Final Project Report

ENPM 605 - Python Application for Robotics
Zeid Kootbally

Group 5
David Ajayi
Hamsaavarthan Ravichandar
Manoj Kumar Selvaraj

May 18, 2025

Table of Contents

Introduction

The project focuses on implementing an autonomous navigation system using ROS 2 in a Gazebo simulation environment, where the ‘ros-bot’ robot package must detect and interact with ArUco-marked cubes while navigating locations predefined in a parameters (.yaml) file based on marker IDs obtained from the camera. The system integrates mapping, waypoint navigation, ArUco-marker detection detection, and behavior execution, requiring the robot to dynamically react to its environment based on detected markers and preconfigured motion parameters.

Core Tasks:

1. Modifying Husarion World
    - Three Gazebo cylinders (radius: 0.5m, height: 0.5m) were introduced at specified coordinates to challenge navigation and enhance obstacle avoidance logic.
    - Two cubes were placed strategically, each featuring unique ArUco markers for detection and interaction.
    - A Gazebo camera was positioned to monitor Cube #1.
2. Mapping the Environment
    - Use slam_toolbox to create a map of the simulated world, ensuring accurate representation of static obstacles and navigation paths.
    - Store the generated map for later use in navigation tasks.
3. Navigating to Marked Cubes
    - Subscribe to /aruco_markers to detect ArUco marker IDs on cube faces.
    - Convert marker pose from camera frame to map frame using transformation logic.
    - Guide the robot to a position opposite the detected marker and prepare for further movement decisions.
4. Waypoint Navigation Using Parameters
    - Retrieve goal poses from a YAML parameter file, which stores positions, orientations, and navigation behaviors associated with specific marker IDs.
    - Move the robot to different predefined waypoints, ensuring accurate pose execution and parameter-based control.
5. Executing Circling Behavior Around Cube #2
    - Detect the ArUco marker ID on Cube #2 to determine the robot’s final goal.
    - Compute a circular path around the cube using trajectory planning techniques.
    - Execute movement around Cube #2 in a clockwise or counterclockwise direction, depending on parameter settings.
6. Final Navigation Task
    - Use the detected marker ID to retrieve the robot’s final goal location from the parameter file.
    - Navigate to the final destination while considering environmental constraints and obstacle avoidance strategies.

Environment Setup

The final project uses a modified Husarion World, which contains two cubes with AruCo markers and one Gazebo camera facing one side of the cube (Figure 1).

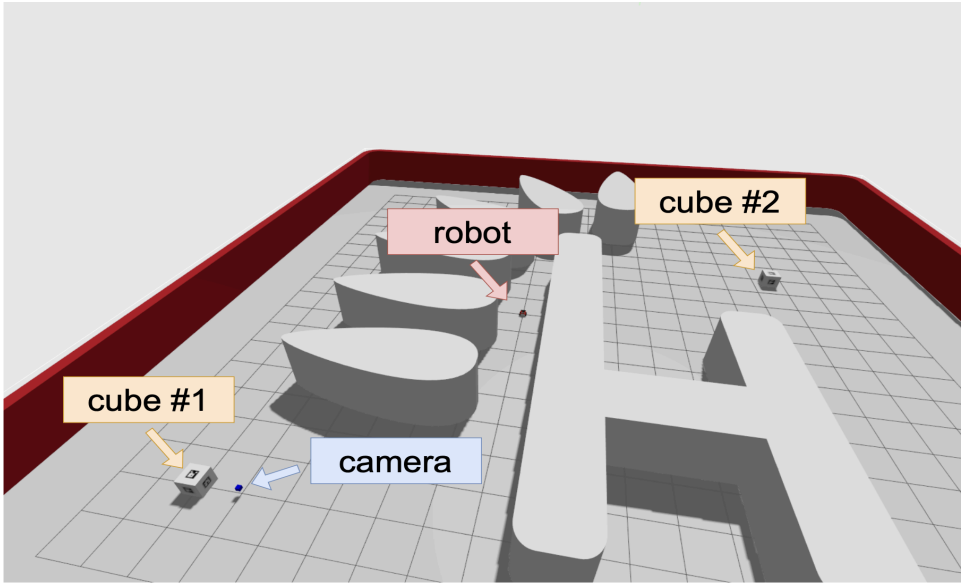

Fig 1: Modified Husarion world with two cubes, one robot, and one camera.

To enhance the Husarion Gazebo world, we integrated three additional static obstacles, Gazebo cylinders at predefined coordinates to challenge obstacle avoidance logic. These modifications were applied within the Gazebo SDF (Simulation Description Format) file used by the Gazebo launch file, ensuring correct object placement, dimensions, and scene integration.

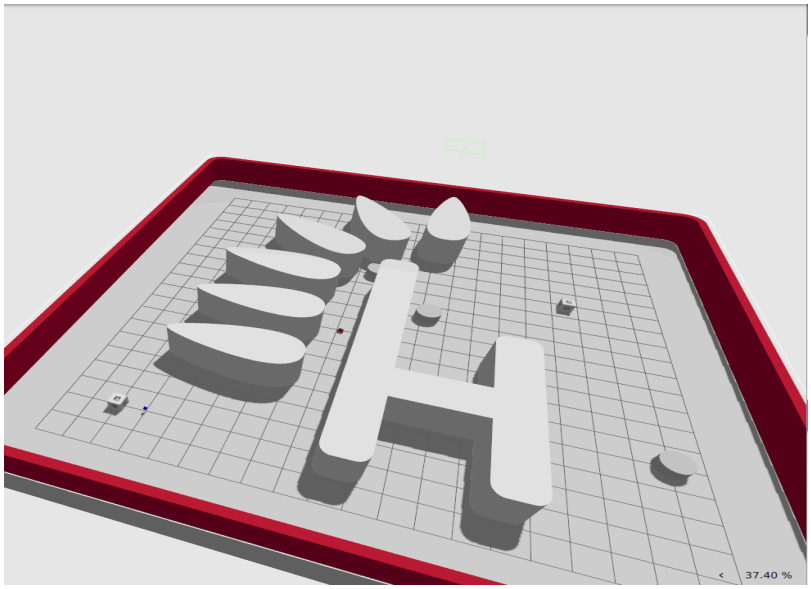

Fig 2: Enhanced Husarion world with three static cylindrical obstacles

Parameter File Design

The cube navigation parameters were structured in cube_nav_params.yaml to define goal poses, orientations, and navigation behaviors required for waypoint-based movement. The key parameters include:

- Cube 2 Goal: Stores goal positions for Cube #2, specifying its ID, position, orientation (quaternion or Euler), and circular movement direction (clockwise/counterclockwise).
- Final Goal: Defines four final navigation destinations, each associated with a marker ID, ensuring the robot reaches a designated endpoint.
- Waypoint Pose Extraction: Goal positions were manually collected from Gazebo's Component Inspector, aligning navigation tasks with detected marker locations.

```yaml
/**:
  ros__parameters:
    cube2_goal:
      goal_0:
        id: 2
        position: [3.75, -9.75]
        orientation: [0.7071, 0.0, 0.0, 0.7071]
        circular_direction: 0
      goal_1:
        id: 4
        position: [2.5, -8.5]
        orientation: [1.0, 0.0, 0.0, 0.0]
        circular_direction: 1
      goal_2:
        id: 5
        position: [3.75, -7.25]
        orientation: [0.7071, 0.0, 0.0, -0.7071]
        circular_direction: 0
      goal_3:
        id: 6
        position: [5.0, -8.5]
        orientation: [0.0, 0.0, 0.0, 1.0]
        circular_direction: 1

    final_goal:
      "2":
        position: [-1.8, 2.1]
        orientation: [1.0, 0.0, 0.0, 0.0]
      "4":
        position: [-8.0, -2.1]
        orientation: [1.0, 0.0, 0.0, 0.0]
      "5":
        position: [-6.0, -9.0]
        orientation: [1.0, 0.0, 0.0, 0.0]
      "6":
        position: [8.5, 8.5]
        orientation: [1.0, 0.0, 0.0, 0.0]
```

Fig 3: Parameters (cube_nav_params.yaml) file

Marker Detection

ArUco Marker Detection

ArUco markers are detected in images using OpenCV's cv2.aruco.detectMarkers() function, which identifies the ID. Later, the marker IDs are converted to their corresponding unique target coordinate information in the Gazebo world.

```
# Detect ArUco markers in the image
aruco_dict = cv2.aruco.Dictionary_get(cv2.aruco.DICT_5X5_50)
parameters = cv2.aruco.DetectorParameters_create()
corners, ids, _ = cv2.aruco.detectMarkers(cv_image, aruco_dict, parameters=parameters)
```

Fig 4: Aruco detection

Camera-Marker Transformation

The relative position of the marker with respect to the camera is obtained through subscribing the topic '/arcuo_markers', and the pose is transformed to the global '/map' frame using ROS TF transformations, so that the ArUco marker detected by the camera frame can be added to the ROS TF Tree using a TF broadcaster.

The camera object is hardcoded with pose (-8,7)

```
##################Change 8.0 and 7.0 based on camera position#####################
x = marker_pose.position.x - 8.0 # Adjust the position based on camera distance
dist_camera_from_cube = 1.0 # Distance of the camera from the cube
y = marker_pose.position.y + (7.0 + dist_camera_from_cube) - self._initial_bot_pose
```

Fig 5: Marker position

Pose Estimation and Navigation

The marker's position and orientation (pose) with respect to the camera frame are used to calculate the goal 1 position, based on the distance between camera and ArUco marker, cube size, required clearance from the cube 1, which is later used by the robot to navigate to the marker or perform tasks like rotation facing it.

TF Broadcast and Goal Publishing

The marker's transformed position with respect to the '/map' frame is broadcast using ROS TF, which can be then obtained using TF Listener. Finally, the obtained goal position calculated is published to the '/goal_1_position' topic from the ArucoCamera node, allowing the robot's navigator node, CubeNavigator, to plan a feasible path and navigate to the updated goal position.

```
av-1] [INFO] [1747548576.591372551] [cube_navigator_node]: Detected ArUco marker IDs: 4
```

Fig 6: Output of Detected Aruco

Waypoint Navigation

- The robot defines a series of goals (waypoints) it needs to reach, using coordinates and orientations. These are represented as PoseStamped messages containing position (x, y, z) and orientation (quaternion values: x, y, z, w).
- The navigate1() and navigate2() functions are responsible for guiding the robot to specific waypoints (Cube 1, Cube 2, or final goal). These functions use the BasicNavigator to navigate the robot to each goal, sending the goal as a PoseStamped message and waiting for confirmation that the task has been completed.
- In perform_rotation(), the robot follows a circular path around a detected marker by generating multiple waypoints along a circular trajectory. The robot moves through these waypoints sequentially by calculating positions using polar coordinates (angles), ensuring smooth navigation around the marker.
- The robot receives feedback during navigation (self._navigator.getFeedback()) to update the status of its navigation task. If the robot completes a task, it proceeds to the next waypoint or goal.
- Waypoints can be dynamically adjusted based on the detected ArUco marker IDs. After reaching Cube 1, the robot uses the detected marker's position from the parameter to set new goals for Cube 2 or the final goal, ensuring adaptive navigation.

Circular Behaviour Implementation

The perform_rotation() function is responsible for guiding the robot along a circular path around a detected marker.

- Parameters include:
    - **direction**: Determines if the robot rotates clockwise (True) or counterclockwise (False).
    - **radius**: The radius of the circular path.
    - **steps**: The number of waypoints along the circular path.

The robot generates a list of angles based on the number of waypoints (steps) using the formula:

```
angles_deg = [i * (360 / steps) for i in range(steps)]
```

Fig 7: List of angles for waypoints

- The angles are either reversed or adjusted depending on the rotation direction (clockwise or counterclockwise).
- For each angle, the robot computes the x and y coordinates along the circular path using polar coordinates (radius and angle).
- The robot's orientation (yaw) is adjusted to face along the tangent of the circle, and the orientation is converted to a quaternion.

- A series of waypoints is created using PoseStamped messages, each containing the calculated position and orientation of the robot at a specific point along the circular path.
- These waypoints are added to a list and used to guide the robot.
- The robot follows the circular path by navigating through the generated waypoints using the followWaypoints() method.
- Feedback is provided throughout the navigation process to ensure the robot is successfully completing the circular motion.

Code Architecture

The code architecture consists of two primary ROS 2 nodes: **CubeNavigator** and **CameraAruco** nodes.

CubeNavigator

- Handles robot navigation using Nav2's BasicNavigator API to reach dynamically assigned waypoints.
- Loads navigation parameters (`cube2_goal` & `final_goal`) from 'cube_nav_params.yaml' YAML configuration files.
- Executes movement commands based on detected ArUco marker positions, retrieved from the `CameraAruco` node.
- Implements circular motion logic around Cube #2 by generating waypoints along a planned trajectory.
- Publishes TF transforms for detected objects, ensuring pose data is correctly mapped between different coordinate frames.
- Handles navigation task results and feedback through the ROS 2 action-based approach (`TaskResult`).

CameraAruco

- Subscribes to ArUco marker detection topic (`/aruco_markers`) and extracts pose of the ArUco marker data with respect to the camera frame.
- Performs coordinate frame transformations to adjust marker poses relative to the global map frame.
- Publishes detected marker positions as TF transforms, allowing correct integration into the navigation pipeline.
- Periodically checks for available transforms and updates goal positions in response to detected markers.
- Sends marker-based goal positions to `CubeNavigator` via ROS2 publishing to the topic `/goal_position_1`.

Workflow

1. Detect ArUco Markers using image processing in `CameraAruco` node.
2. Transform Marker Pose into the map frame and publish it as TF.
3. Retrieve Navigation Goals based on detected marker IDs from YAML parameters.
4. Navigate to Cube 1 & Cube 2, adjusting orientation dynamically.
5. Perform Circular Motion around Cube 2 using trajectory waypoints.
6. Detect Final Goal Marker, transform pose, and navigate to the final destination.

Communication

The nodes communicate via ROS 2 topics. CubeNavigator subscribes to goal positions and camera data, while CameraAruco handles marker detection and publishes updated goal positions.

Multi-Node Execution

Both nodes run concurrently using a MultiThreadedExecutor in the same executable, enabling real-time marker detection and navigation tasks.

This architecture ensures a modular and efficient approach to marker-based navigation, integrating real-time perception, TF transformations, and autonomous waypoint execution for structured movement control.

Results

The Cube Navigator System successfully demonstrated autonomous navigation, ArUco marker detection, and dynamic waypoint execution in the modified Husarion Gazebo environment. The implemented workflow allowed the robot to detect cube markers, transform their poses, and adjust navigation behaviors accordingly.

1. Marker-Based Navigation Execution

The robot successfully detected ArUco markers using OpenCV's ArUco library, extracting marker poses for waypoint assignment. Pose transformations accurately convert camera frame coordinates to map frame, ensuring precise localization. Navigation to the goals were dynamically assigned based on detected markers, integrating cube_nav_params.yaml for structured waypoint retrieval, as shown in Fig.8 and Fig.9 respectively.

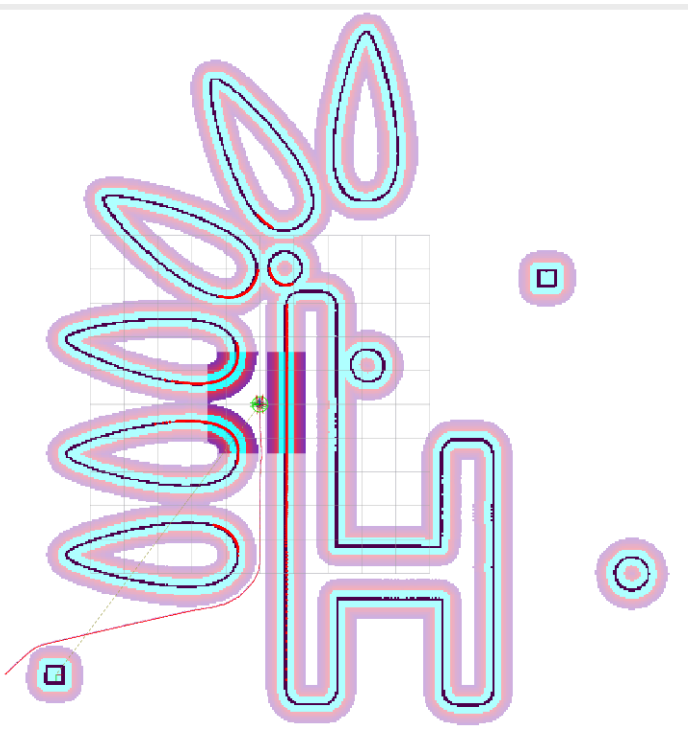

Fig 8: Navigation to Cube1

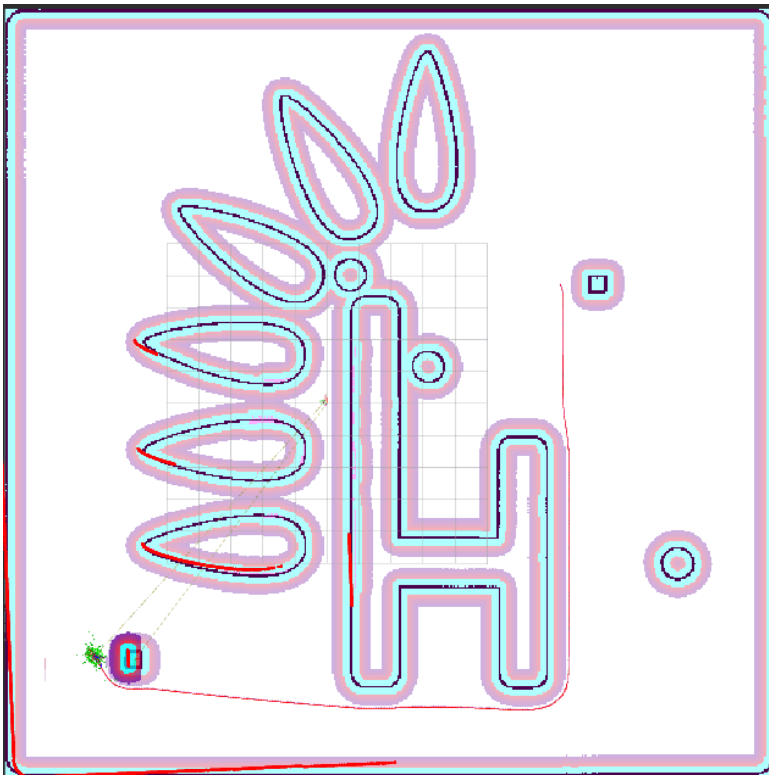

Fig 9: Navigation to Cube2 based on detected Aruco

2. Circular Motion Around Cube #2

The robot executed circular trajectories around Cube #2, following predefined waypoint generation logic. The direction of rotation (clockwise/counter-clockwise) was dynamically adjusted based on project parameters. Smooth trajectory execution confirmed effective motion planning, minimizing positional drift, as shown in Fig.10 below.

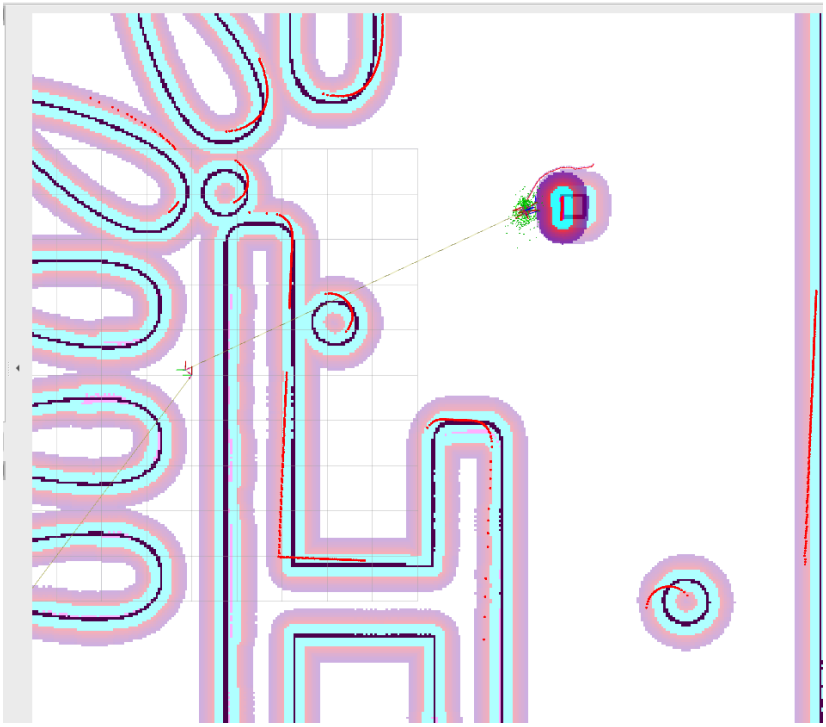

Fig 10: Waypoints for Rotation

3. Final Goal Reachability & Obstacle Handling

The robot successfully reached its final goal after detecting the last marker, validating overall navigation logic. Obstacle integration challenged the navigation system, requiring adaptive path adjustments for collision avoidance. The Husarion world modifications, including camera placement and environmental constraints, enhanced realistic testing conditions. The path taken by the robot during its final goal position is shown below, Fig.11.

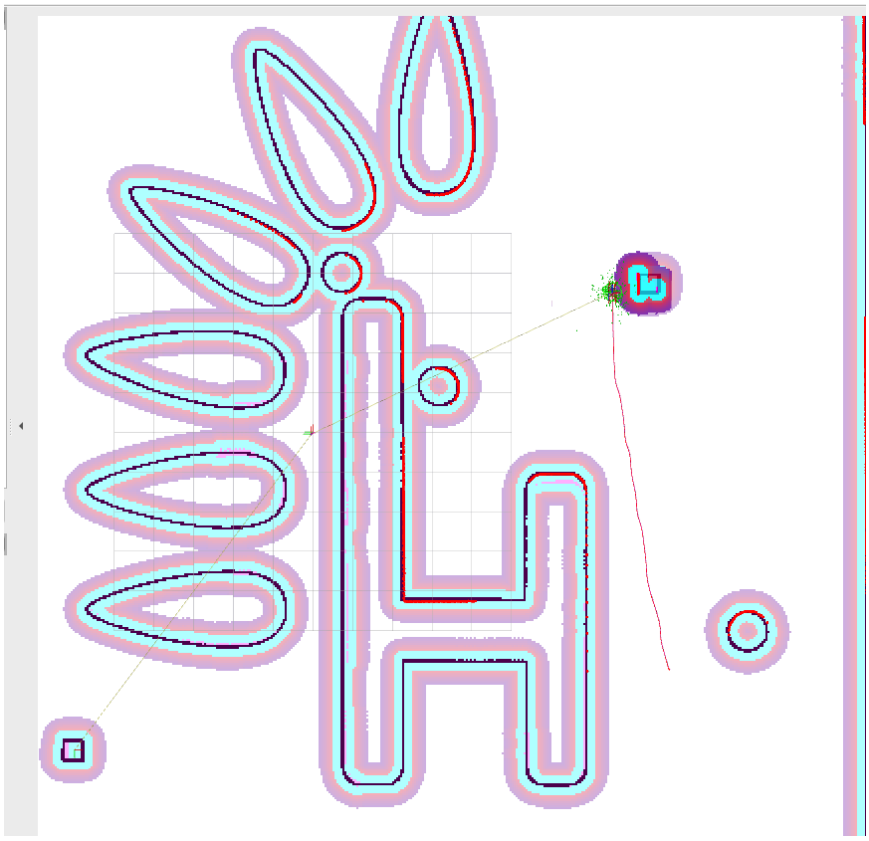

Fig 11: Navigation to Final Goal

While the navigation framework functioned as expected, marker detection accuracy depended on camera viewpoint and environmental factors. Threshold tuning and pose refinement were critical in improving reliable marker identification and navigation consistency. Overall, the project demonstrated a structured, parameter-driven autonomous navigation framework, aligning with real-time simulation-based robotics principles.

Contributions

David Ajayi

- Modified sdf to add cylinders

Hamsaavarthan Ravichandar

- Worked on subscribing to the /aruco_markers topic to read the ArUco marker data from the camera frame.
- The marker's position was then transformed into the global map frame using ROS 2's TF system.
- Navigating to cube 1, after calculating the coordinates in '/map/' frame

Manoj Kumar Selvaraj

- slam_toolbox for mapping
- Reading the Aruco ID from /camera topic and then navigating to cube 2
- Implementing circular rotation and navigation to the final goal

Bibliography

- *DEMO Link: https://drive.google.com/file/d/1uBkIAyNQVId7NzaDUtwefGhUmTxeCKmG/view?usp=sharing*