EECS 4421 Robotics: Final Project Report

Due Date and Time: December 2, 2025 / 11:59 pm

Name: Kevin Nguyen, **Student ID:** 217228255

Name: Kevin Duong, **Student ID:** 217043936

Name: Tom Vo, **Student ID:** 218799171

Name: Anagha Koroth, **Student ID:** 218819516

Abstract

This project presents the design and implementation of an autonomous mobile robot developed using ROS2, Gazebo, and Python, capable of navigating a simulated indoor environment through a combination of visual marker detection and LiDAR-based obstacle sensing. The robot follows a predefined path, identifies ArUco markers to locate navigation checkpoints, and safely responds to obstacles by stopping and resuming when the path becomes clear. The robot is implemented with camera perception with range sensing, which proves that the system demonstrates robust autonomous behaviour suitable for structured indoor spaces. While the initial proposal planned Nav2 navigation inside the lab map, the final implementation adopts a simplified marker-driven approach to ensure reliable performance in simulation. The results demonstrate consistent marker detection, stable navigation, and effective obstacle handling, providing a strong foundation for future improvements.

Table of Contents

1. Introduction

Autonomous mobile robots are essential in modern environments such as laboratories, hospitals, and industrial facilities. These robots need to navigate efficiently, detect obstacles, identify targets, and perform assigned tasks without human intervention.

In this project, we designed and implemented a simulated autonomous mobile robot using ROS2, Gazebo, and Python. The robot navigates through a virtual enclosed map environment, follows predefined routes, detects ArUco markers representing target locations, and detects obstacles using LiDAR. Beyond its technical capabilities, the robot is designed with potential real-world human-interactive applications in mind. It could be extended to assist users by delivering small items such as coffee or tea, transporting a mobile garbage bin, or performing simple service tasks in lab or office settings.

2. Background and Motivation

Autonomous navigation typically requires a combination of:

- Perception (understanding surroundings)
- Localization (knowing where the robot is)
- Planning (deciding where to go)
- Control (physically moving)

For this project, we used a marker-driven navigation strategy, where markers serve as local beacons that guide the robot's movement. This simplifies localization and allows focus on perception and control. We used ArUco markers because they are easy to detect reliably, each marker has a unique ID, and they work well in simulation. Obstacle detection was handled using a laser scanner, which provides 360 degree distance readings. This combination of visual and range-based sensing allows the robot to navigate safely towards its target.

3. System Architecture

The robot's navigation system consists of three major subsystems that operate together in real time. The perception subsystem includes the camera and LiDAR. The camera detects ArUco markers, extracts their IDs, and estimates their pose relative to the robot. The LiDAR provides 360 degree obstacle detection so that the robot can determine whether it is safe to move forward. The control subsystem uses these perception inputs to compute commands that ensure the robot rotates towards the given marker and moves forward. Finally, the task subsystem tracks which markers the robot must visit and transition from one target to the next one once it is completed.

3.1 Hardware

Although the robot is simulated in Gazebo, its design mimics a typical differential-drive mobile robot:

Sensors

- Camera Sensor: Used for ArUco marker detection via OpenCV.
- LiDAR (360° laser scanner): Provides distance measurements for obstacle detection.
- IMU: Helps stabilize orientation during turns.

Actuation

- Differential drive wheels (left and right velocity control)

Robot Model

- URDF/xacro model
- The camera is mounted on top of the robot.
- LiDAR mounted between the camera and base.

3.2 Software

ROS2: Provides communication through topics, services, and actions.
Gazebo: Simulates robot dynamics, sensors, physics, and the custom enclosed map.
Python: Core navigation and perception logic implemented in Python.
OpenCV ArUco Library: Used to detect marker IDs and calculate marker offsets.

3.3 Node Overview

The system consists of several ROS2 nodes working together in real time.

- The camera node publishes images for processing.
- The ArUco detection node extracts marker IDs and pose information.
- The LiDAR processing node analyzes distance data to detect obstacles.
- The navigation controller publishes velocity commands to /cmd_vel.
- The supervisor node coordinates all subsystems, prioritizing obstacle detection over marker targeting or path-following.

4. Map and Environmental Design

The initial proposal included a detailed recreation of the LAS1004 lab space:

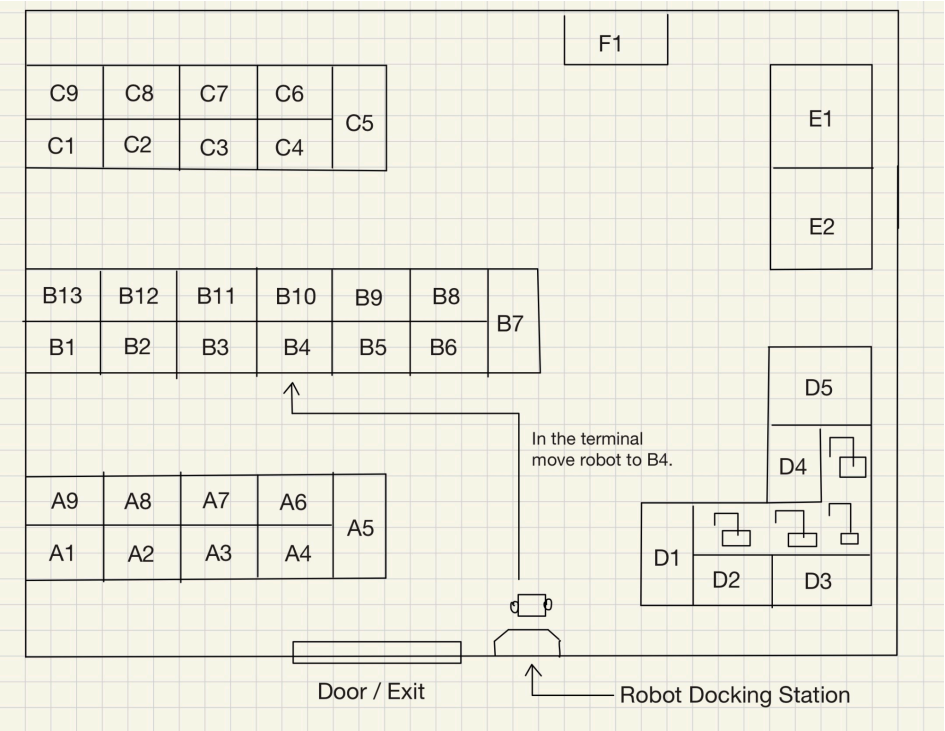

Figure 1: The initial map design from the proposal

However, due to time and complexity constraints, the final implementation adopted a simplified rectangular environment designed specifically for:

- Reliable camera and LiDAR sensor performance
- Smooth predefined trajectories
- Controlled ArUco marker placement

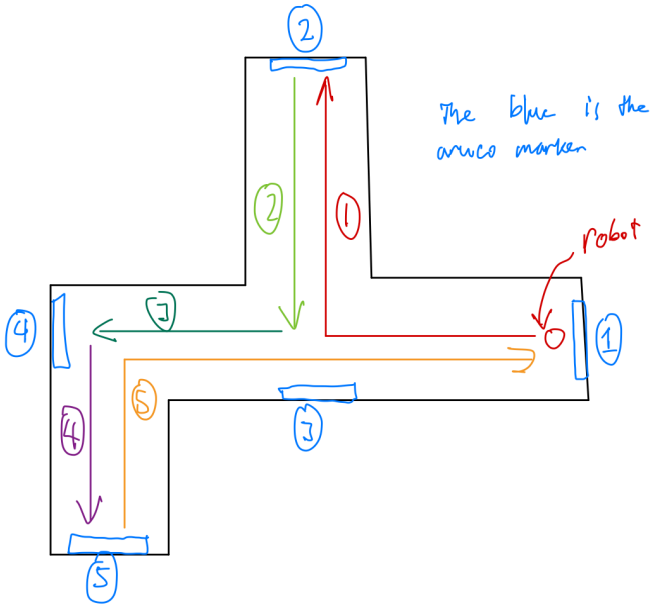

Figure 2: The sketch of final map layout

Final Environment Includes

- Enclosed walls
- Flat ground plane
- Five ArUco markers placed along the route
- Ample space for obstacle demonstrations
- A clear start and end point

The map's simplicity allowed a stable demonstration and reduced the drift introduced by Gazebo physics or poorly tuned differential-drive parameters.

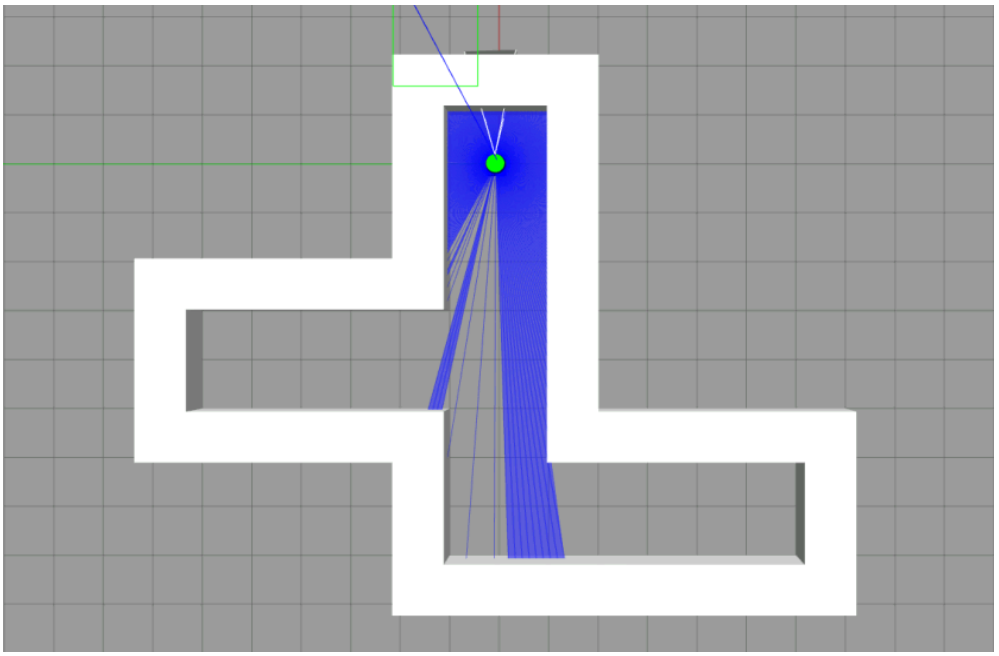

Figure 3: The map layout in Gazebo with the robot

5. Navigation and Marker-Based Targeting

The navigation system used a hybrid approach where both visual feedback and predefined movement were used to identify different markers.

5.1 Predefined Path Strategy

1. The robot starts by moving forward from the starting position
2. Follows a sequence of timed or distance-controlled motions
3. Adjusts heading at specific points
4. Uses ArUco markers to confirm navigation checkpoints

This deterministic approach ensures the robot never strays significantly from its intended route, even under sensor noise.

5.2 ArUco Marker Detection

Marker Detection:

1. Camera frame received via ROS2
2. Convert to grayscale
3. Apply the ArUco dictionary
4. Detect markers and extract ID
5. Compute orientation and distance
6. Send detection output to the supervisor node

Robot Behavior When Marker Detected

- Orientation corrected to face the marker
- The robot moves forward until close
- The robot stops when within the threshold distance
- Confirmation output printed in the terminal

6. Obstacle Detection

Obstacle detection is performed by using the LiDAR scanner

Methodology:

1. Evaluate the minimum distance in the forward-facing cone
2. If an object is detected, stop the robot immediately
3. The robot prints messages such as:
   "Obstacle detected – please move out of the way."
4. Once the object moves, resume motion automatically

7. Project Development

The original proposal and the final implementation are different in structure but still have similar goals. The robot is still a mobile device that could help and interact with people in an open environment.

7.1 Original Proposal

- Build the map's layout of the lab room (LAS 1004)
- Simulated trash collection
- Nav2-based navigation
- User terminal commands to choose the destination
- Physical robot in real life, if time allowed

7.2 Final Implementation

- Custom map
- Predefined navigation path

- ArUco-based station identification
- LiDAR obstacle detection
- Autonomous movement from start to destination
- Successful integration of camera + LiDAR
- Successful messages display when an object is detected

7.3 Reason for Changes

- Nav2 was unstable with the robot model
- SLAM performance in simulation was inconsistent
- The lab's map significantly increased development time
- ArUco-based navigation provided reliability and easy debugging to make changes if necessary.
- Custom environment allowed predictable behavior for evaluation

8. Results and Evaluation

The robot's simulation will be based on three criteria

8.1 Marker Detection

- Reliable detection from a far distance (<5m)
- The orientation estimation is accurate

8.2 Obstacle Detection

- The robot consistently stopped when encountering objects
- Reaction speed is less than 1 second from detection
- Resumed motion once the obstacle was removed

Works well for pedestrians and small boxes placed in the path

Limitations:

- No dynamic replanning around obstacles
- Large obstacles block the entire route

8.3 Navigation Stability

- The robot follows the predefined path smoothly
- Robot drift was minimal in straight segments
- Turning precision improved after adjusting PID parameters
- The system remained stable over repeated tests run

9. Challenges

Some of the challenges faced during the project were having the robot recognize the ArUco targets and drive towards them. Integrating obstacle detection with ArUco

target-based navigation. Ensuring the robot follows path points smoothly without straying off from its original path.

10. Demonstration

During the demonstration, the robot successfully followed its predefined navigation path and accurately drove toward each ArUco marker as intended. The vision system reliably detected the markers, allowing the robot to align itself and continue progressing through each checkpoint. When an obstacle was placed in front of the robot, the LiDAR system immediately triggered a stop and displayed a terminal message instructing that the object be moved out of its way. Once the obstacle was cleared, the robot automatically resumed its movement and continued its task without any manual intervention.

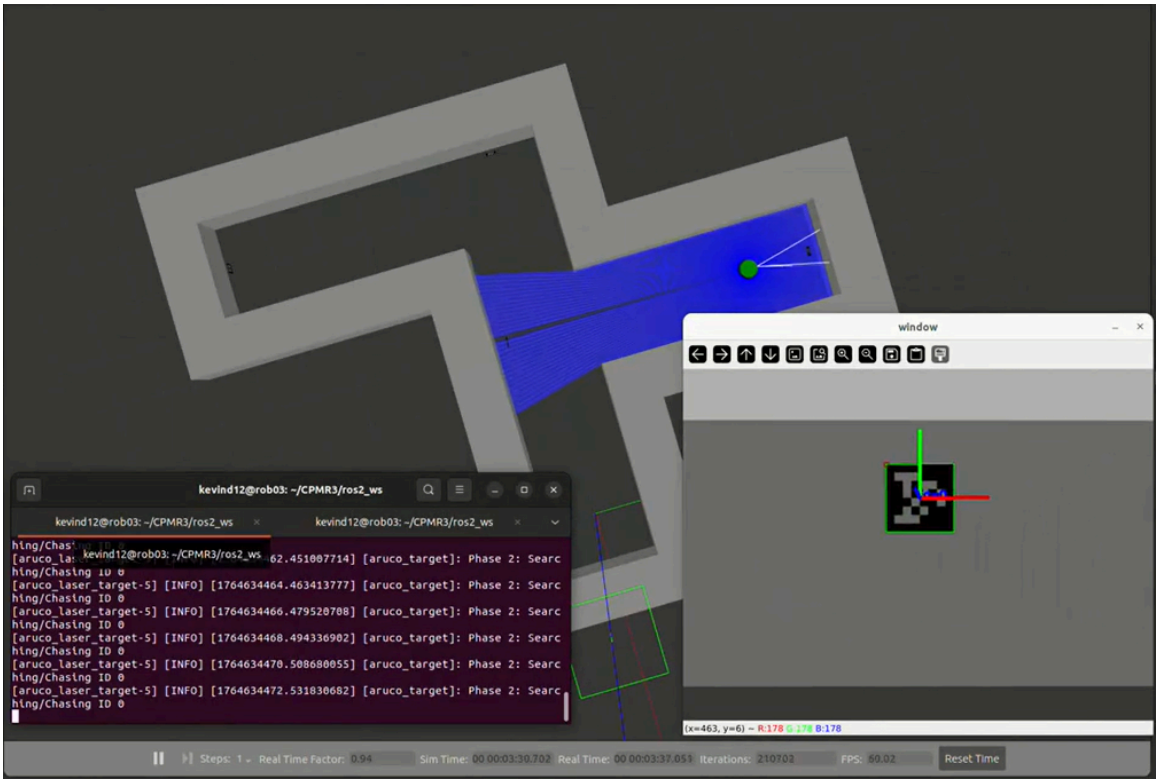

Figure 4: The robot following its predefined path was able to track the ArUco and drive towards it.

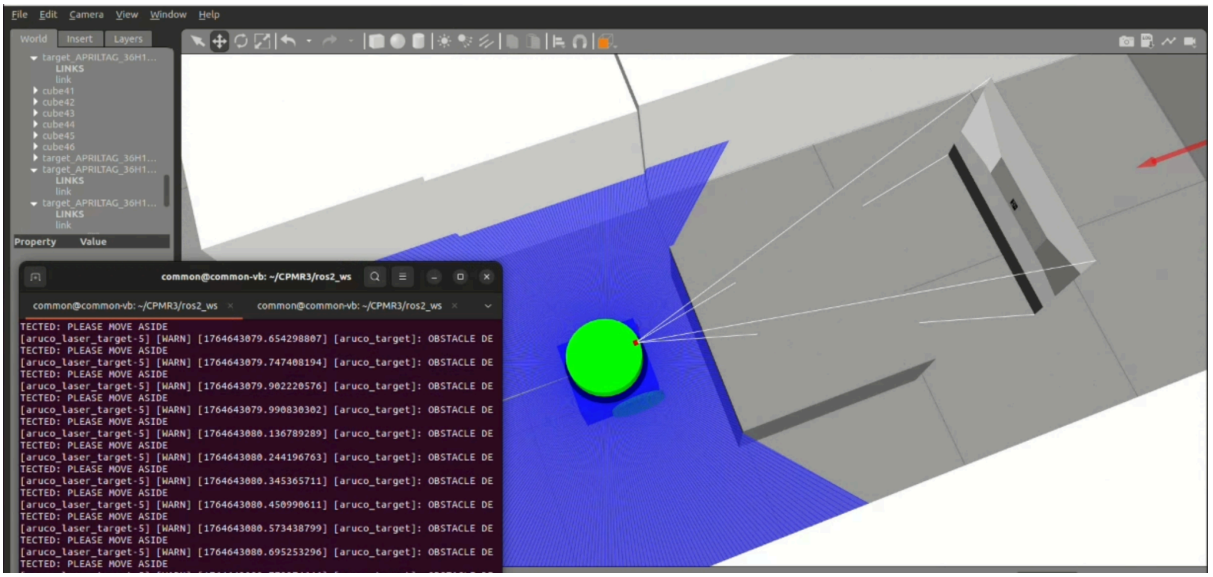

Figure 5: The robot successfully stops when an object is blocking the path. In the terminal a message “OBSTACLE DETECTED: PLEASE MOVE ASIDE” will be displayed

11. Future Work

Future improvements could enhance the system significantly. One important extension is enabling the robot to navigate around large obstacles rather than simply stopping. Another improvement is allowing the robot to recognize when a path is completely blocked and move on to the next task automatically. Additionally, implementing real-world applications where robots can assist humans.

12. Conclusion

This project demonstrates a functional marker-based navigation system that integrates camera perception, LiDAR obstacle avoidance, and motion control using ROS2. By navigating toward ArUco markers and handling tasks respectively, the robot showcases core autonomous navigation principles. While limitations remain, the system provides a strong foundation for future development for mobile robots that could be used for various services.

Appendix

GitHub Files Link: [Autonomous Mobile Robot]

The map file default.json:


```
{
        "cube1":  {"x": -8.5, "y":  1.5, "size": 1},
        "cube2":  {"x": -8.5, "y":  0.5, "size": 1},
        "cube3":  {"x": -8.5, "y": -0.5, "size": 1},
        "cube4":  {"x": -8.5, "y": -1.5, "size": 1},
        "cube5":  {"x": -8.5, "y": -2.5, "size": 1},
        "cube6":  {"x": -8.5, "y": -3.5, "size": 1},
        "cube7":  {"x": -8.5, "y": -4.5, "size": 1},
        "cube8":  {"x": -8.5, "y": -5.5, "size": 1},
        "cube9":  {"x": -8.5, "y": -6.5, "size": 1},
        "cube10":  {"x": -6.5, "y":  1.5, "size": 1},
        "cube11":  {"x": -7.5, "y":  1.5, "size": 1},
        "cube12":  {"x": -7.5, "y": -6.5, "size": 1},
        "cube13":  {"x": -6.5, "y": -6.5, "size": 1},
        "cube14":  {"x": -5.5, "y": -6.5, "size": 1},
        "cube15":  {"x": -5.5, "y": -5.5, "size": 1},
        "cube16":  {"x": -5.5, "y": -4.5, "size": 1},
        "cube17": {"x": -5.5, "y": -3.5, "size": 1},
        "cube18": {"x": -5.5, "y": -2.5, "size": 1},
        "cube19": {"x": -5.5, "y": -1.5, "size": 1},
        "cube20": {"x": -5.5, "y":  1.5, "size": 1},
        "cube21": {"x": -5.5, "y":  2.5, "size": 1},
        "cube22": {"x": -5.5, "y":  3.5, "size": 1},
        "cube23": {"x": -5.5, "y":  4.5, "size": 1},
        "cube24": {"x": -5.5, "y":  5.5, "size": 1},
        "cube25": {"x": -5.5, "y":  6.5, "size": 1},
        "cube26": {"x": -4.5, "y": -1.5, "size": 1},
        "cube27": {"x": -4.5, "y":  6.5, "size": 1},
        "cube28": {"x": -3.5, "y": -1.5, "size": 1},
        "cube29": {"x": -3.5, "y":  6.5, "size": 1},
        "cube30": {"x": -2.5, "y": -1.5, "size": 1},
        "cube31": {"x": -2.5, "y":  1.5, "size": 1},
        "cube32": {"x": -2.5, "y":  2.5, "size": 1},
        "cube33": {"x": -2.5, "y":  3.5, "size": 1},
        "cube34": {"x": -2.5, "y":  4.5, "size": 1},
        "cube35": {"x": -2.5, "y":  5.5, "size": 1},
        "cube36": {"x": -2.5, "y":  6.5, "size": 1},
        "cube37": {"x": -1.5, "y": -1.5, "size": 1},
        "cube38": {"x": -1.5, "y":  1.5, "size": 1},
        "cube39": {"x": -0.5, "y": -1.5, "size": 1},
        "cube40": {"x": -0.5, "y":  1.5, "size": 1},
        "cube41": {"x":  0.5, "y": -1.5, "size": 1},
```

```
        "cube42": {"x":  0.5, "y":  1.5, "size": 1},
        "cube43": {"x":  1.5, "y": -1.5, "size": 1},
        "cube44": {"x":  1.5, "y": -0.5, "size": 1},
        "cube45": {"x":  1.5, "y":  0.5, "size": 1},
        "cube46": {"x":  1.5, "y":  1.5, "size": 1}
}
```

Code for aruco_laser_target.py:

```
import math
import time
import numpy as np
import rclpy
from rclpy.node import Node
from rclpy.parameter import Parameter
import cv2
from cv_bridge import CvBridge, CvBridgeError
from sensor_msgs.msg import Image
from sensor_msgs.msg import CameraInfo, LaserScan
from nav_msgs.msg import Odometry
from std_srvs.srv import SetBool
from geometry_msgs.msg import Twist, Pose, Point, Quaternion
from packaging.version import parse

def euler_from_quaternion(quaternion):
    x = quaternion.x
    y = quaternion.y
    z = quaternion.z
    w = quaternion.w
    sinr_cosp = 2 * (w * x + y * z)
    cosr_cosp = 1 - 2 * (x * x + y * y)
    roll = np.arctan2(sinr_cosp, cosr_cosp)
    sinp = 2 * (w * y - z * x)
    sinp = max(-1.0, min(1.0, sinp))
    pitch = np.arcsin(sinp)
    siny_cosp = 2 * (w * z + x * y)
    cosy_cosp = 1 - 2 * (y * y + z * z)
    yaw = np.arctan2(siny_cosp, cosy_cosp)
    return roll, pitch, yaw

if parse(cv2.__version__) >= parse('4.7.0'):
    def local_estimatePoseSingleMarkers(corners, marker_size, mtx, distortion):
        marker = np.array([[-marker_size /2, marker_size / 2, 0],
                           [marker_size /2, marker_size / 2, 0],
                           [marker_size /2, -marker_size / 2, 0],
                           [-marker_size /2, -marker_size / 2, 0]],
                           dtype = np.float32)
```

```
        trash = []
        rvecs = []
        tvecs = []
        for c in corners:
            nada, R, t = cv2.solvePnP(marker, c, mtx, distortion, False,
cv2.SOLVEPNP_IPPE_SQUARE)
            rvecs.append(R)
            tvecs.append(t)
            trash.append(nada)
        return rvecs, tvecs, trash

class ArucoTarget(Node):
    _DICTS = {
        "apriltag_36h10" : cv2.aruco.DICT_APRILTAG_36H10,
        "4x4_100" : cv2.aruco.DICT_4X4_100,
    }

    def __init__(self, tag_set="apriltag_36h10", target_width=0.30):
        super().__init__('aruco_targetv2')
        self.get_logger().info(f'{self.get_name()} created')

        self._active = False

        # CONFIGURATION
        self.start_point = [-4, 0, 0] # Phase 1 target
        self.origin_point = [0, 0, 0] # Phase 3 target (Return Home)

        self.target_list = [0, 1, 2, 3, 4]  # Phase 2 targets
        self.target_index = 0

        # STATE MACHINE FLAGS
        self.reached_start = False
        self.target_close_counter = 0
        self.required_frames_close = 15
        self.last_log_time = time.time()
        self.mission_complete = False

        # Position tracking
        self._cur_x = 0.0
        self._cur_y = 0.0
        self._cur_theta = 0.0

        # ROS Setup
        self.declare_parameter('image', "/mycamera/image_raw")
        self.declare_parameter('info', "/mycamera/camera_info")
        self._image_topic = self.get_parameter('image').get_parameter_value().string_value
        self._info_topic = self.get_parameter('info').get_parameter_value().string_value
```

```
        self.create_subscription(Image, self._image_topic, self._image_callback, 1)
        self.create_subscription(CameraInfo, self._info_topic, self._info_callback, 1)
        self.create_subscription(LaserScan, "/scan", self._scan_callback, 1)
        self.create_subscription(Odometry, "/odom", self._odom_callback, 1)
        self.create_service(SetBool, '/startup', self._startup_callback)

        self.obstacle_detected = False
        self.avoid_bias = 0.0
        self.stop_completely = False

        self._cmd_pub = self.create_publisher(Twist, '/cmd_vel', 10)
        self._bridge = CvBridge()

        # ArUco Setup
        dict_name = ArucoTarget._DICTS.get(tag_set.lower(),
cv2.aruco.DICT_APRILTAG_36H10)
        if parse(cv2.__version__) < parse('4.7.0'):
            self._aruco_dict = cv2.aruco.Dictionary_get(dict_name)
            self._aruco_param = cv2.aruco.DetectorParameters_create()
        else:
            self._aruco_dict = cv2.aruco.getPredefinedDictionary(dict_name)
            self._aruco_param = cv2.aruco.DetectorParameters()
            self._aruco_detector = cv2.aruco.ArucoDetector(self._aruco_dict, self._aruco_param)
        self._target_width = target_width
        self._image = None
        self._cameraMatrix = None
        self._distortion = None

    def _startup_callback(self, request, resp):
        if request.data:
            self.get_logger().info("Startup received. Robot Active.")
            self._active = True
            resp.success = True
            resp.message = "Robot Active"
        else:
            self._active = False
            resp.success = True
            resp.message = "Robot Stopped"
        return resp

    def _odom_callback(self, msg):
        pose = msg.pose.pose
        self._cur_x = pose.position.x
        self._cur_y = pose.position.y
        _, _, yaw = euler_from_quaternion(pose.orientation)
        self._cur_theta = yaw

    def _short_angle(self, angle):
```

```
        if angle > math.pi: angle -= 2 * math.pi
        if angle < -math.pi: angle +=
2 * math.pi
        return angle

    def _compute_speed(self, diff, max_speed, min_speed, gain):
        speed = abs(diff) * gain
        speed = min(max_speed, max(min_speed, speed))
        return math.copysign(speed, diff)

    def _drive_to_point(self, target):
        """ Returns True if reached, Twist command if driving """
        goal_x, goal_y, goal_theta = target[0], target[1], target[2]

        x_diff = goal_x - self._cur_x
        y_diff = goal_y - self._cur_y
        dist = math.sqrt(x_diff**2 + y_diff**2)

        twist = Twist()

        if dist < 0.15:
            # Reached position
            return True, twist # Stop

        # Navigate
        heading = math.atan2(y_diff, x_diff)
        diff = self._short_angle(heading - self._cur_theta)

        if abs(diff) > 0.2:
            twist.angular.z = self._compute_speed(diff, 0.5, 0.2, 2.0)
        else:
            twist.linear.x = self._compute_speed(dist, 0.3, 0.05, 0.5)
            twist.angular.z = diff * 0.5

        return False, twist

    def _scan_callback(self, msg):
        ranges = msg.ranges
        valid_entries = [(r, i) for i, r in enumerate(ranges) if np.isfinite(r) and r > 0.03]
        if len(valid_entries) > 0:
            closest_dist, closest_index = min(valid_entries, key=lambda x: x[0])
            total_indices = len(ranges)
            mid_point = total_indices / 2
            center_width = total_indices * 0.3
            center_start = mid_point - (center_width / 2)
            center_end = mid_point + (center_width / 2)

            if closest_dist < 0.5:
```

```
            self.obstacle_detected = True
            if center_start < closest_index < center_end:
                self.avoid_bias = 0.0
                self.stop_completely = True
                self.get_logger().warn("OBSTACLE DETECTED: PLEASE MOVE ASIDE")
            elif closest_index < mid_point:
                self.avoid_bias = -0.3
                self.stop_completely = False
            else:
                self.avoid_bias = 0.3
                self.stop_completely = False
        else:
            self.obstacle_detected = False
            self.stop_completely = False
            self.avoid_bias = 0.0
    else:
        self.obstacle_detected = False
        self.stop_completely = False
        self.avoid_bias = 0.0

def _info_callback(self, msg):
    self._distortion = np.reshape(msg.d, (1,5))
    self._cameraMatrix = np.reshape(msg.k, (3,3))

def _image_callback(self, msg):
    if not self._active:
        return
    try:
        self._image = self._bridge.imgmsg_to_cv2(msg, "bgr8")
    except CvBridgeError:
        return

    twist = Twist()

    # EMERGENCY STOP
    if self.obstacle_detected and self.stop_completely:
        self._cmd_pub.publish(Twist())
        if self._image is not None:
            cv2.imshow('window', self._image)
            cv2.waitKey(3)
        return

    # --- LOGGING ---
    if time.time() - self.last_log_time > 2.0:
        if not self.reached_start:
            self.get_logger().info(f"Phase 1: Moving to Start (-4,0)...")
        elif self.target_index < len(self.target_list):
```

```
                self.get_logger().info(f"Phase 2: Searching/Chasing ID
{self.target_list[self.target_index]}")
            elif not self.mission_complete:
                self.get_logger().info("Phase 3: All Targets Found. Returning to Origin (0,0)...")
            else:
                self.get_logger().info("Mission Complete. Resting at Origin.")
            self.last_log_time = time.time()

        # ARUCO DETECTION
        grey = cv2.cvtColor(self._image, cv2.COLOR_BGR2GRAY)
        if parse(cv2.__version__) < parse('4.7.0'):
            corners, ids, rejected = cv2.aruco.detectMarkers(grey, self._aruco_dict)
        else:
            corners, ids, rejected = self._aruco_detector.detectMarkers(grey)

        frame = cv2.aruco.drawDetectedMarkers(self._image, corners, ids)

        # CONTROL LOGIC
        # 1. GO TO START POINT FIRST
        if not self.reached_start:
            reached, nav_twist = self._drive_to_point(self.start_point)
            if reached:
                self.get_logger().info("Start Position Reached. Beginning Search.")
                self.reached_start = True
                twist = Twist() # Stop
            else:
                twist = nav_twist

        # 2. SEARCH AND CHASE LOGIC
        else:
            if self.target_index >= len(self.target_list):
                # 3. RETURN TO ORIGIN (PHASE 3)
                reached_home, home_twist = self._drive_to_point(self.origin_point)

                if reached_home:
                    if not self.mission_complete:
                        self.get_logger().info("Returned to Origin (0,0). FULL MISSION
COMPLETE.")
                        self.mission_complete = True
                    twist = Twist() # Stop
                else:
                    twist = home_twist
            else:
                # SEARCHING FOR TARGETS
                current_id = self.target_list[self.target_index]
                target_found = False
                dist_to_target = 0.0
                target_angle = 0.0
```

```
            if ids is not None:
                ids_list = ids.flatten().tolist()
                if current_id in ids_list:
                    target_found = True
                    idx = ids_list.index(current_id)

                    if parse(cv2.__version__) < parse('4.7.0'):
                        rvec, tvec, _ = cv2.aruco.estimatePoseSingleMarkers(corners,
self._target_width, self._cameraMatrix, self._distortion)
                    else:
                        rvec, tvec, _ = local_estimatePoseSingleMarkers(corners,
self._target_width, self._cameraMatrix, self._distortion)

                    t = tvec[idx].flatten()
                    dist_to_target = math.sqrt(t[0]**2 + t[1]**2 + t[2]**2)
                    target_angle = math.atan2(t[0], t[2])

                    for r, t_vec in zip(rvec, tvec):
                        if parse(cv2.__version__) < parse('4.7.0'):
                            cv2.aruco.drawAxis(frame, self._cameraMatrix, self._distortion, r, t_vec,
self._target_width)
                        else:
                            cv2.drawFrameAxes(frame, self._cameraMatrix, self._distortion, r, t_vec,
self._target_width)

            if target_found:
                # CHECK DISTANCE
                if dist_to_target < 1.0:
                    self.target_close_counter += 1
                    if self.target_close_counter > self.required_frames_close:
                        self.get_logger().info(f"Target {current_id} Reached (<1m). Switching to
next.")
                        self.target_index += 1
                        self.target_close_counter = 0
                        twist = Twist() # Stop briefly
                    else:
                        # Keep chasing to be sure
                        steer = -1.0 * (target_angle + self.avoid_bias)
                        twist.angular.z = steer
                        twist.linear.x = 0.15
                else:
                    # FOUND BUT FAR -> CHASE
                    self.target_close_counter = 0
                    steer = -1.2 * (target_angle + self.avoid_bias)
                    twist.angular.z = steer

                    if abs(target_angle) < 0.5 and not self.obstacle_detected:
```

```
                    twist.linear.x = 0.25
                else:
                    twist.linear.x = 0.05
            else:
                # NOT FOUND -> SPIN SEARCH
                self.target_close_counter = 0
                twist.angular.z = 0.4 # Constant spin
                twist.linear.x = 0.0

        if self._image is not None:
            cv2.imshow('window', frame)
            cv2.waitKey(3)
        self._cmd_pub.publish(twist)

def main(args=None):
    rclpy.init(args=args)
    # Ensure target_width is correct for simulation
    node = ArucoTarget(tag_set="apriltag_36h10", target_width=0.20)
    try:
        rclpy.spin(node)
        rclpy.shutdown()
    except KeyboardInterrupt:
        pass

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