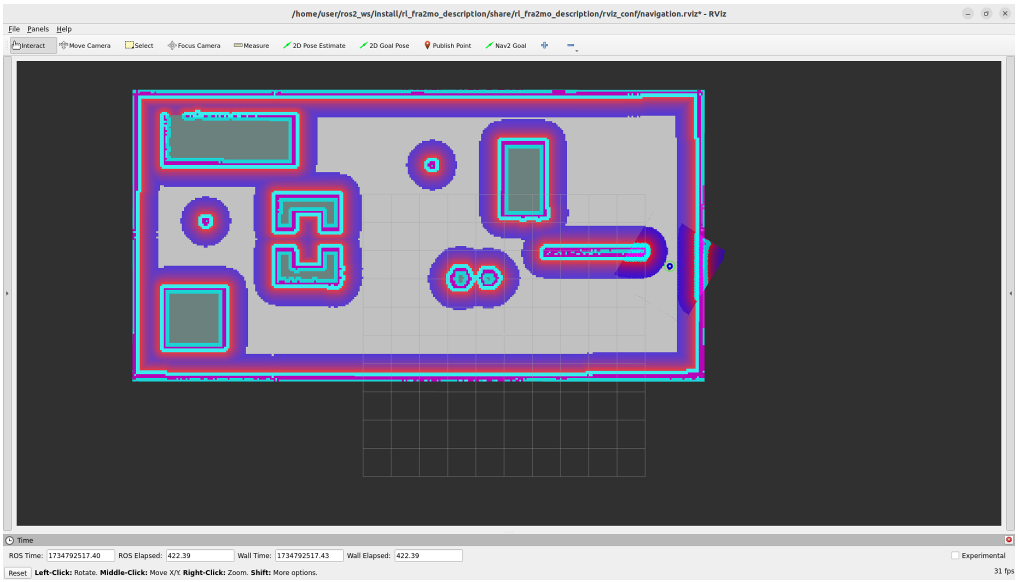

Figure 3.19: Map of the environment with combined parameters of Resolution, M.T.D., M.T.H.

3.2.2 Final Considerations

1. **Resolution** : Lower values improve map accuracy but increase computational load. A value of 0.065 strikes an optimal balance, producing precise maps with efficient execution times.

2. **Minimum Travel Distance** : A value of 0.1 reduces unnecessary movement while maintaining smooth trajectories, avoiding the lag and disruptions observed with lower values.

3. **Minimum Travel Heading** : Setting 0.05 minimizes computational overhead by limiting frequent adjustments in rotation, resulting in smoother navigation and faster mapping.

4. **Transform Publish Period** :Shorter periods improved map and trajectory updates but increased computational load. Higher values risk incomplete maps. The original value (0.02) provided a reliable baseline resulting in less precise trajectories.

5. **Inflation Radius and Cost Scaling Factor**: Lowering the inflation radius (0.1) and increasing the cost scaling factor (6.0) result in more cautious navigation, which may be helpful in obstacle-dense environments. However, default values provide a good balance for general mapping tasks.

6. **Overall Best Combination**: The combination of `resolution` = 0.065, `M.T.D.` = 0.1, and `M.T.H.` = 0.05 delivered the most efficient and accurate results, completing the mapping task in just **3.90 minutes**.

These results highlight the importance of fine-tuning navigation parameters to optimize both the speed and accuracy of autonomous exploration tasks.

Chapter 4

Vision-based navigation of the mobile platform

4.1 Creation of a launch File for navigation and Aruco marker detection

To integrate navigation with the detection of Aruco marker using the robot's camera, we created a new launch file named `vision_navigation.launch.py`. This launch file runs both the `follow_waypoint` node for goal navigation and the `fra2mo_explore.launch.py` file, which handles the exploration and mapping processes. Below is the structure of the `vision_navigation.launch.py` file:

```
#vision_navigation.launch.py
def generate_launch_description():
    explore_launch = IncludeLaunchDescription(
        PythonLaunchDescriptionSource([
            PathJoinSubstitution([
                FindPackageShare("rl_fra2mo_description"),
                "launch",
                "fra2mo_explore.launch.py"
            ])
        ])
    )

    follow_waypoint_node = Node(
        package='rl_fra2mo_description',
        executable='follow_waypoints.py',
        parameters=[{"target": "aruco"}],  # Specifica il parametro target
        output='screen'
```

```
    )
    # Declare arguments
    declared_arguments = [
        DeclareLaunchArgument(
            'marker_id',
            default_value='115',
            description='ID del marker ArUco da rilevare.'
        ),
        DeclareLaunchArgument(
            'marker_size',
            default_value='0.1',
            description='Dimensione del marker ArUco in metri.'
        ),
        DeclareLaunchArgument(
            'camera_topic',
            default_value='/videocamera',
            description='Topic della fotocamera da usare per il rilevamento.'
        ),
        DeclareLaunchArgument(
            'camera_info_topic',
            default_value='/camera_info',
            description='Topic delle informazioni della fotocamera.'
        ),
        DeclareLaunchArgument(
            'marker_frame',
            default_value='aruco_marker_frame',
            description='Frame ID associato al marker.'
        ),
        DeclareLaunchArgument(
            'reference_frame',
            default_value='map',
            description='Frame di riferimento per la posa calcolata.'
        ),
        DeclareLaunchArgument(
            'corner_refinement',
            default_value='LINES',
            description='Metodo di affinamento degli angoli del marker.'
        ),
        DeclareLaunchArgument(
            'camera_frame',
```

```
            default_value='camera_link_optical',
            description='Metodo di affinamento degli angoli del marker.'
        ),

    ]
    # Node configuration
    aruco_single_node = Node(
        package='aruco_ros',
        executable='single',
        name='aruco_single',
        output='screen',
        parameters=[{
            'image_is_rectified': True,
            'marker_size': LaunchConfiguration('marker_size'),
            'marker_id': LaunchConfiguration('marker_id'),
            'reference_frame': 'map',
            'camera_frame': 'camera_link_optical',
            'marker_frame': LaunchConfiguration('marker_frame'),
            'corner_refinement': LaunchConfiguration('corner_refinement'),
            # 'use_sim_time': True
        }],
        remappings=[
            ('/camera_info', LaunchConfiguration('camera_info_topic')),
            ('/image', LaunchConfiguration('camera_topic')),
        ]
    )
    return LaunchDescription([
        *declared_arguments,
        explore_launch,
        follow_waypoint_node,
        aruco_single_node,
        posebroadcaster_node
    ])
```

The `vision_navigation.launch.py` file ensures that both the navigation and Aruco marker detection processes are run simultaneously.

4.2 Implementation of a 2D navigation task

To complete the navigation task, the robot was programmed to follow the outlined logic:

- Navigate to the proximity of obstacle 9.
- Detect the Aruco marker using the robot's camera. Upon detection, retrieve the marker's pose with respect to the map frame.
- Return the robot to its initial position.

To publish the Aruco marker pose as a static transformation, we implemented the `PoseToTFBroadcaster.py` node, which subscribes to the `/aruco_single/pose` topic and broadcasts the pose on `/tf_static` topic, following the example in the ROS2 Humble documentation at this link.

4.2.1 Proximity navigation to the Aruco marker

A new goal, `goal 11`, was added to the `goals.yaml` file. This goal was set at the following coordinates to position the robot near the Aruco marker:

- $x = 3.72$ m,
- $y = -0.74$ m

The robot navigated to this location, ensuring it was close enough to detect the marker.

4.2.2 Aruco marker detection and pose retrieval

When the robot reached the vicinity of `goal 11`, it detected the Aruco marker using the same approach as implemented in Homework 3. Once the marker was detected, its pose relative to the map frame was retrieved. To achieve this, a new python script, `PoseToTFBroadcaster.py`, was created. This script is a ROS2 node responsible for retrieving the pose of the detected Aruco marker and broadcasting its transformation in the map frame.
The node `aruco_sub_pub` subscribes to the `/aruco_single/pose` topic, which provides the pose of the detected Aruco marker as a `PoseStamped` message.
The pose data is converted into a `TransformStamped` message, where:

- The parent frame is set to `map`.
- The child frame is set to `aruco_marker_frame`.

The position and orientation of the Aruco marker are then broadcasted as a static transformation on the `/tf_static` topic. To prevent redundant computations, a flag `marker_published` is used to ensure the transformation is published only once.
Below is the structure of the `PoseToTFBroadcaster.py` script:

```python
#PoseToTFBroadcaster.py
class PoseToTFBroadcaster(Node):
    def __init__(self):
        super().__init__('aruco_sub_pub')

        # Subscriber al topic /aruco_single/pose
        self.subscription = self.create_subscription(
            PoseStamped,
            '/aruco_single/pose',
            self.pose_callback,
            10
        )

        self.tf_static_broadcaster = StaticTransformBroadcaster(self)
        self.marker_published = False
        self.subscription
        self.get_logger().info('Nodo avviato e in ascolto su /aruco_single/pose')

    def pose_callback(self, msg):

        if self.marker_published:
            # Se il marker e' gia' stato pubblicato, evita ulteriori aggiornamenti
            return

        # Converte il messaggio Pose in un TransformStamped
        t = TransformStamped()

        t.header.stamp = self.get_clock().now().to_msg()
        t.header.frame_id = 'map'  # Frame di riferimento principale
        t.child_frame_id = 'aruco_marker_frame'  # Frame del marker

        self.get_logger().info('Transformazione pubblicata su /tf_static')
```

```
        # Imposta posizione
        t.transform.translation.x = msg.pose.position.x
        t.transform.translation.y = msg.pose.position.y
        t.transform.translation.z = msg.pose.position.z

        # Imposta orientamento
        t.transform.rotation.x = msg.pose.orientation.x
        t.transform.rotation.y = msg.pose.orientation.y
        t.transform.rotation.z = msg.pose.orientation.z
        t.transform.rotation.w = msg.pose.orientation.w

        # Pubblica la trasformazione su /tf_static
        self.tf_static_broadcaster.sendTransform(t)
        self.marker_published = True  # Imposta il flag a True
        self.get_logger().info('Trasformazione statica pubblicata per il marker.')
        #self.get_logger().info('Transformazione pubblicata su /tf_static')

def main():
    rclpy.init()
    node_aruco = PoseToTFBroadcaster()
    try:
        rclpy.spin(node_aruco)
    except KeyboardInterrupt:
        node_aruco.get_logger().info('Nodo interrotto dall utente')
    finally:
        node_aruco.destroy_node()
        rclpy.shutdown()

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

4.2.3 Return to initial position

After retrieving the ArUco marker's pose, the robot was commanded to return to its starting position, defined as `goal 0`, with the following coordinates:

```
goals:
  - name: "goal_0"
    position:
      x: 0.0
      y: 0.0
```

```
        z: 0.0
      orientation:
        x: 0.0
        y: 0.0
        z: 0.0
        w: 0.0
```

The implemented logic allowed the robot to successfully navigate to the vicinity of the ArUco marker, detect it, and retrieve its pose in the map frame. The robot then returned to its initial position, completing the navigation task.

To simplify the execution of the `PoseToTFBroadcaster.py` script, we integrated its functionality directly into the `vision_navigation.launch.py` file. This addition ensures that the node is launched automatically as part of the vision navigation setup.

The following configuration was added to the `vision_navigation.launch.py` file:

```
#Node addition for PoseToTFBroadcaster
posebroadcaster_node = Node(
    package='rl_fra2mo_description',
    executable='PoseToTFBroadcaster.py',
    output='screen'
)
```

By adding this node, the `PoseToTFBroadcaster.py` script is executed automatically, allowing the `/aruco_single/pose` topic to be subscribed to and the marker pose to be broadcasted on `/tf_static` without manual intervention. This modification enhances the automation and usability of the navigation pipeline.