Robotics Lab course

Homework 4
Control a mobile robot to follow a trajectory

Student:
Vincenzo Palomba
P38000180

Teacher:
Prof. Mario Selvaggio
Assistants:
Andrea Capuozzo
Claudio Chiariello

GitHub Repository

It is possible to find all the code used in this report at: https://github.com/vincip99/RoboticsLabHW4.

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

(**a**) Launch the Gazebo simulation `/launch/gazebo_fra2mo.launch.py` and spawn the mobile robot in the world `leonardo_race_field` in the pose:

$$x = -3 \text{ m}, \; y = 3.5 \text{ m}, \; yaw = -90 \; deg$$

with respect to the map frame. The argument for the yaw in the call of `spawn_model` is `Y`.

Into the `gazebo_fra2mo.launch.py` file the wollowing arguments were added to spwan the robot accordingly in the desired initial pose:

```
# Declare launch arguments for robot pose
declared_arguments.append(
    DeclareLaunchArgument(
        'x',
        default_value='-3.0',
        description='Initial X position of the robot'
    )
)
declared_arguments.append(
    DeclareLaunchArgument(
        'y',
        default_value='3.5',
        description='Initial Y position of the robot'
    )
)
declared_arguments.append(
    DeclareLaunchArgument(
        'yaw',
        default_value=str(-pi / 2),
        description='Initial yaw (orientation in radians) of the robot'
    )
)

# Substitutions for position and yaw
spawn_x = LaunchConfiguration('x')
spawn_y = LaunchConfiguration('y')
spawn_yaw = LaunchConfiguration('yaw')
```

Listing 1: launch file: gazebo_fra2mo.launch.py

(**b**) Modify the world file `leonardo_race_field.sdf` moving the obstacle 9 in position:

$$x = -3 \text{ m}, \; y = -3.3 \text{ m}, \; z = 0.1 \text{ m}, \; yaw = 90 \text{ deg}$$

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

Listing 2: gazebo world: leonardo_race_field.sdf

(**c**) Place the ArUco marker number 115 on obstacle 9 in an appropriate position, such that it is visible by the mobile robot's camera (you have to add it to the robot) when it comes in the proximity of the object.

First, the camera was added via a macro definition. The below XML snippet defines a fixed joint (`camera_joint`) and links for a camera sensor attached to the robot. It includes a simple visual

representation, a fixed optical frame (`camera_optical_joint`) and Gazebo simulation properties for a camera sensor.

```
<!-- Fixed joint connecting camera_link to parent -->
<joint name="camera_joint" type="fixed">
  <parent link="${parent}"/>
  <child link="camera_link"/>
  <origin xyz="0.100925 0.0 0.07670" rpy="0 0 0" />
</joint>

<link name="camera_link">
  <!-- Simple visual representation of the camera -->
  <visual>
    <geometry>
      <box size="0.01 0.01 0.01"/>
    </geometry>
    <material name="Red"/>
  </visual>
</link>

<joint name="camera_optical_joint" type="fixed">
    <parent link="camera_link"/>
    <child link="camera_link_optical"/>
    <origin xyz="0 0 0" rpy="${-M_PI_2} 0 ${-M_PI_2}"/>
</joint>

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

<!-- Gazebo simulation properties -->
<gazebo reference="camera_link">

    <material>Gazebo/Red</material>

      <sensor name="camera" type="camera">
          <camera>
          <horizontal_fov>1.047</horizontal_fov>
          <image>
              <width>320</width>
              <height>240</height>
          </image>
          <clip>
              <near>0.1</near>
              <far>100</far>
          </clip>
          </camera>
          <always_on>1</always_on>
          <update_rate>30</update_rate>
          <visualize>true</visualize>
          <topic>camera</topic>
      </sensor>
</gazebo>
```

Listing 3: camera xacro: camera_gazebo_macro.xacro

Then the ArUco marker was defined as a separate link (`arucotag_link`) positioned relative to the `obstacle_09`, with a visual representation including a texture (arucotag-115.png). A fixed joint (arucotag_joint) optionally attaches the marker to the main obstacle, ensuring it moves with the obstacle. The marker's dimensions and material properties are also specified.

```
<!-- ArUco Marker as a Separate Link -->
<link name="arucotag_link">
  <pose>-0.005 0.75 0.20 0 0 1.57</pose> <!-- Position relative
    ↪ to the obstacle's link -->

  <visual name="arucotag_visual">
    <geometry>
      <box>
        <size>0.2 0.005 0.2</size>
      </box>
    </geometry>
    <material>
      <diffuse>1 1 1 1</diffuse>
      <specular>0.4 0.4 0.4 1</specular>
      <pbr>
        <metal>
          <albedo_map>model://arucotag/arucotag-115.png</albedo_map>
        </metal>
      </pbr>
    </material>
  </visual>
</link>

<!-- Optionally, you could add a joint to attach the ArUco
   ↪ marker to the main obstacle -->
<joint name="arucotag_joint" type="fixed">
  <parent>link</parent>
  <child>arucotag_link</child>
  <pose>0 0 0 0 0 0</pose> <!-- This makes the ArUco marker
    ↪ part of the obstacle -->
</joint>
```

Listing 4: sdf: obstacle_09.sdf

Using the Nav2 Simple Commander API enable an autonomous navigation task

(**a**) Define 4 goals in a dedicated `.yaml` file. They must have the following poses with respect to the map frame:

- `Goal_1` $x = 0$ m, $y = 3$ m $yaw = 0$ deg;
- `Goal_2` $x = 6$ m, $y = 4$ m $yaw = 30$ deg;
- `Goal_3` $x = 7.0$ m, $y = -1.4$ m $yaw = 180$ deg;
- `Goal_4` $x = -1.6$ m, $y = -2.5$ m $yaw = 75$ deg;

In the `rl_fra2mo_description/config`, the `goal.yaml` file has been created to addresses all the pose to follow.

The `goal.yaml` file was created, to addresses all the waypoints that the differential drive robot must follow. Each point is expressed in a sequential order, containing both the goal position and orientation. The orientation is expressed as roll-pitch-yaw angles such to easily express the desired pose.

```
waypoints:
  - position:
      x: 6.5
```

```
      y: -1.4
      z: 0.1
    orientation:
      roll: 0.0
      pitch: 0.0
      yaw: 3.14
  - position:
      x: -1.6
      y: -2.5
      z: 0.1
    orientation:
      roll: 0.0
      pitch: 0.0
      yaw: 1.309
  - position:
      x: 6.0
      y: 4.0
      z: 0.1
    orientation:
      roll: 0.0
      pitch: 0.0
      yaw: 0.5236
  - position:
      x: 0.0
      y: 3.0
      z: 0.1
    orientation:
      roll: 0.0
      pitch: 0.0
      yaw: 0.0
```

Listing 5: yaml file: goal.yaml

(**b**) Modify `follow_waypoint.py` or `reach_goal.py` to send the defined goals to the mobile platform in a given order. Go to the next one once the robot has arrived at the current goal. The order of the explored goals must be `Goal_3` → `Goal_4` → `Goal_2` → `Goal_1`.

To modify the provided code to follow the specific order of goals, one need to ensure that the goal poses list contains the waypoints in the desired order. First, the `load_waypoints` function was implemented. It reads waypoints from a YAML file (`aruco_path.yaml`) that are defined into he world frame, represented by position and orientation data as rpy angles. (referred as 'map' frame into the assignment).

```
def load_waypoints(yaml_file):
    """Load waypoints from a YAML file."""
    with open(yaml_file, 'r') as file:
        return yaml.safe_load(file)
```

Listing 6: script file: follow_waypoint.py

The commander script acts on the differential drive robot referring to its map frame that has it's initial position as origin of the reference frame. Using the `world_to_map` function, the waypoints are transformed from the world frame to the map frame. The transformation can be described by the following relationship:

$$p^{map} = o^{map}_{world} + R^{map}_{world} P^{world}$$

where:

$$o^{map}_{world} = \begin{bmatrix} -3 \\ 3.5 \end{bmatrix}$$

and

$$R_{world}^{map} = \begin{bmatrix} cos(-\pi/2) & -sin(-\pi/2) \\ sin(-\pi/2) & cos(-\pi/2) \end{bmatrix}$$

This transformation involves translation and rotation defined by the variables translation and rotation.

```python
def world_to_map(world_pose, translation, rotation):
    """
    Transform a pose from the world frame to the map frame.

    Args:
        world_pose (PoseStamped): The pose in the world frame.
        translation (dict): The translation vector (x, y, z) from the world
            ↪ frame to the map frame.
        rotation (float): The rotation (in radians) from the world frame to the
            ↪ map frame.

    Returns:
        PoseStamped: The transformed pose in the map frame.
    """
    # Extract the position of the pose in the world frame
    px_world = world_pose.pose.position.x
    py_world = world_pose.pose.position.y
    pz_world = world_pose.pose.position.z

    # Apply the translation
    px_translated = px_world - translation["x"]
    py_translated = py_world - translation["y"]
    pz_translated = pz_world - translation.get("z", 0.0)  # Default z
        ↪ translation to 0 if not provided

    # Apply the rotation
    # Rotation is a simple 2D transformation on the (x, y) coordinates
    px_map = math.cos(-rotation) * px_translated - math.sin(-rotation) *
        ↪ py_translated
    py_map = math.sin(-rotation) * px_translated + math.cos(-rotation) *
        ↪ py_translated

    # Create the new pose in the map frame
    map_pose = PoseStamped()
    map_pose.header.frame_id = 'map'
    map_pose.header.stamp = world_pose.header.stamp
    map_pose.pose.position.x = px_map
    map_pose.pose.position.y = py_map
    map_pose.pose.position.z = pz_translated

    # Transform the orientation
    # Convert the world quaternion to RPY
    qx = world_pose.pose.orientation.x
    qy = world_pose.pose.orientation.y
    qz = world_pose.pose.orientation.z
    qw = world_pose.pose.orientation.w
    roll, pitch, yaw = euler_from_quaternion([qx, qy, qz, qw])

    # Apply the rotation to the yaw
    yaw_map = yaw + (-rotation)

    # Convert back to a quaternion
    quaternion_map = quaternion_from_euler(roll, pitch, yaw_map)
    map_pose.pose.orientation.x = quaternion_map[0]
    map_pose.pose.orientation.y = quaternion_map[1]
    map_pose.pose.orientation.z = quaternion_map[2]
    map_pose.pose.orientation.w = quaternion_map[3]

    return map_pose
```

Listing 7: script file: follow_waypoint.py

Then, into the main function, the robot's initial pose is set using `navigator.setInitialPose`. This allows the navigation system to align the robot's starting position with the map. The list of waypoints (transformed to the map frame) is passed to the `navigator.followWaypoints()`

method. The robot follows the list in the provided order, with the feedback loop ensuring the robot tracks progress at each waypoint.

```
def main():
    # Initialize ROS 2 node
    rclpy.init()
    navigator = BasicNavigator()

    # Set our demo's initial pose
    initial_pose = PoseStamped()
    initial_pose.header.frame_id = 'map'
    initial_pose.header.stamp = navigator.get_clock().now().to_msg()
    initial_pose.pose.position.x = 0.0
    initial_pose.pose.position.y = 0.0

    # Orientation in RPY (roll, pitch, yaw)
    roll = 0.0   # Roll angle in radians
    pitch = 0.0  # Pitch angle in radians
    yaw = 0.0    # Yaw angle in radians
    # Convert RPY to Quaternion
    quaternion = rpy_to_quaternion(roll, pitch, yaw)

    # Assign Quaternion to PoseStamped
    initial_pose.pose.orientation.x = quaternion["x"]
    initial_pose.pose.orientation.y = quaternion["y"]
    initial_pose.pose.orientation.z = quaternion["z"]
    initial_pose.pose.orientation.w = quaternion["w"]
    navigator.setInitialPose(initial_pose)

    # Get the package share directory and load the YAML file
    package_share_directory =
        ↪ get_package_share_directory('rl_fra2mo_description')
    yaml_file = os.path.join(package_share_directory, 'config', 'goal.yaml')
    waypoints = load_waypoints(yaml_file)

    def create_pose(transform):

        # Transformation from world frame to map frame (only for slam)
        translation = {"x": -3.0, "y": 3.5}
        rotation = -math.pi / 2  # 90 degrees

        pose = PoseStamped()
        pose.header.frame_id = 'world'
        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"]

        # Convert RPY to quaternion
        quaternion = rpy_to_quaternion(
            transform["orientation"]["roll"],
            transform["orientation"]["pitch"],
            transform["orientation"]["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"]

        map_pose = world_to_map(pose, translation, rotation)

        return map_pose

    # Convert loaded waypoints into PoseStamped messages
    goal_poses = list(map(create_pose, waypoints["waypoints"]))

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

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

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

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

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

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

            current_waypoint = feedback.current_waypoint + 1
            current_goal = goal_poses[feedback.current_waypoint]
            print_goal(current_goal, current_waypoint)

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

Listing 8: script file: follow_waypoint.py

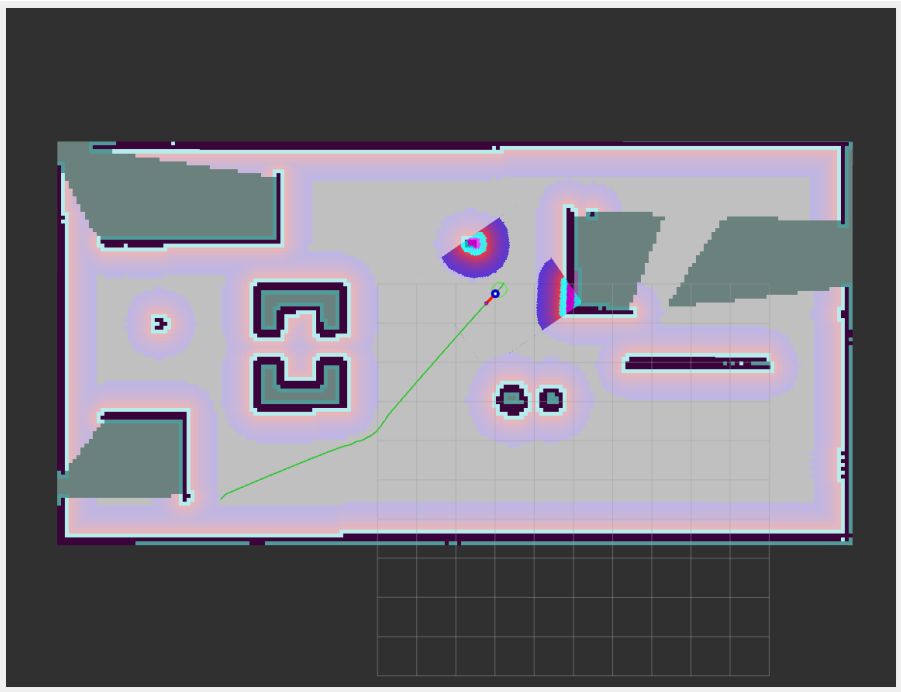


Figure 1: mobile robot on rviz following waypoints

(**c**) Record a bagfile of the executed robot trajectory and plot it in the XY plane.

To record a bagfile of the executed trajectory the following command must be used:

```
$ ros2 bag record /pose
```

It records the topic `/pose`. This message type convey different levels of detail about the robot's pose:

– **PoseStamped**
  This contains the robot's position (x, y, z) and orientation (as a quaternion) in a specific coordinate frame (map frame in this case), along with a timestamp.
– **PoseWithCovarianceStmped**
  This is an extended version of PoseStamped, which adds covariance information. Covariance represents the uncertainty in the robot's pose.

Using plotjuggler it is possible to plot the XY trajectory of the mobile robot with respect to the map frame: Finally by, using the custom timeseries functionality on plotjuggler, it is possible to

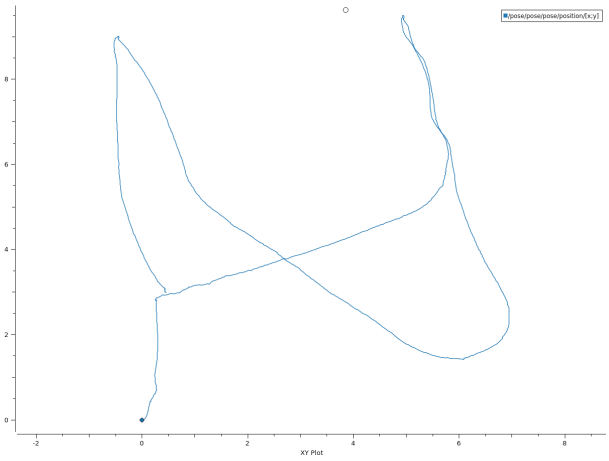


Figure 2: XY graph of the mobile robot in map frame using `goals.yaml`

transform the odometry data from the map frame to the world frame by apllying the following tranformation:

$$p_x^{world} = p_y^{map} + offset_x$$
$$p_y^{world} = -p_x^{map} + offset_x$$

obtaining the following xy plot:

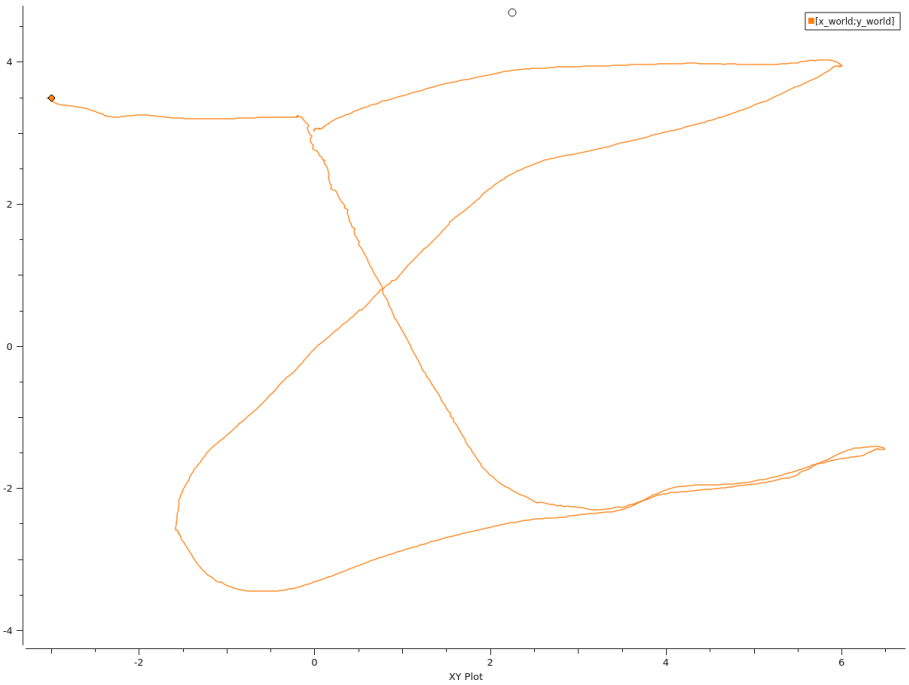


Figure 3: XY graph of the mobile robot in world frame using `goals.yaml`

Map the environment tuning the navigation stack's parameters

(**a**) Modify, add or remove the previous goals to get a complete map of the environment and save it (put in the report the .png of the map).

To map the whole environment, a new yaml file was created (`mapping.yaml`) by adding and changing some of the previous waypoints. The new map was saved by running in the maps folder

```
$ ros2 run nav2\_map\_server map\_saver\_cli -f new_map.
```

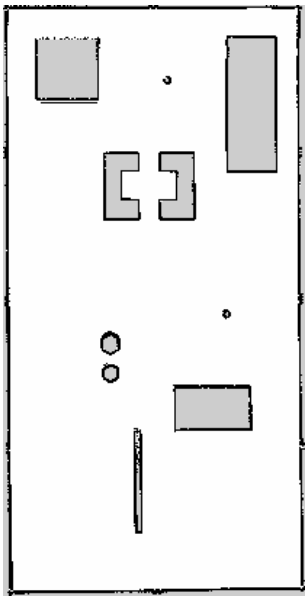


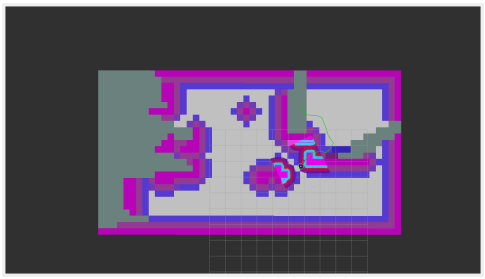
Figure 4: `new_map.png`

(**b**) Change the parameters of the navigation config (try at least 4 different configurations). The suggested parameters that you can change are:

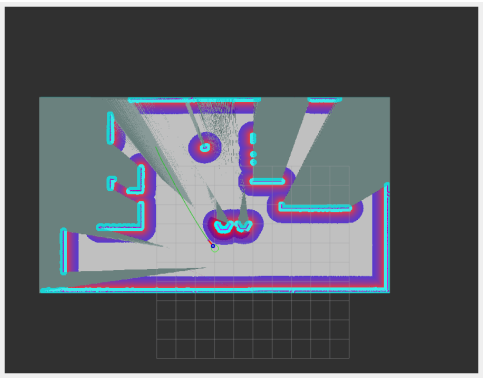
- In file `slam.yaml`: tune parameters `minimum_travel_distance`, `minimum_travel_heading`, `resolution` and `transform_publish_period`.
- In file `explore.yaml`: change the `inflation_radius` and `cost_scaling_factor` for global and local costmaps.

Below an explanation of the above parameters:

- `minimum_travel_distance` Specifies the minimum distance the robot must travel before triggering a SLAM update.
  - Smaller values increase map accuracy but require more computational resources.
  - Larger values are suited for open spaces with fewer features.
- `minimum_travel_heading` Sets the minimum angular change required to update the map.
  - Lower values capture more rotational changes, which is ideal for navigating in tight environments.
  - Higher values are suitable for open spaces to reduce computational load.
- `resolution` Defines the size of each grid cell in the map.
  - Finer resolutions (e.g., 0.01m) produce more detailed maps but are computationally intensive.
  - Coarser resolutions (e.g., 0.1m) are efficient and ideal for larger areas with fewer details.
- `transform_publish_period` Controls how frequently coordinate transforms (e.g., `map` → `odom`) are updated.
  - Higher frequencies are essential for dynamic environments to ensure accurate localization.



(a) Simulation with resolution = 0.4

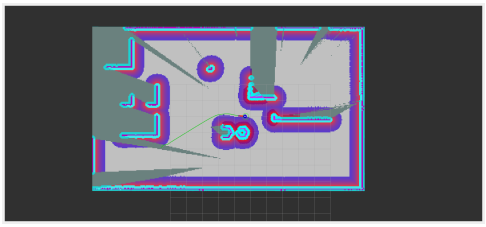


(b) Simulation with resolution = 0.02

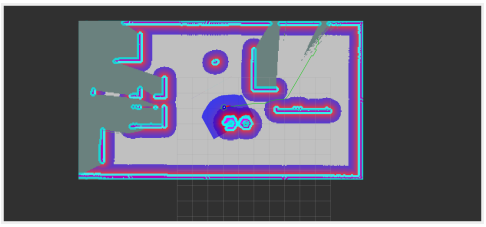
Table 1: Comparison of simulations with different resolutions.

    - Lower frequencies reduce CPU usage in static environments.

- `inflation_radius` Defines the buffer zone around obstacles where navigation costs are increased.
    - Larger values ensure safer navigation by keeping the robot farther from obstacles.
    - Smaller values allow closer navigation to obstacles but increase the risk of collisions.



(a) Simulation with inflation radius = 0.4



(b) Simulation with inflation radius = 1.5

Table 2: Comparison of simulations with different inflation radius.

- `cost_scaling_factor` Determines how steeply the navigation cost increases near obstacles.
    - Higher values make the robot avoid obstacles more aggressively.
    - Lower values allow the robot to navigate closer to obstacles, which may be necessary in tight spaces.

Four different configurations have been considered interesting and are explained below:

- Coarser Resolution for larger areas:
    - `minimum_travel_distance` = 1.0
    - `minimum_travel_heading` = 0.3
    - `resolution` = 0.15
    - `transform_publish_period` = 0.25
    - `inflation_radius` = 1.2
    - `cost_scaling_factor` = 2.5
- Fine Resolution and Conservative Navigation:
    - `minimum_travel_distance` = 0.02
    - `minimum_travel_heading` = 0.02
    - `resolution` = 0.02
    - `transform_publish_period` = 0.08
    - `inflation_radius` = 0.6
    - `cost_scaling_factor` = 4.0

- High-Frequency Updates for Dynamic Environments:
  - `minimum_travel_distance` = 0.2
  - `minimum_travel_heading` = 0.1
  - `resolution` = 0.04
  - `transform_publish_period` = 0.01
  - `inflation_radius` = 0.5
  - `cost_scaling_factor` = 1.5
- Aggressive Navigation for Tight Spaces:
  - `minimum_travel_distance` = 0.03
  - `minimum_travel_heading` = 0.03
  - `resolution` = 0.02
  - `transform_publish_period` = 0.05
  - `inflation_radius` = 0.25
  - `cost_scaling_factor` = 1.0

(**c**) Comment on the results you get in terms of robot trajectories, execution timings, map accuracy, etc.

1. **Coarser Resolution for Larger Areas:**
   - These parameters are tuned for **efficient mapping and navigation in large-scale areas**. The chosen values prioritize computational efficiency and robustness in environments where:
     - Precision mapping is less critical.
     - The robot has ample room to maneuver.
   - **Key scenarios:**
     - **Large open spaces:** Ideal for environments such as warehouses, open fields, or large halls where high precision is unnecessary.
     - **Robust navigation:** Suitable for robots that prioritize speed and efficiency over fine detail.
   - **Advantages:**
     - **High computational efficiency:** These parameters reduce the burden on processing hardware, making them ideal for robots with limited resources.
     - **Robustness in navigation:** The robot can handle large-scale environments without frequent map updates.
   - **Trade-offs:**
     - **Reduced precision:** Coarser resolution may result in less detailed maps, which can be a limitation in cluttered or dynamic spaces.
     - **Lower adaptability:** Slower update rates make the robot less responsive to rapid changes in the environment.
   - Overall, this configuration is effective for large-scale areas where computational efficiency and robustness are more critical than fine-grained precision.
2. **Fine Resolution and Conservative Navigation:**
   - The fine resolution and conservative navigation parameters are particularly well-suited for the following scenarios:
     - **Tight and confined spaces:** These settings allow for precise navigation, reducing the risk of collisions in narrow or cluttered areas.
     - **Dynamic environments:** Frequent updates enable the robot to capture changes in the surroundings, ensuring responsive navigation in rapidly evolving conditions.
     - **Sensitive operations:** The conservative navigation approach is ideal for handling fragile items or operating near humans, where cautious and deliberate movement is necessary.
   - While these parameters provide high accuracy and safety, they come with certain trade-offs:

    - **Increased computational demand:** The fine resolution and frequent updates require more processing power, which may not be feasible on limited hardware.
    - **Slower navigation in open areas:** In large, open environments where precision is less critical, these settings can unnecessarily slow down the robot's progress.
  - Overall, these settings prioritize safety and precision, making them ideal for detailed and cautious navigation but less efficient in scenarios where speed is a priority.

3. **High-Frequency Updates for Dynamic Environments:**
  - This configuration is designed for **fast-changing and dynamic environments**, where the robot must react quickly to alterations in the surroundings, such as moving obstacles or frequent changes in the map. The parameters prioritize responsiveness and adaptability.
  - **Key scenarios:**
    - **Dynamic environments:** Ideal for areas with moving obstacles or frequently changing layouts, such as warehouses or public spaces.
    - **Fast-moving tasks:** Suitable for robots operating at higher speeds where timely updates are crucial to avoid collisions.
  - **Advantages:**
    - **High responsiveness:** Frequent updates in the transform and costmaps ensure the robot is always aware of its surroundings.
    - **Improved obstacle avoidance:** The lower inflation radius and cost-scaling factor enable the robot to plan paths efficiently while maintaining a reasonable safety margin.
  - **Trade-offs:**
    - **Higher computational load:** Frequent updates require significant processing resources, which can strain the robot's hardware.
    - **Reduced mapping precision:** The slightly coarser resolution may result in less detailed maps, which could impact tasks requiring fine-grained navigation.
  - This configuration strikes a balance between speed and safety, making it suitable for fast-paced, dynamic environments but less effective in highly detailed mapping scenarios.

4. **Aggressive Navigation for Tight Spaces:**
  - This configuration is optimized for **tight and highly constrained environments**, where the robot needs to make sharp maneuvers and navigate around small obstacles efficiently. The settings prioritize agility and compact navigation footprints.
  - **Key scenarios:**
    - **Crowded or cluttered areas:** Perfect for environments like small indoor spaces, factories, or cluttered storage areas where precision and maneuverability are critical.
    - **Robots with tight turning radii:** These settings allow robots with limited mobility to operate effectively in confined spaces.
  - **Advantages:**
    - **Compact navigation:** The low inflation radius and cost-scaling factor allow the robot to plan paths very close to obstacles, maximizing the use of available space.
    - **Quick reactions:** Moderate transform update rates and a small minimum travel distance enable the robot to adjust rapidly to minor changes in the environment.
  - **Trade-offs:**
    - **Higher collision risk:** The aggressive navigation settings reduce the safety margin, increasing the risk of collisions if the robot or environment is highly unpredictable.
    - **Lower suitability for open areas:** In large, open spaces, the robot may exhibit unnecessary sharp maneuvers, leading to inefficient paths.
  - This configuration is ideal for scenarios where space is at a premium and the robot needs to navigate nimbly. However, it is not suitable for wide-open environments where smoother, larger-scale maneuvers would be more efficient.

In conclusion, after testing each of these configurations on the mobile robot, the High-Frequency Updates for Dynamic Environments proved to be the most effective setup. This configuration demonstrated superior performance across key metrics, including trajectory execution, execution

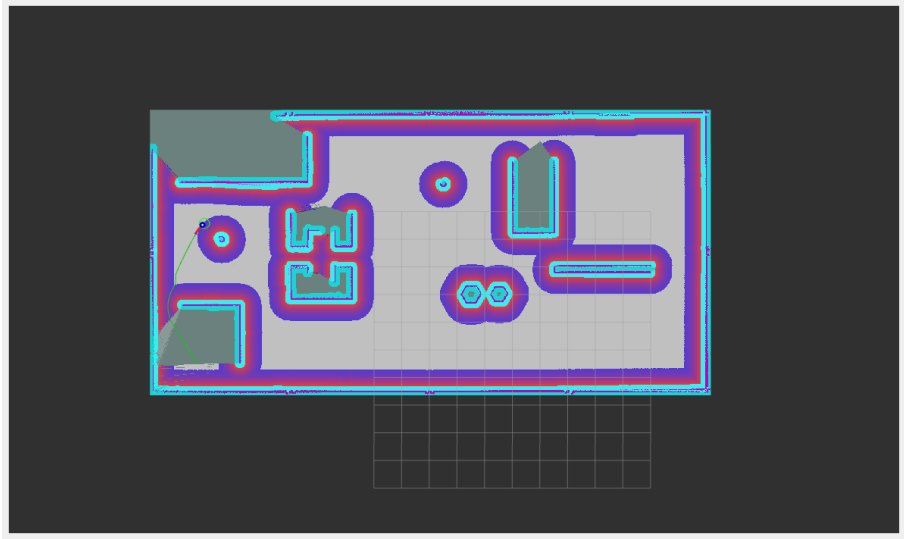


Figure 5: Dynamic configuration

time and map accuracy. These results highlight the importance of responsiveness in dynamic environments, where the ability to adapt quickly to changes can greatly enhance overall navigation performance. While other configurations, such as Coarser Resolution for Larger Areas or Aggressive Navigation for Tight Spaces, may be advantageous in specific scenarios, the High-Frequency Updates for Dynamic Environments offered the best balance between speed, precision, and reliability in diverse and unpredictable conditions.

Vision-based navigation of the mobile platform

(**a**) Create a launch file running both the navigation and the `aruco_ros` node using the robot camera you previously added to the robot model.

As one can see, below there's the code used in `vision_navigation.launch.py`. Besides the gazebo simulation, it launches navigation and slam by using the `fra2mo_explore.launch.py` previously developed, while also starting the `aruco_ros` nodes to detect markers using the robot's camera.

```
# Include gazebo_fra2mo.launch file
gazebo_launch = IncludeLaunchDescription(
    PythonLaunchDescriptionSource(
        PathJoinSubstitution([fra2mo_dir, 'launch', 'gazebo_fra2mo.launch.py'])
    ),
    launch_arguments={'use_sim_time': use_sim_time}.items()
)

# Include fra2mo_explore.launch file
explore_launch = IncludeLaunchDescription(
    PythonLaunchDescriptionSource(
        PathJoinSubstitution([fra2mo_dir, 'launch', 'fra2mo_explore.launch.py'])
    ),
    launch_arguments={'use_sim_time': use_sim_time}.items()
)

# Include aruco_ros single.launch.py
aruco_ros_node = IncludeLaunchDescription(
    PythonLaunchDescriptionSource(
        PathJoinSubstitution([aruco_ros_dir, 'launch', 'single.launch.py'])
    ),
    launch_arguments={
        'marker_size': '0.1',  # Marker size in meters
        'marker_id': '115',    # Marker ID to detect
        'use_sim_time': use_sim_time
    }.items()
)
```

Listing 9: launch file: vision_navigation.launch.py

(**b**) implement a 2D navigation task following this logic

- Send the robot in proximity of obstacle 9.
- Make the robot look for the ArUco marker. Once detected, retrieve its pose with respect to the map frame.
- Return the robot to the initial position.

To fulfill the requirements a ROS2 Python script was written taking inspiration from the `follow_waypoints.py` commander script. The code defines an `ArucoNavigator` class that integrates Aruco marker pose detection with robot navigation. The robot uses the `ArucoNavigator` class from the `nav2_simple_commander` to navigate to predefined positions while detecting Aruco markers.

1. **Class Initialization (`__init__`)**
   The `ArucoNavigator` class is initialized with:
   - A `BasicNavigator` instance to handle robot navigation.
   - A subscription to the `/aruco_single/pose` topic where the Aruco marker's pose is published.
   - Translation and rotation parameters to convert poses from the world frame to the map frame, to allow the commander api to act on the differential drive robot correctly.

```
super().__init__('aruco_navigator')

# Initialize the navigator
self.navigator = BasicNavigator()
self.aruco_pose = None

# Aruco pose subscriber
self.create_subscription(
    PoseStamped,
    '/aruco_single/pose',  # Topic where Aruco pose is published
    self.aruco_pose_callback,
    10
)

# Map transformation parameters (world frame to map frame)
self.translation = {"x": -3.0, "y": 3.5}
self.rotation = -math.pi / 2  # -90 degrees
```

Listing 10: script: aruco_navigation_task.py

2. **Pose Transformation (`transform_to_map_frame`)**
   The `transform_to_map_frame` method takes the position and orientation of the Aruco marker in the world frame and converts it to the map frame. The transformation involves applying a translation and a 2D rotation. The transformed pose is returned as a `PoseStamped` message. The transformation function is written as:

```
def transform_to_map_frame(self, position, orientation):
    # Unpack inputs
    px_world, py_world, pz_world = position
    roll_world, pitch_world, yaw_world = orientation

    # Apply translation
    px_translated = px_world - self.translation["x"]
    py_translated = py_world - self.translation["y"]

    # Apply rotation (2D transformation)
    px_map = math.cos(-self.rotation) * px_translated -
        ↪ math.sin(-self.rotation) * py_translated
    py_map = math.sin(-self.rotation) * px_translated +
        ↪ math.cos(-self.rotation) * py_translated
```

```
    # Transform orientation
    yaw_map = yaw_world + (-self.rotation)
    q_map = quaternion_from_euler(roll_world, pitch_world, yaw_map)

    # Create the PoseStamped
    map_pose = PoseStamped()
    map_pose.header.frame_id = 'map'
    map_pose.header.stamp = self.get_clock().now().to_msg()
    map_pose.pose.position.x = px_map
    map_pose.pose.position.y = py_map
    map_pose.pose.position.z = pz_world
    map_pose.pose.orientation.x = q_map[0]
    map_pose.pose.orientation.y = q_map[1]
    map_pose.pose.orientation.z = q_map[2]
    map_pose.pose.orientation.w = q_map[3]
    return map_pose
```

Listing 11: script: aruco_navigation_task.py

3. **Aruco Pose Callback** (`aruco_pose_callback`)
   This callback function is triggered when an Aruco marker pose is detected. The position and orientation of the Aruco marker are extracted from the message, and the orientation is converted from quaternion to Euler angles. The transformed pose in the map frame is logged.
   The callback is as follows:

```
def aruco_pose_callback(self, msg):
    """ Callback when the Aruco marker is detected. """
    self.get_logger().info("Aruco marker detected!")
    self.aruco_pose = msg

    # Extract the position and orientation from the Aruco marker pose
    position = (msg.pose.position.x, msg.pose.position.y,
        ↪ msg.pose.position.z)
    orientation_quat = (
        msg.pose.orientation.x,
        msg.pose.orientation.y,
        msg.pose.orientation.z,
        msg.pose.orientation.w
    )

    # Convert quaternion to roll, pitch, yaw
    roll, pitch, yaw = euler_from_quaternion(orientation_quat)
    orientation = (roll, pitch, yaw)

    transformed_pose = self.transform_to_map_frame(position, orientation)

    # Print the transformed pose in the map frame
    self.get_logger().info(
        f"Aruco marker pose in map frame: \n"
        f"Position: x = {transformed_pose.pose.position.x:.3f}, "
        f"y = {transformed_pose.pose.position.y:.3f}, "
        f"z = {transformed_pose.pose.position.z:.3f}\n"
        f"Orientation: x = {transformed_pose.pose.orientation.x:.3f}, "
        f"y = {transformed_pose.pose.orientation.y:.3f}, "
        f"z = {transformed_pose.pose.orientation.z:.3f}, "
        f"w = {transformed_pose.pose.orientation.w:.3f}"
    )
```

Listing 12: script: aruco_navigation_task.py

4. **Robot Navigation** (`navigate_to_position`)
   The `navigate_to_position` method uses the `BasicNavigator` class to send the robot to a target position. It waits for the task to complete and logs the remaining distance while navigating.
   Here is the navigation function:

```
def navigate_to_position(self, pose_stamped):
    """ Send a navigation goal and wait for completion. """
    self.navigator.goToPose(pose_stamped)
```

```
    while not self.navigator.isTaskComplete():
        feedback = self.navigator.getFeedback()
        if feedback:
            self.get_logger().info(f"Distance remaining:
                ↪ {feedback.distance_remaining:.2f} meters")
```

Listing 13: script: aruco_navigation_task.py

5. **Main Task (`main`)**
   The main function orchestrates the robot's navigation and Aruco marker detection. It defines starting and goal positions, transforms them into the map frame, and then sends the robot to the goal position. Once the robot reaches the goal, it waits for the Aruco marker to be detected, logs the transformed pose, and finally returns to the start position.
   The main function is as follows:

```
def main(self):
    """ Main task: navigate, detect Aruco, and return. """
    self.navigator.waitUntilNav2Active(localizer="smoother_server")

    # Define start and goal positions in the world frame
    start_position_world = (-3.0, 3.5, 0.1)
    start_orientation_world = (0.0, 0.0, 1.57)  # Yaw = 90 degrees

    final_position_world = (-3.87, -4.02, 0.1)
    final_orientation_world = (0.0, 0.0, 1.54)  # Yaw slightly different

    # Transform start and final positions to the map frame
    start_pose = self.transform_to_map_frame(start_position_world,
        ↪ start_orientation_world)
    final_pose = self.transform_to_map_frame(final_position_world,
        ↪ final_orientation_world)

    # Navigate to goal
    self.get_logger().info("Navigating to obstacle 9...")
    self.navigate_to_position(final_pose)

    # Wait for Aruco marker detection
    self.get_logger().info("Looking for Aruco marker...")
    while not self.aruco_pose:
        rclpy.spin_once(self, timeout_sec=0.5)

    # Log Aruco pose in the map frame
    self.get_logger().info("Aruco marker detected")

    # Return to start position
    self.get_logger().info("Returning to start position...")
    self.navigate_to_position(start_pose)

    self.get_logger().info("Task complete. Shutting down.")
```

Listing 14: script: aruco_navigation_task.py

(**c**) Publish the Aruco pose as TF To publish the detected pose of the aruco marker and publish it as a TF the `aruco_pose_tf.cpp` node was developed. It uses transformation between the camera frame and the world to publish the aruco pose in the world frame.

1. **Handling the ArUco Pose**: The callback method `handleArucoPose` first checks if the pose has already been stabilized to avoid processing the same pose multiple times. Then, it looks up the required transforms between various coordinate frames (camera to base link, base link to odom, odom to map, etc.) using the `tf_buffer_`.

```
// Transform the pose through all frames
geometry_msgs::msg::PoseStamped pose_in_camera_link;
tf2::doTransform(*msg, pose_in_camera_link,
    ↪ transform_camera_optical_to_camera);
```

Figure 6: aruco task in rviz

```
geometry_msgs::msg::PoseStamped pose_in_base_link;
tf2::doTransform(pose_in_camera_link, pose_in_base_link,
    ↪ transform_camera_to_base_link);

geometry_msgs::msg::PoseStamped pose_in_base_footprint;
tf2::doTransform(pose_in_base_link, pose_in_base_footprint,
    ↪ transform_base_link_to_base_footprint);

geometry_msgs::msg::PoseStamped pose_in_odom;
tf2::doTransform(pose_in_base_footprint, pose_in_odom,
    ↪ transform_base_footprint_to_odom);

tf2::doTransform(pose_in_odom, transformed_pose, transform_odom_to_map);
```

Listing 15: script: aruco_pose_tf.cpp

2. **Applying Fixed Map-to-World Transformation**: The node applies a fixed transformation from the map frame to the world frame, including translation and rotation. This is done in the `applyMapToWorldTransform` method:

```
void applyMapToWorldTransform(const geometry_msgs::msg::PoseStamped
    ↪ &pose_in_map,
                              geometry_msgs::msg::PoseStamped
                                  ↪ &pose_in_world)
{
    // Fixed transformation: map to world
    double x_map = pose_in_map.pose.position.x;
    double y_map = pose_in_map.pose.position.y;
    double yaw_map = -M_PI_2;  // -pi/2 rad (90 degrees clockwise)

    // Translation offset
    double x_offset = -3.0;
    double y_offset = 3.5;

    // Apply rotation (yaw) from map frame to world frame
    pose_in_world.pose.position.x = x_offset + (x_map * cos(yaw_map) -
        ↪ y_map * sin(yaw_map));
    pose_in_world.pose.position.y = y_offset + (x_map * sin(yaw_map) +
        ↪ y_map * cos(yaw_map));

    // Keep the z value the same as in the map frame
    pose_in_world.pose.position.z = pose_in_map.pose.position.z;

    // Orientation: No change in orientation since it's the same for both
        ↪ frames
    pose_in_world.pose.orientation = pose_in_map.pose.orientation;
}
```

Listing 16: script: aruco_pose_tf.cpp

3. **Publishing the Transformed Pose as Static Transform**: Once the pose is transformed into the world frame, the node creates a `TransformStamped` message and broadcasts the transformed pose as a static transform using the `tf_broadcaster_`. This is achieved using the `StaticTransformBroadcaster` which broadcasts the pose of the ArUco marker in the world frame.

```
// Broadcast the static transform (tf_static) for ArUco marker in world
    ↪ frame
geometry_msgs::msg::TransformStamped tf_stamped;
tf_stamped.header.stamp = this->now();
tf_stamped.header.frame_id = world_frame_;
tf_stamped.child_frame_id = "aruco_marker";

tf_stamped.transform.translation.x = saved_pose_.pose.position.x;
tf_stamped.transform.translation.y = saved_pose_.pose.position.y;
tf_stamped.transform.translation.z = saved_pose_.pose.position.z;

tf_stamped.transform.rotation.x = saved_pose_.pose.orientation.x;
tf_stamped.transform.rotation.y = saved_pose_.pose.orientation.y;
tf_stamped.transform.rotation.z = saved_pose_.pose.orientation.z;
tf_stamped.transform.rotation.w = saved_pose_.pose.orientation.w;

// Publish static transform
tf_broadcaster_->sendTransform(tf_stamped);
```

Listing 17: script: aruco_pose_tf.cpp

By running the `aruco_pose_tf` node it is finally possible to retrieve the marker pose in the world frame:

```
[INFO] [1734950287.816010582] [aruco_pose_tf]: Aruco TF Publisher node started.
[INFO] [1734950326.578582634] [aruco_pose_tf]: Aruco pose transformed to world frame:
[INFO] [1734950326.578628279] [aruco_pose_tf]: Position (x, y, z): (-3.781351, -3.658261, 0.168813)
[INFO] [1734950326.578639108] [aruco_pose_tf]: Orientation (x, y, z, w): (0.502081, 0.491458, 0.496303, 0
.509968)
[INFO] [1734950326.578765630] [aruco_pose_tf]: Published static transform: [frame: world -> aruco_marker]
[INFO] [1734950326.578779214] [aruco_pose_tf]: Translation:
    x : -3.78
    y : -3.66
    z : 0.17
Orientation:
    x : 0.50
    y : 0.49
    z : 0.50
    w : 0.51
```

Figure 7: marker detected pose