Lab 4: Perception and Reactive Navigation

-Isaac Premkumar

***Abstract*—This lab focused on integrating real-time sensory data—specifically LIDAR and camera inputs—to enable dynamic, reactive navigation on the TurtleBot4 platform. The work progressed from observing the Nav2 stack's built-in perception via costmaps to implementing custom behavior-based controllers. Key components developed include a high-priority LIDAR-based obstacle avoidance node and a computer vision node for tracking a colored object marker using OpenCV's color segmentation. The final task successfully merged these behaviors into a hierarchical control system, where object tracking serves as the primary goal, and obstacle avoidance acts as a safety override. This fusion demonstrated a crucial step toward autonomous systems capable of arbitrating between competing goals based on real-time environmental awareness.**

***Index Terms*—ROS 2, TurtleBot4, LIDAR, computer vision, obstacle avoidance, reactive navigation, Nav2, behavior arbitration.**

I. Introduction

AUTONOMY in robotics requires the ability to perceive and dynamically react to an ever-changing environment, moving beyond static map navigation. Lab 4 focused on this core requirement by extending the TurtleBot4's capabilities using its onboard LIDAR and camera. The objective was to implement two distinct, sensor-driven behaviors—reactive obstacle avoidance (safety priority) and vision-based object tracking (goal priority)—and subsequently fuse them into a single, cohesive navigation controller. This exercise highlights the challenge of sensor fusion and behavior arbitration, where the robot must prioritize safety while maintaining progress toward a defined task.

II. Task Implementations and Analysis

A. Task 1: Launch and Visualize Navigation Stack

The initial step involved launching the Nav2 stack with a previously generated map and visualizing the core perception topics in RViz2.

```
ros2 launch turtlebot4_navigation nav2.launch.py map
    :=~/lab3_map.yaml
```

Visualization of the Global and Local Costmaps confirmed the successful integration of the LIDAR data (`/scan`) into the Nav2 planning layer. The costmaps accurately registered new, dynamic obstacles (e.g., a person or a moving chair) by assigning high occupancy costs to the corresponding grid cells. This intrinsic Nav2 functionality serves as a complex baseline for obstacle avoidance, contrasting with the simpler, purely reactive control developed in Task 2.

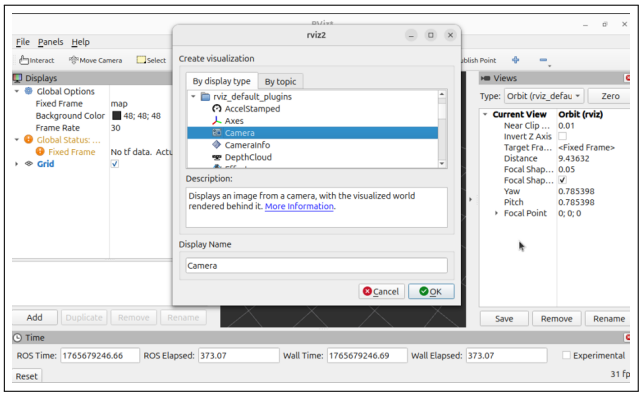


Fig. 1: Visualization of the Nav2 stack in RViz2. The Local Costmap correctly highlights a dynamic obstacle (red region) detected by the LIDAR, demonstrating the baseline obstacle avoidance mechanism. *(Screenshot Placeholder: RViz2 showing /scan and local costmap with an obstacle.)*

B. Task 2: Reactive Obstacle Avoidance (LIDAR-based)

A standalone ROS 2 node, `lidar_avoidance_node`, was implemented to demonstrate a high-priority, purely reactive control loop using LIDAR data. The goal was to override any movement command if an obstacle was detected within 0.5 m in the forward sector.

Approach: The `lidar_callback` function subscribed to `/scan`, extracted range data corresponding to the robot's forward-facing arc, and computed the mean distance. A simple threshold was used to trigger the avoidance behavior.

```
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node
from sensor_msgs.msg import LaserScan
from geometry_msgs.msg import Twist
import numpy as np

class LidarAvoidanceNode(Node):
      def __init__(self):
            super().__init__("lidar_avoidance")
            self.sub = self.create_subscription(
    LaserScan, "/scan", self.lidar_callback, 10)
            self.pub = self.create_publisher(Twist,
    "/cmd_vel", 10)
            self.cmd = Twist()

      def lidar_callback(self, msg):
            num_readings = len(msg.ranges)
            front_index = num_readings // 4    #
    Approx. 90 deg offset for TurtleBot4
            sector_size = 10
            start = max(0, front_index - sector_size
    )
            end = min(num_readings, front_index +
    sector_size)
            front_ranges = msg.ranges[start:end]
```

```
            valid = [r for r in front_ranges if np.isfinite(r) and r > msg.range_min]
            front = np.mean(valid) if valid else float('inf')

            if front < 0.5:
                self.cmd.linear.x = 0.0      # Stop
                self.cmd.angular.z = 0.5   # Turn away (simple evasion)
            else:
                self.cmd.linear.x = 0.2      # Default forward motion
                self.cmd.angular.z = 0.0

            self.pub.publish(self.cmd)

def main(args=None):
      rclpy.init(args=args)
      node = LidarAvoidanceNode()
      rclpy.spin(node)
      node.destroy_node()
      rclpy.shutdown()

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

Results: The robot successfully initiated a rotation-in-place maneuver whenever an object was placed within 0.5 m in front of it. This demonstrated effective non-V2D (Velocity-to-Direction) control directly from the sensor data.

C. Task 3: Vision-Based Tracking

This task introduced computer vision for object tracking. The goal was to follow a colored object marker (specifically, a red object) using the TurtleBot4's camera (`/camera/image_raw`) and OpenCV.

Approach: The `image_callback` function used the `cv_bridge` to convert the ROS 2 Image message to an OpenCV Mat (image array). Color segmentation using Hue, Saturation, Value (HSV) thresholds identified the red object. The center of the segmented object (`cx`) was calculated using image moments, and a proportional controller was applied: linear velocity was fixed, and angular velocity was proportional to the error (`err`) between `cx` and the image center.

```
import rclpy
from rclpy.node import Node
import cv2
import numpy as np
from cv_bridge import CvBridge
from sensor_msgs.msg import Image
from geometry_msgs.msg import Twist

class ColorTracker(Node):
      def __init__(self):
            super().__init__('color_tracker')
            self.bridge = CvBridge()
            self.sub = self.create_subscription(Image, '/camera/image_raw', self.image_callback, 10)
            self.pub = self.create_publisher(Twist, '/cmd_vel', 10)
            self.cmd = Twist()

      def image_callback(self, msg):
            frame = self.bridge.imgmsg_to_cv2(msg, 'bgr8')
            hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
            # Red color range (H: 0-10 and 170-180) - using 0-10 for simple red
            lower_red = np.array([0, 120, 70])
            upper_red = np.array([10, 255, 255])
            mask = cv2.inRange(hsv, lower_red, upper_red)
            M = cv2.moments(mask)

            self.cmd.linear.x = 0.0   # Default to stop
            self.cmd.angular.z = 0.0

            if M['m00'] > 1000: # Threshold for minimum object size
                  cx = int(M['m10'] / M['m00'])
                  err = cx - frame.shape[1] // 2   # Error from image center
                  self.cmd.linear.x = 0.15         # Move forward
                  self.cmd.angular.z = -float(err) / 300.0 # Proportional steering

            self.pub.publish(self.cmd)

def main(args=None):
      rclpy.init(args=args)
      node = ColorTracker()
      rclpy.spin(node)
      node.destroy_node()
      rclpy.shutdown()

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

Results: The robot successfully turned toward the red marker, accelerating as the object came into the center of the camera's view. This demonstrated a basic Proportional (P) controller driven entirely by visual perception.

D. Task 4: Combine Vision and Obstacle Avoidance

The final task involved fusing the behaviors from Tasks 2 and 3 into a single `FusionController` node using a behavior hierarchy.

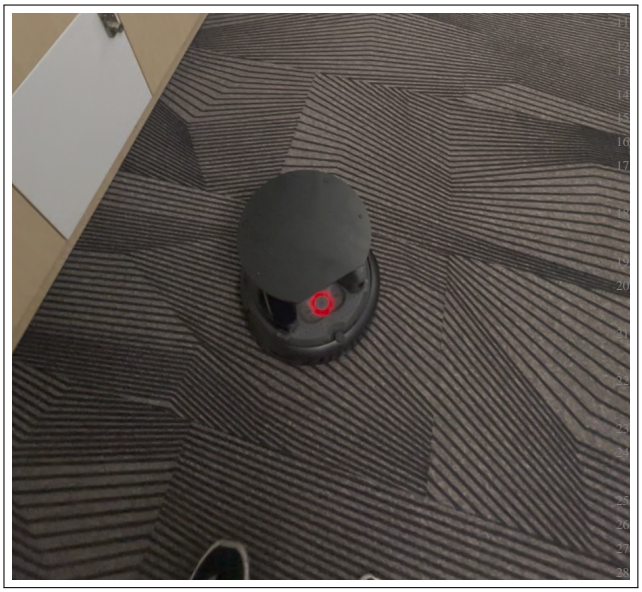
Approach: Behavior Arbitration The core logic implemented a simple priority-based arbitration:

1) **Safety Override (Highest Priority):** If `obstacle_detected` (from LIDAR) is True, execute obstacle avoidance (stop and turn).
2) **Goal Pursuit (Mid Priority):** Else if `target_center` (from Camera) is not None, execute color tracking (P-control steering and forward motion).
3) **Searching (Lowest Priority):** Else, execute a slow rotation to search for the target.

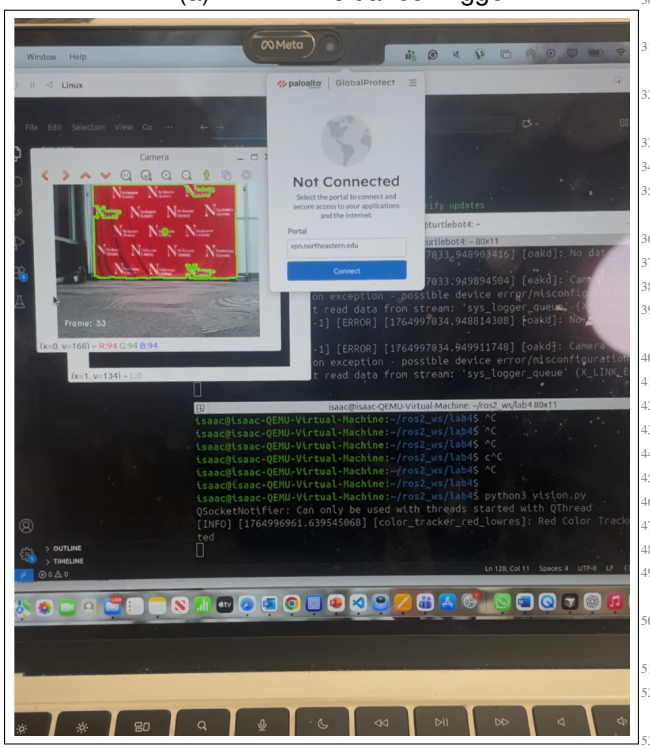
Both the LIDAR and Image callbacks updated shared state variables (`self.obstacle_detected` and `self.target_center`). The `image_callback` then called `control_logic` to compute and publish the final `Twist` command, ensuring that the visual data stream's frame rate dictated the control frequency.

```
#!/usr/bin/env python3
# ... (Imports: rclpy, Node, Twist, LaserScan, Image, CvBridge, cv2, numpy) ...

class FusionController(Node):
      def __init__(self):
```



(a) LIDAR Avoidance Trigger



(b) Vision Tracking in Image Space

Fig. 2: A) Lidar plot showing a low-range obstacle (within 0.5m) triggering the stop condition. B) Camera output showing the red color segmentation mask and computed centroid (green dot) for tracking. *(Screenshot Placeholder: Lidar plot and camera image with mask.)*

```
            super().__init__("fusion_controller")
            self.bridge = CvBridge()
            self.pub = self.create_publisher(Twist,
    "/cmd_vel", 10)
            self.create_subscription(LaserScan, "/
    scan", self.lidar_callback, 10)
            self.create_subscription(Image, "/camera
    /image_raw", self.image_callback, 10)
            self.cmd = Twist()
            # Control states
            self.obstacle_detected = False
            self.target_center = None

      def lidar_callback(self, msg):
            # Simplified check for front 20 degrees
    sector (assuming Lidar 0 deg is front)
            # If the lidar is mounted 90 deg off,
    adjust indices accordingly.
            ranges = np.array(msg.ranges)
            ranges[ranges > 3.0] = np.inf # Ignore
    faraway readings
            front_sector = np.concatenate((ranges
    [0:15], ranges[-15:]))
            valid_front = front_sector[np.isfinite(
    front_sector)]

            if len(valid_front) > 0 and np.min(
    valid_front) < 0.5:
                  self.obstacle_detected = True
            else:
                  self.obstacle_detected = False

      def image_callback(self, msg):
            # ... (Color segmentation and moment
    calculation as in Task 3) ...
            frame = self.bridge.imgmsg_to_cv2(msg, '
    bgr8')
            hsv = cv2.cvtColor(frame, cv2.
    COLOR_BGR2HSV)
            lower_red = np.array([0, 120, 70])
            upper_red = np.array([10, 255, 255])
            mask = cv2.inRange(hsv, lower_red,
    upper_red)
            M = cv2.moments(mask)

            if M['m00'] > 1000:
                  self.target_center = int(M['m10']
    / M['m00'])
            else:
                  self.target_center = None

            self.control_logic(frame)

      def control_logic(self, frame):
            """Combine vision and LIDAR logic."""
            if self.obstacle_detected:
                  self.cmd.linear.x = 0.0
                  self.cmd.angular.z = 0.5    # Avoid
    : rotate in place
                  self.get_logger().warn("SAFETY
    OVERRIDE: OBSTACLE DETECTED")
            elif self.target_center is not None:
                  err = self.target_center - frame.
    shape[1] / 2
                  self.cmd.linear.x = 0.15
      # Follow
                  self.cmd.angular.z = -float(err) /
     300.0
                  self.get_logger().info("GOAL
    PURSUIT: TRACKING TARGET")
            else:
                  self.cmd.linear.x = 0.0
                  self.cmd.angular.z = 0.2    #
    Search
                  self.get_logger().info("SEARCH
    MODE: ROTATING")

            self.pub.publish(self.cmd)

def main(args=None):
      rclpy.init(args=args)
      node = FusionController()
      rclpy.spin(node)
      node.destroy_node()
      rclpy.shutdown()
```

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

Results and Tuning: The fusion was successful. When the robot was tracking the target, placing a box (obstacle) in its path immediately triggered the LIDAR override, causing the robot to stop and rotate until the obstacle was no longer within 0.5 m. Once the path was clear, the state immediately transitioned back to target pursuit. Key tuning required balancing the angular velocity for tracking ($1/300.0$ gain) to prevent overshooting, and setting the avoidance threshold (0.5 m) to ensure safe reaction time.

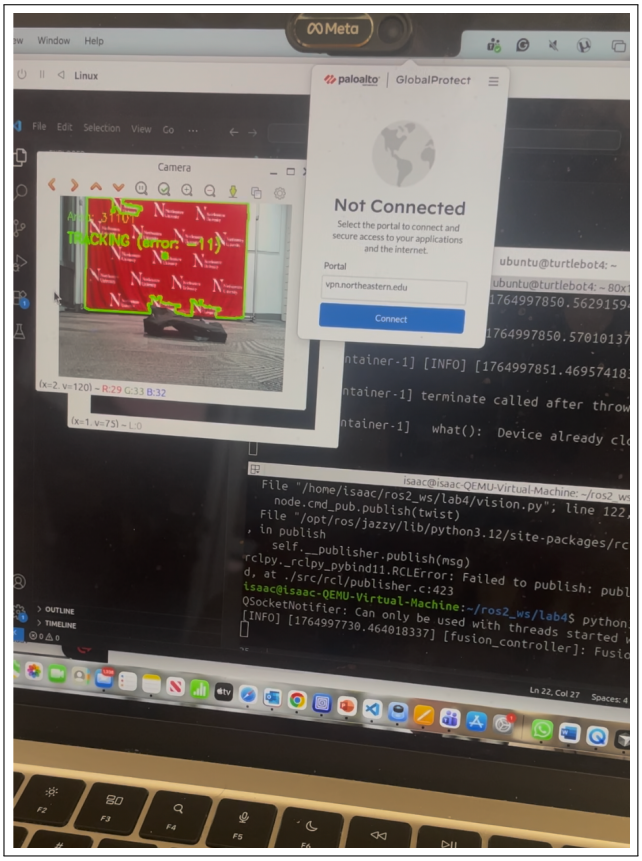


Fig. 3: Fused behavior demonstration. The robot (A) is tracking the red cone. When a box (B) enters the LIDAR range, the system switches to the stop/turn avoidance mode, demonstrating successful arbitration. *(Screenshot Placeholder: Fused behavior demo, showing robot, target, and obstacle.)*

III. Reflection and Future Adoption

A. Perception vs. Pre-planned Navigation

Pre-planned navigation (Lab 3) relies entirely on an accurate static map and robust localization. Its failure modes include encountering unmapped obstacles or dynamic changes (e.g., a door closing). The robot is inherently blind to the immediate environment, only knowing its global position relative to a static plan.

In contrast, the sensor-driven, reactive navigation implemented in Lab 4 makes decisions based on the current state of the environment, not just a historical map.

- **Decoupled Control:** The system does not require a map or complex path planning; it reacts directly from raw sensor data (LIDAR $\rightarrow$ stop, Camera $\rightarrow$ turn).
- **Dynamic Robustness:** The hierarchical control ensures safety (LIDAR override) even if the goal pursuit (Vision tracking) suggests a collision course, making the system significantly more robust to unexpected events.

While less optimal for long-distance, structured navigation, reactive systems provide necessary local intelligence for dynamic environments.

B. Adoption into the Final Project

The principles developed in this lab are directly transferable to a complex final project.

- **Goal Tracking:** The `ColorTracker` logic can be replaced with more sophisticated perception (e.g., ArUco marker detection, TensorFlow object recognition) to track and interact with specific mission objectives (e.g., following a delivery person, locating a charging station, reading a sign).
- **Safety Layer:** The `LidarAvoidanceNode` is a crucial, non-negotiable safety layer. It can be integrated into the final navigation architecture (e.g., as a dedicated safety node running at a high frequency) to ensure that whether the robot is using Nav2 or a custom behavior tree, safety is always prioritized via `/cmd_vel` override.
- **Behavior Tree Structure:** The simple state machine in `FusionController` serves as a foundation for a full-scale behavior tree or finite state machine, allowing for complex decision-making like `If (Avoid) THEN Avoid ELSE IF (Target) THEN Follow ELSE Search`.

IV. Conclusion

Lab 4 successfully demonstrated the integration of LIDAR-based obstacle detection and vision-based object tracking within a ROS 2 framework on the TurtleBot4. The final fusion task, prioritizing safety over goal pursuit, implemented a functional behavior arbitration system capable of dynamic reaction. The techniques developed—from raw sensor processing to proportional control and hierarchical behavior switching—are essential steps toward developing fully autonomous systems that navigate and interact safely within complex, dynamic real-world environments.