Management Center Innsbruck

Department of Technology & Life Sciences

Master's program Mechatronics & Smart Technologies

Report

composed as part of the course
WS 2024 Handling Technology (MECH-M-3-HHT-HHT-ILV)

about

Aruco Marker Detection in Simulated ROS2 Systems

from

Liam Nolan

Study program	Master's program Mechatronics & Smart Technologies
Year	MA-MECH-23-VZ
Course	WS 2024 Handling Technology (MECH-M-3-HHT-HHT-ILV)
Name of supervisor	Prof. Dr.-Ing. Sebastian Repetzki
Submission deadline	March 02, 2025

February 23, 2025

Contents

Chapter 1

Introduction and Background

Autonomous robot docking is a critical capability for mobile robots, enabling tasks such as automated charging, precise navigation, and object interaction. This project focuses on developing and simulating an **Aruco marker-based docking system for the TurtleBot4** using the **Gazebo Ignition simulator**. The system estimates the position of a target docking station using Aruco markers and subsequently sends navigation commands to a simulated **TurtleBot4** for accurate docking.

The project leverages **OpenCV** for Aruco marker detection, **ROS 2 Nav2** for path planning, and **RViz** for visualization. Aruco markers provide a reliable and computationally efficient method for visual localization, allowing the robot to determine the docking target's pose within its environment. The detected pose is then transformed into the correct frame of reference using **TF2**, ensuring proper navigation commands are executed.

To accomplish docking, the **Nav2 stack** is utilized for real-time path planning and obstacle avoidance. The TurtleBot4 first constructs a **SLAM**-based map of its environment before autonomously planning and executing the docking maneuver. The entire system is implemented in **simulation** due to hardware limitations and as an academic exercise in learning **Gazebo Ignition**, an advanced robotics simulation tool.

Simulating this docking system is valuable for testing navigation algorithms in a controlled environment, mitigating risks associated with real-world hardware failures. Furthermore, understanding **frame transformations**, **perception-based localization**, and **robot control** in simulation lays a strong foundation for real-world autonomous robot applications. The insights gained from this project contribute to research in robotic navigation, automated docking, and the integration of visual perception with autonomous control systems.

Chapter 2

Methodology and Theory

2.1 Gazebo Ignition

Gazebo Ignition is an advanced simulation environment designed for robotic applications, providing realistic physics, sensor data, and a configurable world model. It allows developers to test algorithms and robot behaviors in a controlled setting before deploying them on actual hardware.

For this project, the existing TurtleBot4 simulation provided by **Clearpath Robotics** was utilized as the base environment. This simulation includes a pre-configured TurtleBot4 model with its sensors and motion controllers, making it suitable for testing autonomous docking using Aruco markers.

2.1.1 Modifications to Include the Aruco Marker

To implement docking functionality, the simulation was modified to include an `marker.dae` file representing the docking target. Gazebo supports 3D models in STL and DAE formats, which can be referenced in the world's `SDF` (Simulation Description Format) file.

The Aruco marker was integrated by adding the following entry to the `warehouse.sdf` world file:

```
    <include>
      <uri>model://marker0</uri>
      <name>aruco_marker</name>
      <pose>1 0 0 0 0 0</pose>
    </include>
```

Listing 2.1. Adding the Aruco marker to the warehouse.sdf world file

This modification ensures that the Aruco marker is included in the environment as a static object, allowing the TurtleBot4 to detect it and navigate accordingly during docking operations.

2.1.2 Launching the Simulation

The simulation, including the modified world with the Aruco marker, was launched using the following ROS 2 command:

```
ros2 launch turtlebot4_ignition_bringup turtlebot4_ignition.launch.
    py slam:=true nav2:=true rviz:=true localization:=true
```

Listing 2.2. Launching the TurtleBot4 simulation with Aruco marker

This command enables the following key components:

- **SLAM**: Builds a map of the environment while the robot navigates.
- **Nav2**: Implements path planning and navigation.
- **RViz**: Provides a visualization of the robot's state and sensor data.
- **Localization**: Ensures that the robot can determine its position relative to the environment.

By integrating these components within Gazebo Ignition, the TurtleBot4 can autonomously navigate to the detected Aruco marker and execute docking maneuvers based on real-time vision-based localization.

2.2 Aruco Markers and Their Theory

Aruco markers are a type of fiducial marker used in computer vision applications to provide an easy and efficient method for pose estimation and localization. These markers are widely utilized in robotics, augmented reality (AR), and autonomous navigation due to their high detection accuracy and robustness.

2.2.1 Structure of Aruco Markers

An Aruco marker is a square grid containing a unique binary pattern. The marker consists of:

- A black border that helps with detection and segmentation.
- An inner grid of black and white pixels encoding a unique identifier.
- A predefined dictionary that holds multiple marker patterns for identification.

Each marker is part of a predefined dictionary, such as `DICT_5X5_250`, which contains 250 unique markers, each with a 5x5 internal grid. OpenCV provides several Aruco dictionaries, allowing users to select a marker set that balances uniqueness and detection speed.

2.3 Aruco Marker Detection

The TurtleBot4 publishes a simulated camera feed to the topic `/oakd/rgb/preview/image_raw`. This topic contains raw image data from the robot's RGB camera, which is used for

Aruco marker detection. A custom ROS 2 node subscribes to this topic, processes the image, detects Aruco markers using OpenCV, and visualizes the detected markers.

2.3.1 Subscribing to the Camera Topic

To access the camera feed, the ROS 2 node subscribes to the image topic and converts the incoming ROS Image message to an OpenCV-compatible format using `cv_bridge`:

```
from sensor_msgs.msg import Image
from cv_bridge import CvBridge

class ArucoNode(Node):
    def __init__(self):
        super().__init__('aruco_node')
        self.bridge = CvBridge()
        self.subscription = self.create_subscription(
            Image,
            '/oakd/rgb/preview/image_raw',
            self.listener_callback,
            10)

    def listener_callback(self, data):
        self.get_logger().info('Receiving␣video␣frame')
        current_frame = self.bridge.imgmsg_to_cv2(data)
```

Listing 2.3. Subscribing to the camera topic

2.3.2 Detecting Aruco Markers

The node uses OpenCV's Aruco module to detect markers using the `DICT_5X5_250` dictionary. The detection function finds markers and extracts their corner points and IDs:

```
import cv2
import numpy as np

ARUCO_DICT = {
    "DICT_5X5_250": cv2.aruco.DICT_5X5_250
}

self.this_aruco_dictionary = cv2.aruco.getPredefinedDictionary(
    ARUCO_DICT["DICT_5X5_250"])
self.this_aruco_parameters = cv2.aruco.DetectorParameters()

corners, marker_ids, _ = cv2.aruco.detectMarkers(
    current_frame, self.this_aruco_dictionary, parameters=self.
        this_aruco_parameters)
```

Listing 2.4. Detecting Aruco markers

2.3.3 Visualizing Detected Markers

Once markers are detected, they are drawn onto the camera image using OpenCV functions to highlight their position and orientation:

```python
if marker_ids is not None:
    cv2.aruco.drawDetectedMarkers(current_frame, corners,
        marker_ids)
    rvecs, tvecs, _ = cv2.aruco.estimatePoseSingleMarkers(
        corners, self.aruco_marker_side_length, self.mtx, self.dst)

    for i in range(len(marker_ids)):
        cv2.drawFrameAxes(current_frame, self.mtx, self.dst, rvecs[
            i], tvecs[i], 0.05)

cv2.imshow("camera", current_frame)
cv2.waitKey(1)
```

Listing 2.5. Drawing detected Aruco markers

This ensures that the detected Aruco markers are visually represented on the live camera feed, aiding in debugging and system validation.

2.4 Pose Estimation and Frame Transformation

Pose estimation and frame transformation are essential components of Aruco marker-based navigation. The system determines the marker's position and orientation relative to the camera and then transforms this pose into a global reference frame (e.g., the map frame) using coordinate transformations.

2.4.1 Pose Estimation Using Aruco Markers

Pose estimation is the process of determining the position (x, y, z) and orientation (θ, ϕ, ψ) of an object in 3D space. This is achieved using the Perspective-n-Point (PnP) algorithm, which solves for the camera pose given 2D-3D correspondences.

The transformation from the 3D world coordinate P_w to the 2D image coordinate p_i is given by:

$$p_i = K[R|t]P_w \tag{2.1}$$

where:

- K is the camera intrinsic matrix,
- R is the rotation matrix,
- t is the translation vector.

In the code, this is implemented using OpenCV's `solvePnP()` function:

```python
rvecs, tvecs, _ = cv2.aruco.estimatePoseSingleMarkers(
    corners, self.aruco_marker_side_length, self.mtx, self.dst)
```

Listing 2.6. Pose estimation using solvePnP

Here, `rvecs` (rotation vectors) and `tvecs` (translation vectors) define the marker's pose relative to the camera.

2.4.2 Frame Transformation

Once the pose is estimated in the camera frame, it must be transformed into a global reference frame, such as the map frame. This requires converting the rotation vector into a rotation matrix and applying coordinate transformations.

The homogeneous transformation matrix T for transforming a point from the camera frame to the map frame is:

$$T_{map}^{camera} = \begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix} \tag{2.2}$$

where:

- R is the 3x3 rotation matrix derived from `cv2.Rodrigues(rvec)`.
- t is the 3x1 translation vector.

This is implemented in the code as:

```
rotation_matrix, _ = cv2.Rodrigues(np.array(rvecs[i][0]))
tf_matrix = np.eye(4)
tf_matrix[:3, :3] = rotation_matrix
tf_matrix[:3, 3] = tvecs[i][0]
```

Listing 2.7. Applying frame transformation

2.4.3 Transforming Marker Pose to Map Frame

Using the ROS TF2 library, the detected marker's position is transformed from the camera frame to the map frame. The transformation is applied using:

```
transform = self.tf_buffer.lookup_transform('map', '
    oakd_rgb_camera_optical_frame', rclpy.time.Time())
transformed_t = self.transform_point(tvecs[i], transform)
```

Listing 2.8. Transforming marker pose to map frame

Here, the marker's pose in the camera frame is transformed into the map frame using the stored TF2 transform.

By leveraging these mathematical principles and coordinate transformations, the system accurately determines the Aruco marker's position in the global reference frame, enabling precise robot navigation and docking.

2.5 Navigation and Goal Execution Using Aruco Markers

Once the Aruco marker's pose is estimated in the map frame, it can be used as a navigation target for the TurtleBot4. The goal position is sent to the Nav2 stack, which handles path planning, obstacle avoidance, and movement execution to reach the desired destination.

2.5.1 Sending a Goal to the TurtleBot4

Using the estimated pose of the Aruco marker, a navigation goal is defined and sent to the Nav2 stack. This is typically done using the `NavigateToPose` action in ROS 2. The goal includes:

- The position of the Aruco marker in the map frame.
- The orientation derived from the marker's detected pose.

The following ROS 2 command can be used to send the goal:

```python
from geometry_msgs.msg import PoseStamped
from nav2_simple_commander.robot_navigator import BasicNavigator

navigator = BasicNavigator()
goal_pose = PoseStamped()

goal_pose.header.frame_id = 'map'
goal_pose.pose.position.x = aruco_pose.x
goal_pose.pose.position.y = aruco_pose.y
goal_pose.pose.orientation.w = 1.0

navigator.goToPose(goal_pose)
```

Listing 2.9. Sending goal to Nav2 stack

2.5.2 Path Planning and Obstacle Avoidance Using Nav2

The Nav2 stack is a robust ROS 2 package responsible for autonomous navigation. It consists of several components:

- **Global Planner**: Computes an optimal path from the robot's current position to the goal, avoiding known obstacles.
- **Local Planner**: Continuously refines the robot's trajectory based on real-time sensor data.
- **Costmap Server**: Maintains an up-to-date map of obstacles detected by the robot's LIDAR and other sensors.
- **Controller Server**: Converts the planned path into motor commands that drive the TurtleBot4.

2.5.3 SLAM and LIDAR for Localization and Obstacle Avoidance

Simultaneous Localization and Mapping (SLAM) enables the robot to build a map of its environment while keeping track of its own position. The TurtleBot4 relies on a LIDAR sensor for real-time obstacle detection and avoidance:

- The LIDAR scans the environment and updates the costmap to prevent collisions.
- The SLAM algorithm corrects localization errors using sensor fusion and feature matching.

- Nav2 dynamically adjusts the planned path if new obstacles appear.

2.5.4 Iterative Refinement of the Goal Position

As the robot approaches the goal, periodic updates to the Aruco marker's pose improve positional accuracy. This is achieved by:

- Continuously detecting the marker and refining its estimated position.
- Recomputing the goal position and updating the navigation target if necessary.
- Ensuring precise docking by reducing positional errors in the final approach.

By leveraging the mature ROS 2 Nav2 stack, SLAM, and LIDAR-based obstacle detection, the system ensures efficient and reliable navigation to the Aruco marker's position while continuously refining the goal for improved accuracy.

2.6 Camera Distortion and Parameter Calibration

In robotic systems, accurate pose estimation is crucial for tasks such as navigation and object detection. One of the challenges in achieving precise pose estimation is the effect of camera distortion. Distortion refers to the way in which the image captured by the camera deviates from the true scene, often due to imperfections in the camera lens. To account for this, camera calibration parameters are used to correct the distortion in the images before further processing, such as Aruco marker detection.

2.6.1 Camera Parameters and Distortion

The camera calibration parameters are extracted from the `camera_info` topic published by the camera driver. This topic contains the intrinsic and extrinsic parameters of the camera, which are essential for accurate image processing and 3D pose estimation.

Intrinsic Parameters

The intrinsic parameters of a camera define the internal characteristics of the camera, including the focal length, the optical center, and the distortion coefficients. These parameters are necessary for transforming the image coordinates into real-world coordinates. The main intrinsic parameters are:

- **Focal length** (f_x, f_y): These parameters represent the distance between the camera's lens and the image sensor. They are usually expressed in pixels and are used to scale the image coordinates to real-world dimensions.
- **Optical center** (c_x, c_y): This is the point where the optical axis of the camera intersects the image plane. It is typically located near the center of the image.
- **Distortion coefficients** $(k_1, k_2, k_3, p_1, p_2)$: These coefficients model the radial and tangential distortion introduced by the camera lens. Radial distortion

causes straight lines to appear curved, while tangential distortion occurs due to the misalignment of the lens with the image sensor.

The intrinsic parameters are typically represented in the camera matrix:

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

where f_x and f_y represent the focal lengths in the x and y directions, and c_x and c_y represent the optical center coordinates.

Distortion Model

Camera distortion is typically modeled using a radial distortion model along with tangential distortion:

$$x_{\text{distorted}} = x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) + 2p_1 xy + p_2(r^2 + 2x^2)$$
$$y_{\text{distorted}} = y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) + p_1(r^2 + 2y^2) + 2p_2 xy$$

where $r^2 = x^2 + y^2$ is the radial distance from the center of the image, and p_1, p_2, k_1, k_2, k_3 are the distortion coefficients.

2.6.2 Extraction of Camera Parameters

In ROS 2, the `camera_info` topic provides the camera calibration parameters. This topic includes the camera matrix and distortion coefficients, which can be extracted and used to undistort the images.

The intrinsic parameters can be accessed as follows in the ROS 2 node:

```python
from sensor_msgs.msg import CameraInfo

def camera_info_callback(self, msg):
    # Extract intrinsic parameters from CameraInfo message
    self.mtx = np.array(msg.K).reshape(3, 3)  # Camera matrix
    self.distortion_coeffs = np.array(msg.D)  # Distortion
        coefficients
```

Listing 2.10. Extracting camera parameters from the camera_info topic

Here, the camera matrix K is extracted from the `msg.K` field of the `CameraInfo` message. The distortion coefficients are similarly retrieved from the `msg.D` field.

2.6.3 Correcting Distortion in the Images

Once the intrinsic parameters and distortion coefficients are obtained, they are used to undistort the images before any further processing, such as Aruco marker detection. OpenCV provides the `undistort` function, which takes in the camera matrix and distortion coefficients to correct the distortion in the image.

The distortion correction is applied as follows:

```python
# Undistort the current image frame using the camera parameters
undistorted_frame = cv2.undistort(current_frame, self.mtx, self.
    distortion_coeffs)
```

Listing 2.11. Undistorting the camera image

This function adjusts the image pixels to compensate for radial and tangential distortion, resulting in an undistorted image that more accurately represents the real-world scene.

2.6.4 Impact of Distortion Correction on Aruco Marker Detection

Undistorting the images before marker detection is crucial for the accuracy of Aruco marker localization. Distorted images can lead to incorrect detection of the marker's position and orientation. By using the corrected image, the Aruco detection algorithm is able to more accurately locate the marker corners, leading to more precise pose estimation and improved navigation performance.

By accounting for distortion through camera calibration, the system can correct for lens imperfections and provide a more reliable basis for tasks such as autonomous navigation and docking.

Chapter 3

Results and Performance

In this section, we evaluate the performance of the system in terms of Aruco marker detection, pose estimation, goal sending, and path planning. Despite the success of these key components in principle, the overall simulation experience was plagued by technical challenges, including performance issues and inconsistencies in the Gazebo simulation environment. These issues affected the smooth operation of the system, particularly in terms of visual feedback, marker detection, and path planning accuracy.

3.0.1 Aruco Marker Detection and Pose Estimation

The Aruco marker detection and pose estimation system performed quite well in terms of accuracy. When the marker was detected, the pose estimation system provided reliable and precise results, allowing the robot to accurately determine its relative position to the marker. Figure 3.1 shows an example of the detected marker from the camera feed, where the system was able to clearly identify the marker despite occasional challenges with the simulation.

However, the detection process was not always consistent due to limitations in the simulation environment. The camera feed in Gazebo was sometimes pixelated and exhibited inconsistent frame rates, which led to some instability in marker detection. These visual inconsistencies caused the marker detection to occasionally appear jumpy, making the pose estimation less reliable in certain moments. This can be attributed to the overall lower quality of the simulated camera feed and the limitations of the Gazebo simulation in handling real-time image processing.

3.0.2 Path Planning and Navigation

The path planning algorithm in the system successfully calculated an optimal route to the target goal. Figure 3.2 shows a visualization of the planned path in Rviz2, which provides a clear view of the intended trajectory for the robot. However, the actual navigation to the goal proved to be less reliable than anticipated. The robot frequently strayed from the planned path, requiring frequent path updates. This issue was primarily caused by inconsistent physics within the simulation, which affected the robot's movement and overall navigation behavior.

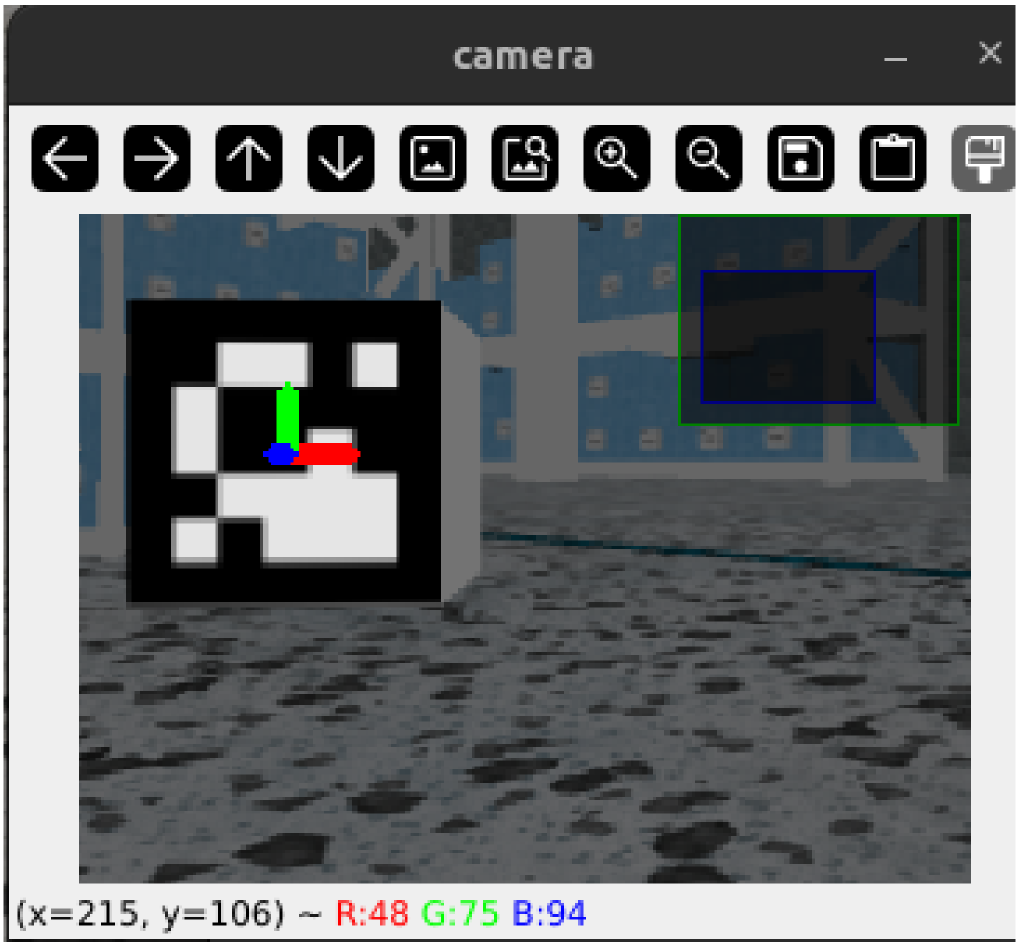
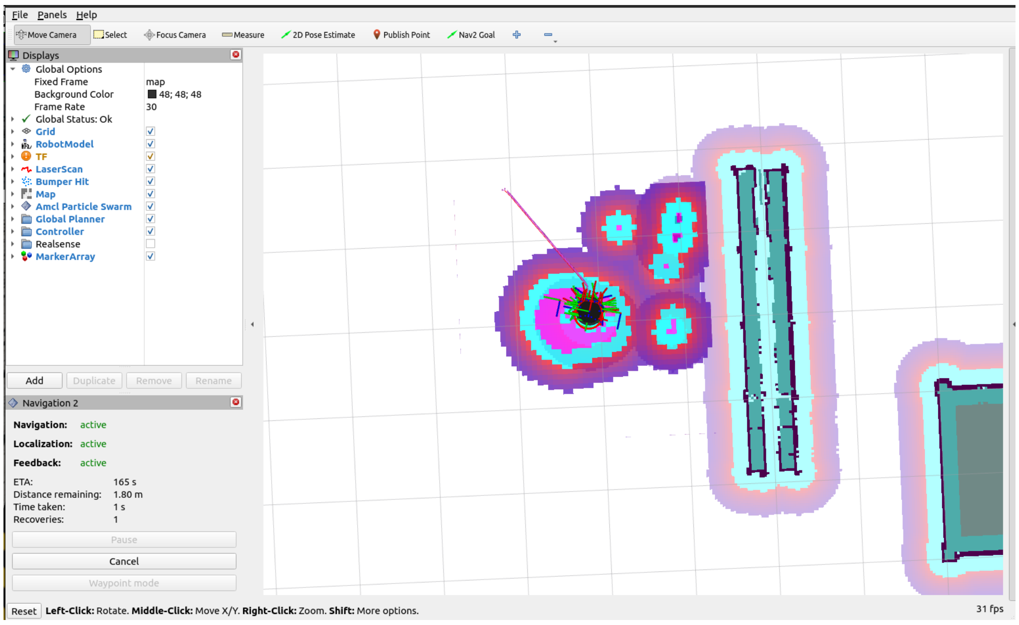


Figure 3.1. A view of the detected marker from the camera feed.



Global Options
Fixed Frame map
Background Color 48; 48; 48
Frame Rate 30
Global Status: Ok
Grid ✓
RobotModel ✓
TF ✓
LaserScan ✓
Bumper Hit ✓
Map ✓
Amcl Particle Swarm ✓
Global Planner ✓
Controller ✓
Realsense ☐
MarkerArray ✓

Navigation: active
Localization: active
Feedback: active

ETA: 165 s
Distance remaining: 1.80 m
Time taken: 1 s
Recoveries: 1

Left-Click: Rotate. **Middle-Click:** Move X/Y. **Right-Click:** Zoom. **Shift:** More options.

Figure 3.2. A view of the planned path in Rviz2.

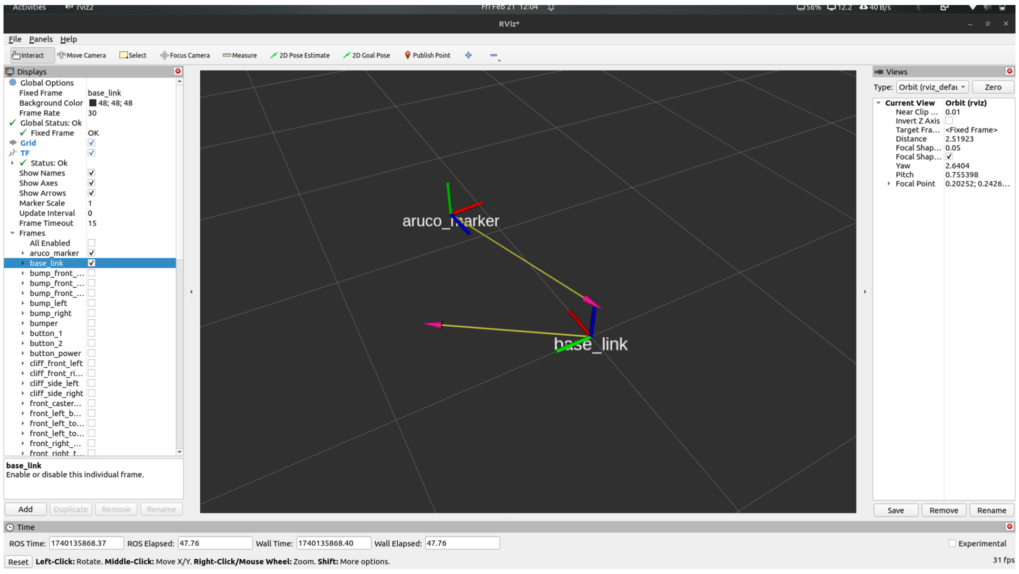
These inconsistencies in navigation led to unrealistic robot behavior, with the robot sometimes taking unexpected routes or failing to follow the planned path due to simulated environmental factors, such as the physics engine's instability. This inconsistency in the simulation environment made the robot's navigation to the goal somewhat erratic and inefficient.

3.0.3 Frame Transformations and Robot Localization

Despite the challenges in marker detection and navigation, the robot was able to maintain accurate localization relative to the Aruco marker, as shown in Figure 3.3. This figure illustrates the tf2 frames, with the marker's position and orientation relative to the base frame of the robot. This localization was critical for ensuring that the robot could navigate effectively towards its target despite the issues with path execution.

3.0.4 Gazebo Simulation Overview

The Gazebo simulation environment, where all the above processes were executed, provided a useful but limited framework for testing the system. Figure 3.4 presents an overview of the Gazebo simulation with the Aruco marker in the environment. While the simulation allowed for the visualization of the system's components and actions, it was not without its own set of challenges. The inconsistencies in the physics engine, combined with issues in rendering and sensor simulation, contributed to a frustrating experience during testing.



Global Options
Fixed Frame base_link
Background Color 48; 48; 48
Frame Rate 30
✓ Global Status: Ok
✓ Fixed Frame OK
Grid ✓
TF ✓
✓ Status: Ok
Show Names ✓
Show Axes ✓
Show Arrows ✓
Marker Scale 1
Update Interval 0
Frame Timeout 15
Frames
All Enabled ☐
aruco_marker ✓
base_link ✓
bump_front_… ☐
bump_front_… ☐
bump_front_… ☐
bump_left ☐
bump_right ☐
bumper ☐
button_1 ☐
button_2 ☐
button_power ☐
cliff_front_left ☐
cliff_front_ri… ☐
cliff_side_left ☐
cliff_side_right ☐
front_caster… ☐
front_left_b… ☐
front_left_to… ☐
front_left_to… ☐
front_right_… ☐
front_right_t… ☐

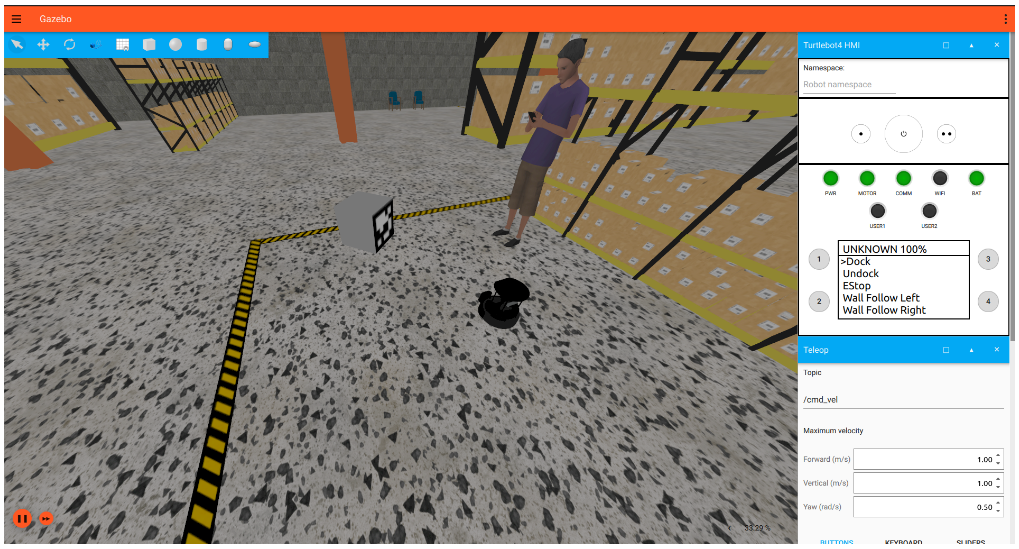
base_link
Enable or disable this individual frame.

Type: Orbit (rviz_defau
Zero
Current View **Orbit (rviz)**
Near Clip … 0.01
Invert Z Axis ☐
Target Fra… <Fixed Frame>
Distance 2.51923
Focal Shap… 0.05
Focal Shap… ✓
Yaw 2.6404
Pitch 0.755398
Focal Point 0.20252; 0.2426…

ROS Time: 1740135868.37 ROS Elapsed: 47.76 Wall Time: 1740135868.40 Wall Elapsed: 47.76

Left-Click: Rotate. **Middle-Click:** Move X/Y. **Right-Click/Mouse Wheel:** Zoom. **Shift:** More options.

Figure 3.3. A view of the tf2 frames showing the marker relative to the base robot frame.



UNKNOWN 100%
>Dock
Undock
EStop
Wall Follow Left
Wall Follow Right

Topic
/cmd_vel
Maximum velocity
Forward (m/s) 1.00
Vertical (m/s) 1.00
Yaw (rad/s) 0.50

Figure 3.4. Overview of the Gazebo simulation with Aruco marker.

3.1 Test Case: Marker Detection and Navigation

To evaluate the accuracy and effectiveness of the system, a test case was conducted where an Aruco marker was placed at a known position relative to the robot. The marker was positioned at coordinates $(x, y) = (1.55, -0.55)$ meters with a rotation of 45° about the Z-axis. The robot, utilizing its onboard camera and Aruco detection system, estimated the marker's position and orientation.

3.1.1 Pose Estimation Accuracy

Upon detecting the marker, the system estimated its pose as $(x, y) = (1.53, -0.58)$ meters with an orientation of 46°. This result demonstrates a high degree of accuracy in both position and angular estimation, with only minor deviations from the ground truth. The small errors in position (approximately 2 cm in x and 3 cm in y) and orientation (1-degree offset) can be attributed to the inherent noise in the simulation's camera feed and processing pipeline. These deviations remained within an acceptable range for successful navigation.

3.1.2 Navigation Performance

Once the marker's position was estimated, the robot proceeded to navigate towards the goal using the computed path plan. The navigation system successfully directed the robot to the target location while adapting to environmental constraints. Despite some minor path deviations due to the simulation's physics inconsistencies, the robot was able to reach the marker's location without significant issues.

3.1.3 Observations and Insights

The results of this test case highlight the robustness of the marker detection and navigation system. The system was able to:

- Accurately estimate the marker's position and orientation with minimal error.
- Successfully generate a viable path towards the marker.
- Navigate effectively to the target despite minor inconsistencies in the simulation.

These findings validate the core functionality of the system while also highlighting areas for improvement, particularly in refining pose estimation stability and enhancing navigation in imperfect simulation conditions.

3.1.4 Future Improvements

The core components of the system—Aruco marker detection, pose estimation, goal sending, and path planning—functioned as expected, but the overall simulation performance left much to be desired. The marker detection and pose estimation were accurate when stable, but the simulation environment introduced inconsistencies that affected the reliability of these processes. Additionally, the navigation was hindered by issues with path following and inconsistent physics, making the goal-reaching behavior less predictable.

To improve system performance, it would be necessary to address the underlying simulation issues, particularly related to the camera feed quality, physics engine stability, and frame rates. Additionally, refining the path planning algorithm to be more adaptive to these variations could help mitigate some of the inconsistencies observed during navigation. Finally, transitioning to a more robust simulation environment or implementing hardware-based testing may provide more reliable and realistic results in future experiments.

Chapter 4

Conclusion

This project successfully integrated Aruco marker detection, pose estimation, and path planning for autonomous robot navigation within a simulated environment using Gazebo. The core functionalities—marker detection, pose estimation, goal sending, and path planning—were demonstrated to work effectively in theory, with the pose estimation providing reliable and accurate results when the marker was detected. The path planning system successfully computed optimal routes to the target goals, and the robot was able to navigate towards the goal under typical conditions.

However, despite these successes, several issues related to the Gazebo simulation environment hindered the overall performance. The simulated camera feed suffered from pixelation and inconsistent frame rates, leading to unstable marker detection. This instability caused occasional jumpiness in the pose estimation system, affecting the robot's ability to reliably interpret its environment. In addition, inconsistencies within the simulation's physics engine resulted in unrealistic navigation behavior, with the robot frequently straying from the planned path, necessitating frequent updates.

Despite these challenges, the project demonstrated the feasibility of combining marker-based localization with autonomous navigation in a robotic system. The technical difficulties encountered during the simulation emphasized the importance of robust simulation environments and real-world testing for such complex tasks. Moving forward, future improvements could focus on refining the simulation's accuracy, enhancing camera feed quality, and stabilizing the physics engine. Additionally, improving the adaptability of the path planning algorithm to better handle these inconsistencies would result in a more reliable and efficient navigation system.

In conclusion, while the project faced technical obstacles during its execution, it highlighted the strengths and weaknesses of the integrated system and provided valuable insights for further refinement and future work in the field of autonomous robotics.