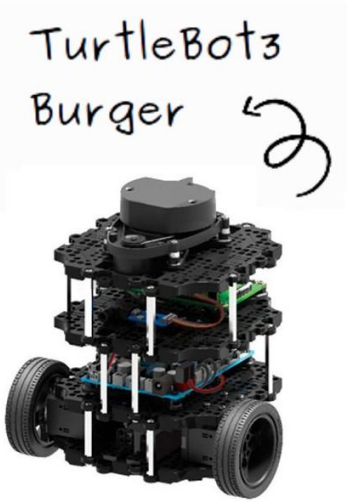
# Introduction to Robotics: Evade and Find.

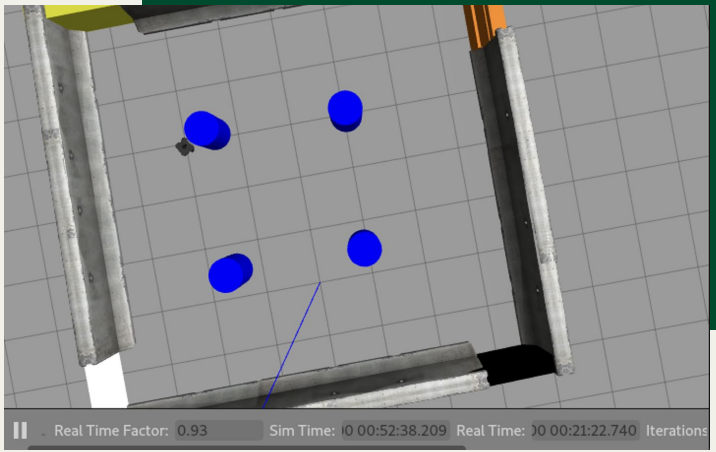



Haider Gilani
Misbah Uddin

# Overview


01 The Problem
02 Solution
03 CTrackNode
04 OANode
05 Testing

# The Problem




## BACKGROUND

Robots often face the challenge of efficiently navigating and locating specific targets while avoiding obstacles. Our robot must locate color-coded targets, and navigate to those targets without colliding with the obstacles obstructing the robot's path. This relies on accurate target detection and tracking which rely on visual sensors that are prone to noise and environmental interference.
Of course, one could imagine may real-world scenarios in which this task is essential, such as for Mars rovers.

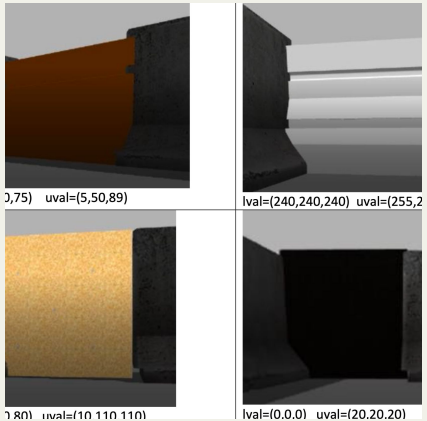
We implemented our solution using two ROS nodes that collaborate to achieve visual navigation and obstacle avoidance to carry out the task.

Obstacle Avoidance Node

Color Tracking Node

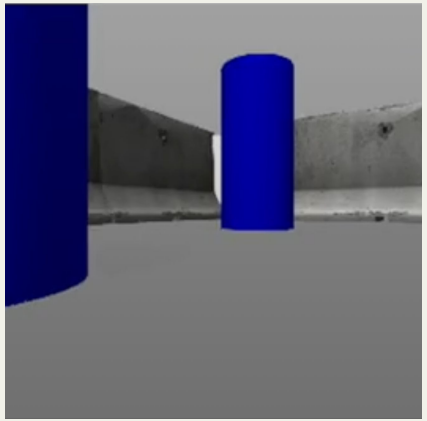
## Color Tracking Node (CTrackNode)

- Captures real-time images from the robot's camera.
- Identifies color-coded targets sequentially using OpenCV techniques (cv2.inRange and cv2.moments).
- Computes linear and angular velocity commands to align and approach the detected target.
- Publishes velocity commands to the /safe_cmd_vel topic for obstacle avoidance integration.
- Saves annotated images (e.g., goal0.jpg) once the target is reached, recording goal number, position, and elapsed time.




## Obstacle Avoidance Node (OANode)

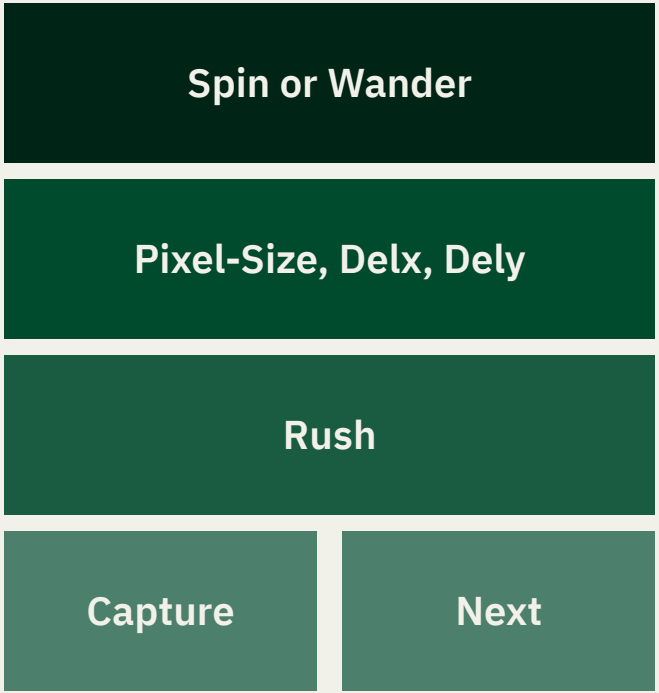
- Subscribes to the /safe_cmd_vel topic for commands from CTrackNode.
- Processes laser scanner data from the /scan topic to detect obstacles on the left, right, or both sides.
- Adjusts or overrides velocity commands based on detected obstacles:
  - Slows down and steers away from detected obstacles.
  - Stops and reorients if obstacles are on both sides.
- Publishes velocity commands to the /cmd_vel topic for the robot's movement.



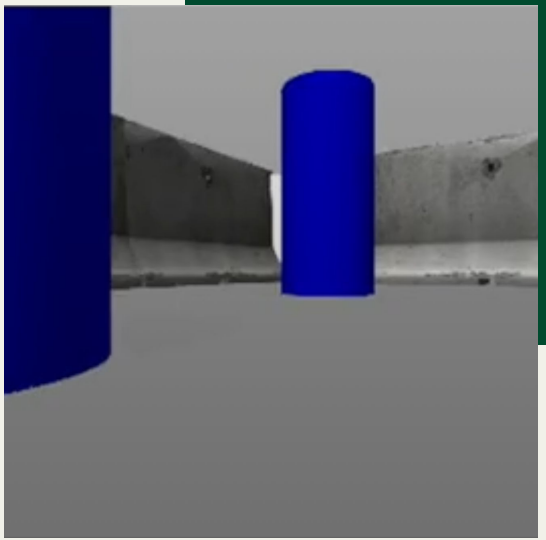
Color Tracking Node (CTrackNode)

- This node loops through an array of color ranges for Orange, White, Yellow, and Black
  - ➢ A timer that notes when this code starts to run is implemented before the loop

- Every iteration includes the following
  - Spinning in a circle to find a **significant** number of pixels that are within each color range
    - Wandering if a spin produces no significant finding
  - Checking the Angular and Horizontal distances to center the pathway
  - Move towards the colored wall and take a screenshot



# Spin and Wander



### Spin

By default, spins counterclockwise until it sees a significant number of colored pixels  at 0.4 m/s

### Wander

If the Bot spins for about 400 degrees (rough estimate), random values for acceleration and velocity and added to shift the bot to a different spot

Useful if its stuck behind a pillar or sees no colors at all

### Cycle Counter

Variable that counts up to 200 which is about 20 seconds in real time for the bot to spin

After breaching 200, it gets set to -80

About 5 seconds of wandering to move the bot in a different spot and spin to scan for colors again

# Pixel Density, Delx, Dely

### Pixel Density

As the bot spins, it looks for a heavy amount of pixels in a certain area to focus on before running towards it

This is to differentiate between “noise”

M[‘m00’] > 1,000,000

### Delx

Angular Adjustment

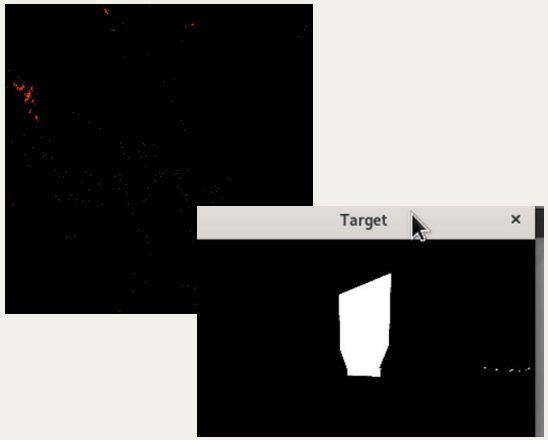
- Distance from central point on x-axis or width of the wall that the robot turns to
  - The bot will make small adjustments left or right so this value is close enough to 0 as possible / always looking at the wall

### Dely

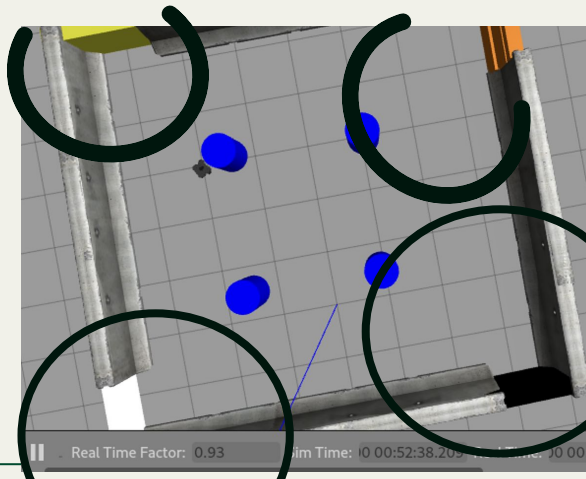
Linear Adjustment

- Takes the distance between the bot and a centered point on the wall from the height
  - In this case, scans are done 60% down from the top of the wall

Brings the bot to a certain range in order to take a screenshot







# Found == True

**The bot's linear position was found using dely, a variable for the distance from a central point on the y axis**

- **Once the bot was within a range of 310 and 340**
  - **Too high meant the bot was too close and needed to move back**
  - **A value too low means the bot needs to keep moving forward**
    - **There were cases where dely was negative,**
      - **Uses the same code that would "reverse" the bot**

**Lvel = (+ or - based on value of dely)Speed * dely/h**

1. Reset Cycle Counter to move on to the next Color
2. Record the current time elapsed - start
   a. To print out time on the screenshot that will be taken
3. Print out the time elapsed and the target color,
   a. Each iteration has a format of [color_name, lower, upper], use color_name in this case
4. Save as an individual jpeg file

# Obstacle Avoidance Node (OANode)

## Detection Mechanism

The OANode uses laser scan data from the /scan topic to detect obstacles.

- Divides the LIDAR readings into three regions:
  - Left Sector: Angles between -30° and 0°
  - Right Sector: Angles between 0° and 30°.
- Iterates through each sector to check distances to objects:
  - If an object is closer than 0.5 meters in a sector, it flags the corresponding direction (left or right) as obstructed.
- Resets flags (obstacle_left, obstacle_right) for every new scan to ensure up-to-date obstacle detection.

## Action for each case

Obstacle on the Left:

- Slows down forward movement to 50% of the current speed.
- Turns the robot right (adjusts angular velocity: -0.2).

Obstacle on the Right:

- Slows down forward movement to 50% of the current speed.
- Turns the robot left (adjusts angular velocity: 0.2).

Obstacles on Both Sides:

- Stops forward motion (linear.x = 0.0).
- Rotates in place slowly (angular.z = 0.3) to attempt reorientation and find a clear path.

## Innovative Choices

Dynamic Adjustment of Velocity:

- Instead of stopping, the robot slows down and adjusts direction, maintaining smoother movement and avoiding sudden halts.

Non-Blocking Behavior:

- When obstacles are detected on both sides, the robot rotates in place instead of freezing, ensuring progress toward a solution.

# Testing