The Galbraith Memorial Mail Robot - Report

By: Alexander Wang [1008076172], Gal Cohen [1007757768]

1. Introduction

This report details the development of a package delivery system for a TurtleBot 3 Waffle Pi robot as a part of the ROB301 Introduction to Robotics course at the University of Toronto. The primary objective of "The Galbraith Memorial Mail Robot" project is to simulate the process of mail delivery within a specific environment. This environment is based on the hallways of the Faculty of Applied Science and Engineering's Galbraith Building, where the robot is required to navigate a closed-loop path and deliver mail to 11 possible offices shown in Figure 1. These offices are represented by 4 different colour patches (orange, green, yellow, blue), connected with a track made from white tape. The robot is to mimic delivery of a parcel of mail by stopping at the desired office, rotating 90° to the left, pausing and then rotating back to continue.

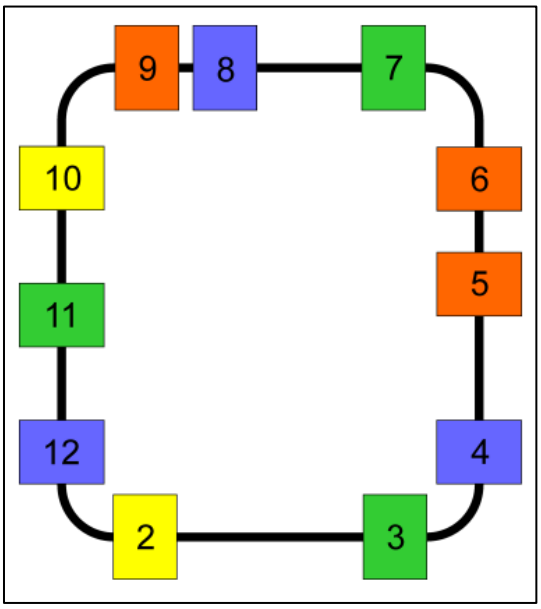


Figure 1: Topological Map of the Closed-loop Path

This report aims to document the entire process of designing, implementing, and testing the autonomous package delivery robot. It will cover the technical details of the control system, the challenges faced, and the solutions implemented. The report will also reflect on the project's success in meeting its objectives and suggest potential improvements for future iterations.

2. Robot Platform

2.1 Specifications

The robot platform chosen for this project is the TurtleBot 3 Waffle Pi. This platform is accompanied with an OpenCR1.0 (ARM Cortex®-M7) microcontroller and a Raspberry Pi 3 for higher-level functions. Its mobility is driven by DYNAMIXEL XM430-W210 actuators, allowing for precise control in velocity, torque, and position. Navigation and environmental sensing are facilitated by the Raspberry Pi Camera Module v2.1, providing RGB data for line-following and localization tasks.

2.2 Component Functions

The Raspberry Pi gathers sensor input from its camera and relays the output commands from the Remote PC to the OpenCR1.0 board. Receiving the commands, the OpenCR1.0 board then outputs the corresponding motor control signals to the DYNAMIXEL actuators to move and rotate the robot.

This platform's set of sensors, motors, and onboard computers renders it ideal for applications requiring real-time processing and autonomous navigation. This makes the TurtleBot 3 Waffle Pi an optimal choice for the autonomous package delivery system developed in this project.

3. Solution Strategy

3.1 Requirements Table

The following table outlines the fundamental requirements that the autonomous mail delivery robot must satisfy for the project to be considered successful.

Requirement ID	Description	Source
R1	Robot must use TurtleBot 3 Waffle Pi with ROS.	Equipment & Software
R2	The robot must process camera data to discern between office doorways and the path.	Map, Subsystem Design
R3	Robot must stop at the desired office, rotate 90 degrees, pause, then continue.	Map, Subsystem Design
R4	Develop a Bayesian-localization algorithm and simulate it to demonstrate probability convergence.	Deliverable 1
R5	Demonstrate full-route line-following without localization. Minimize the number of corrections required.	Deliverable 2
R6	Complete the Bayesian-localization node and demonstrate accurate state probability convergence on the robot.	Deliverable 3
R7	Execute successful mail delivery demonstration. Require 2 or less manual corrections to complete all deliveries.	Deliverable 4

3.2 Tasks Strategy

To complete each of the requirements listed above, we will need to tackle the following tasks:

1. Line Following: Implementing a PID control mechanism to ensure accurate line following, the robot will traverse the designated track without requiring corrections.
2. Office Recognition: Through color sensors, the robot will identify color-coded patches representing offices. It should be able to distinguish the different delivery points along the route.
3. Localization: Utilizing a Bayesian localization algorithm, the robot will localize itself within the environment by correlating the recognized color patches with the known map, ensuring it understands its location relative to the delivery points.
4. Mail Delivery: Finally, the robot will navigate to and deliver packages to the recognized offices, performing the required "delivery" motions.

The progression of the completion of these tasks can be found in the flow chart below.

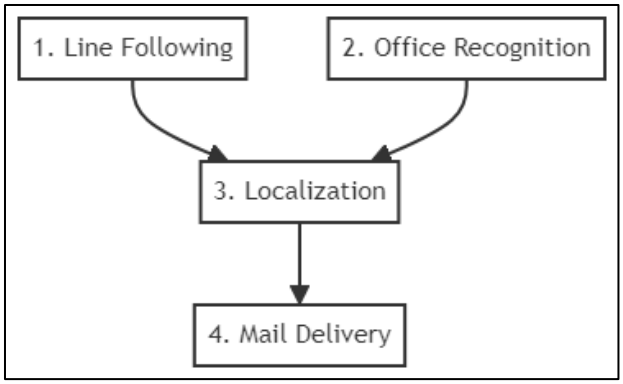


Figure 2: Flowchart of the Solution Strategy

4. Design Methodology

In the development of our Turtlebot 3 Waffle Pi robot for the Galbraith Memorial Mail Robot project, we employed an advanced design methodology that combined sophisticated control strategies with Bayesian localization techniques. This methodology was pivotal in ensuring the robot's accurate and efficient navigation for mail delivery within a specified environment.

4.1 Line Following

The foundation of our control methodology was the implementation of a PID controller, captured within the *follow_the_line* method, designed for line following. The PID controller was finely tuned to balance response speed and system stability, effectively reducing overshoot and ensuring smooth tracking of the line. A distinctive feature of this PID controller was its dynamic adjustment capability. When the error, calculated as the deviation from the desired line index, fell below a set threshold (50 in this case), the integral term was excluded from the control action. This approach significantly enhanced control precision, particularly in scenarios where the robot at the end of a turn into an office, as it would enter in a trajectory parallel to the office and could thus keep going straight until sensing the line again.

The *straight_line* method enabled the robot to move in a straight line upon detecting an office. Additionally, after executing 90-degree turns, the robot employed the *correct_left* function for corrective maneuvers. This function dynamically calculated the necessary angular correction based on the robot's recent angular velocities during the turn, ensuring accurate alignment and orientation post-turn. Using a queue for the angular velocities during the last 6.66 seconds, we took the average and made the correction proportional to it if it is above some threshold, meaning a turn likely happened. This is checked at the beginning of every office so that the robot will move in parallel to the office.

4.2 Office Recognition

A crucial aspect of our localization strategy was the color-based measurement model. We used RGB and HSV color data to estimate the robot's state (last offices visited). The *colour_callback* function was responsible for processing the color data from the camera, updating the robot's current color perception. The observed colors were classified into predefined categories using the *estimate_color* method, which relied on specific HSV thresholds. This color classification was important to the *measurement_model* method, which computed the probability of the robot being in each state based on the observed color.

By using both hue and saturation, we were able to adapt to the specific lighting conditions of the environment, particularly the yellow light in the lab. This lighting tended to skew the color detection of the

white line towards yellow-orange hues, posing a challenge in accurately identifying the distinct colors of the offices. To address this, we took advantage of the saturation component of the HSV. The white line had a considerably lower saturation compared to the orange color of the offices. Thus, we were able to effectively distinguish between colors that appeared similar in hue but varied in saturation. This approach allowed us to reliably detect colors.

4.3 Localization

The *BayesLoc* class is the core of our localization strategy, handling the Bayesian localization logic. We initialized our probability distribution to reflect a uniform likelihood across all potential office locations, representing an unbiased starting point for the localization process. The *state_model* method then updated the state prediction based on the assumption of a 100% probability of the robot moving forward one office at each update. This simplification, while effective, did not account for potential slippage or missed movements. Thus, we had to ensure that our color measurement functionality could ensure that this assumption is true. We increased the frequency of our controller to publish more precise angular velocities based on our line and color measurements. After testing and fine tuning, we found that taking a measurement of the office color after 100 consecutive measurements of the same color (1.66 [s] at 60 Hz), was both reliable, reproducible, and allowed us to be confident we are detecting the actual color of the office.

4.4 Mail Delivery

Our design methodology integrated both the control and localization aspects. In each iteration, the robot followed the line using PID control until a color was confidently detected, indicating that it's at a new office. This triggered a switch to straight-line motion and later on, an update in localization probabilities, as the color measurement became more confident. When the robot's confidence in its state exceeded a threshold of 0.7, it executed the package delivery maneuver, showcasing the application of control and localization in a practical scenario.

The *deliver_package* function does the sequence of actions for delivering a package. It is called when the robot is confident it is in the correct office (from the localisation). It then executes a 90-degree left turn by setting its linear velocity to zero and angular velocity to 1.5708, or half pi, holding this turn for 60 iterations, or 1 second. After completing the turn, the robot pauses for a simulated package delivery, and then turns right, like the left turn but with an angular velocity of -1.5708, again for 60 iterations.

5. Proof-of-Concept Demonstration

The final proof-of-concept demonstration of the autonomous mail delivery system was conducted to evaluate the robot's performance in executing the designated tasks. Shown in Figure 3, the robot undergoing mail delivery to its targeted office location. The line-following task was executed flawlessly, with the robot maintaining its course autonomously without the need for any manual corrections. The accuracy of the robot's localization was impressive, as shown by the data plots provided in the Appendix 8.1. The robot achieving the correct convergence above our predefined confidence threshold (>70%) within four iterations, demonstrating a rapid and efficient localization process. Most importantly, the mail delivery mechanism operated without issues. The robot demonstrated the ability to accurately discern office locations intended for mail drop-off and executed the delivery sequence accordingly.

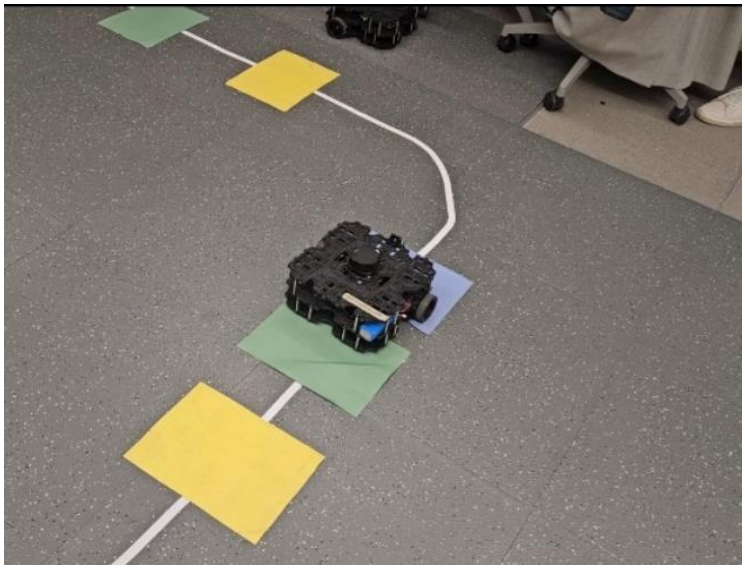


Figure 3: Mail Delivery Robot in Action

6. Potential Improvements

Integrating additional sensors such as the inertial measurement unit or an accelerometer could significantly enhance the state estimation and control processes of the Turtlebot. These sensors, by providing data on the robot's orientation and acceleration, could lead to a more advanced understanding of the robot's movements, particularly in handling uncertainties like slippage or irregularities in the robot's path. By using odometry, such data would be valuable in refining the probability distributions used for localization, ensuring a more accurate and reliable determination of the robot's position within the environment. This enhancement would address one of the current limitations of the system, the assumption of 100% probability of moving forward one tile in every localization update, and thereby improve the overall navigation accuracy of the robot.

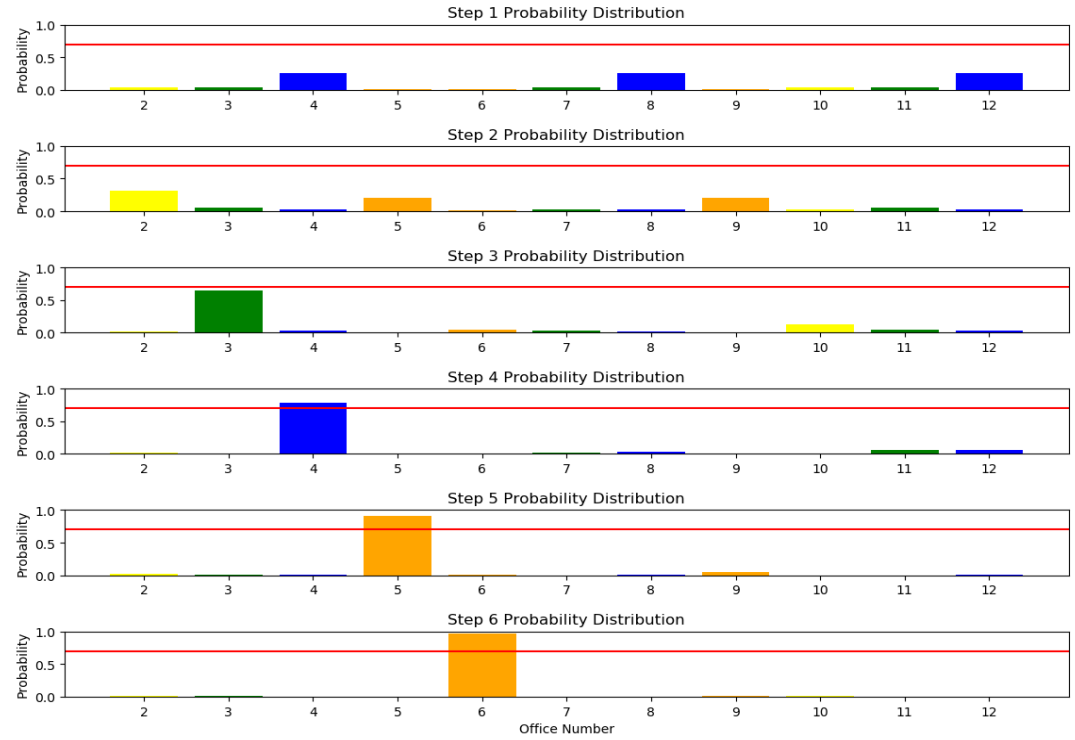
On another note, the application of computer vision techniques for office detection could be used to upgrade the robot's localization and line tracking capabilities. By leveraging the robot's onboard camera, advanced image processing algorithms could be employed to detect and differentiate between the various office colors more precisely, as well as different shapes, like squares for offices, and a short rectangle for the line. This enhancement in the *colour_callback*, *estimate_color* and *camera_callback* functions would allow for a more nuanced interpretation of the visual data, leading to an improvement in the accuracy of the robot's location estimation, as well as anticipating a transition to and from an office, and detecting line curvature to handle upcoming turns. The ability to reliably identify office colors, even under varying lighting conditions or at different angles, would substantially reduce the ambiguity in the robot's environment.

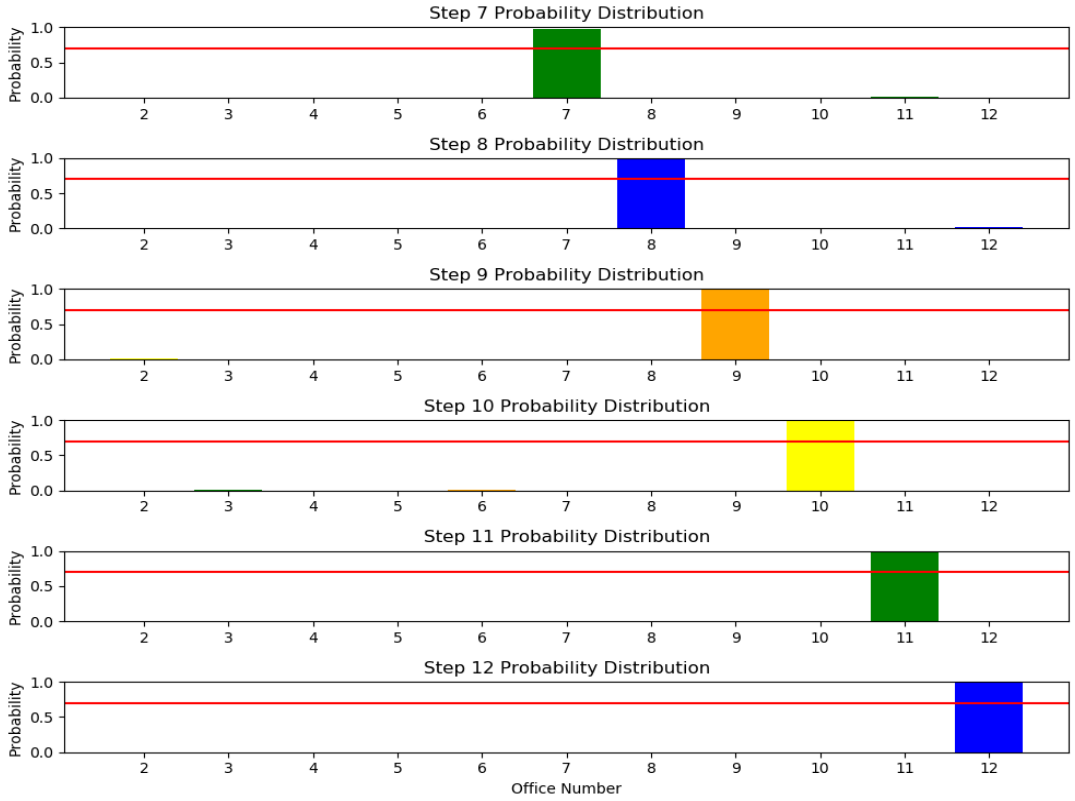
7. Conclusion

In conclusion, this project has successfully demonstrated the viability of an autonomous mail delivery system using the TurtleBot 3 Waffle Pi. Through meticulous design, implementation, and testing, the robot system accomplished its objectives with precision and efficiency. The integration of PID control for line following, color recognition for office detection, and Bayesian localization for accurate navigation culminated in a robust delivery mechanism. The final demonstration provided evidence of the system's reliability and the potential for further advancements. Future work could explore improving the robot's sensory and office recognition capabilities. This project not only serves as a significant learning endeavor in design and testing but also contributes to our understanding of the capabilities of embedded systems.

8. Appendix

8.1 Bayesian Localization Plots





8.2 Code

```
##!/usr/bin/env python
import rospy
import math
from geometry_msgs.msg import Twist
from std_msgs.msg import String, Float64MultiArray, UInt32
import numpy as np
import colorsys
import statistics
import matplotlib.pyplot as plt

# Controller Class
class Controller(object):
    def __init__(self):

        # Publish motor commands
        self.cmd_pub = rospy.Publisher("cmd_vel", Twist, queue_size=1)
        self.index = 0
        self.twist = Twist()
        self.line_filter = []

        # PID Controller Variables
        self.integral = 0
        self.derivative = 0
        self.lasterror = 0
        self.angular_v = 0
        self.angular_filter = []

    def camera_callback(self, msg):
        """
        Callback for line index.
        """

        self.index = msg.data

        # print("raw:" + str(self.index))

        # Line Filter
        if len(self.line_filter) > 5:
            self.line_filter.pop(0)
        self.line_filter.append(self.index)

        # Set detected line index based on Line Filter
        self.index = float(statistics.mean(self.line_filter))

        # print("filtered:" + str(self.index))

    def follow_the_line(self):
        """
        Line-following PID Controller.
        """

        desired = 320
        actual = self.index  # Current sensor reading
```

```
        error = desired - actual

         k_p = 1/340 # 1/350 had overshoot, 1/400 was good with k_p alone # Perhaps
reduce slightly to have less aggressive
        k_i = 4e-6 # mult*(1/10**8)
        k_d = 1e-2 # If you have overshoot from above, increase to increase damping

        # PID calculations
        self.integral = self.integral + error
        self.derivative = error - self.lasterror

        # print(self.integral)

        # Limit integral error to +/-20000
        self.integral = max(min(self.integral, 20000), -20000)

        # print((k_p*error), (k_i*self.integral), (k_d*self.derivative))

        # Calculate Angular Velocity
        if abs(error) >= 50:
                        self.angular_v   =   ((k_p*error)   +   (k_i*self.integral)   +
(k_d*self.derivative))*1.55
        else: # Remove Integral Term if error is low
            self.angular_v = ((k_p*error) + (k_d*self.derivative))*1.55

        # Limit Angular Velocity to +/-0.4
        self.angular_v = max(min(self.angular_v, 0.4), -0.4)

        # print(self.angular_v)

        # Setting the velocities
        self.twist.linear.x = 0.04
        self.twist.angular.z = self.angular_v

        # Store the last set of Angular Velocities
        if len(self.angular_filter) > 400:
            self.angular_filter.pop(0)
        self.angular_filter.append(self.angular_v)

        # Publishing the command
        self.cmd_pub.publish(self.twist)

        # Storing the last error for the next derivative calculation
        self.lasterror = error

        # print(actual, angular_v, self.integral, self.derivative)

    def straight_line(self):
        """
        When detecting an Office, move in a straight line.
        """

        self.twist.linear.x = 0.05
        self.twist.angular.z = 0
```

```
        self.cmd_pub.publish(self.twist)

    def correct_left(self):
        """
          When entering a new Office, detect if a 90 degree turn was made with angular_filter.
        If so, provide left turning correction scaled to the sharpness of the 90 degree turn.
        """

         av = float(statistics.mean(self.angular_filter)) # Find the Average of the last set of angular velocities
        print(av)

         # If a 90 degree turn was performed recently, provide additional left turn correction to Turtlebot
        if av > 0.05:

            for i in range(8): # Provide correction
                self.twist.linear.x = 0.05
                self.twist.angular.z = min(1.5 * av * 10, 2.25) # Scale correction by sharpness of the turn detected. Cap correction to 2.25
                self.cmd_pub.publish(self.twist)

                rate.sleep()

            self.angular_filter = [0] * 400 # Reset stored angular velocity values

    def deliver_package(self):
        """
        Perform package delivery function at desired office
        """

        # Go straight for a duration into office
        for i in range(60):
            self.straight_line()

            rate.sleep()

        # Turn left 90 degrees
        for i in range(60):
            self.twist.linear.x = 0
            self.twist.angular.z = 1.5708
            self.cmd_pub.publish(self.twist)

            rate.sleep()

        # Pause to Deliver Package
        for i in range(60):
            self.twist.linear.x = 0
            self.twist.angular.z = 0
            self.cmd_pub.publish(self.twist)

            rate.sleep()
```

```
        # Turn right 90 degrees
        for i in range(60):
            self.twist.linear.x = 0
            self.twist.angular.z = -1.5708
            self.cmd_pub.publish(self.twist)

            rate.sleep()


# Bayesian Localizer Class
class BayesLoc:
    def __init__(self, p0, colour_map):

        # Sensor Data Variables
             self.colour_sub  =  rospy.Subscriber("mean_img_rgb",  Float64MultiArray,
self.colour_callback)
        self.line_sub = rospy.Subscriber("line_idx", UInt32, self.line_callback)

        # PID Controller
        self.controller = Controller()

        # Bayesian Localization Variables
        self.num_states = len(p0)
        self.colour_map = colour_map
        self.probability = p0 # Current State Variable
        self.state_prediction = np.zeros(self.num_states)

        # Colour Variables
        self.rgb = [] # RGB Value
        self.hsv = [] # HSV Value
        self.colour_filter = [] # Colour Filter (for Controller)
        self.cur_colour = None  # Most recent measured colour (for Controller)

        self.colour_measurments = [] # Colour Filter (for Localization)

        self.colour_codes = [
            [244, 160, 131],
            [163, 183, 163],
            [186, 178, 197],
            [190, 179, 158],
            [150, 150, 150]
        ] # Reference colour RGBs (for Measurement Model)

    def line_callback(self, msg):
        """
        Callback for line index.
        """

        self.controller.camera_callback(msg)

    def colour_callback(self, msg):
        """
         callback function that receives the most recent colour measurement from the
camera.
        """
```

```
        self.rgb = np.array(msg.data)  # [r, g, b]
             self.hsv   =  colorsys.rgb_to_hsv(self.rgb[0]/255.0,  self.rgb[1]/255.0,
self.rgb[2]/255.0)
        self.hsv = [self.hsv[0]*360.0, self.hsv[1]*100.0, self.hsv[2]*100.0]
        # print(self.hsv)
        self.estimate_color()


    def estimate_color(self):
        """
        Estimate currently observied office colour based on HSV values
        """

        if 0 <= self.hsv[0] < 20:           # Use saturation to differentiate between
orange or line (since they look similar in hue due to warm coloured laboratory lights)
            if 20 <= self.hsv[1] < 80:
                color = 0
            else:
                color = 4
        elif 20 <= self.hsv[0] < 25:          # orange 5-25
            color = 0
        elif 25 <= self.hsv[0] <= 55:       # yellow 25-35
            color = 3
        elif 85 <= self.hsv[0] <= 165:     # green 125-165
            color = 1
        elif 220 <= self.hsv[0] <= 310:     # blue 270-290
            color = 2
        else:                               # line
            color = 4

        # Colour Filter (for PID Controller)
        if len(self.colour_filter) > 5:
            self.colour_filter.pop(0)
        self.colour_filter.append(color)

        # Colour Measurements Filter (for Localization)
        if len(self.colour_measurments) > 100:
            self.colour_measurments.pop(0)
        self.colour_measurments.append(color)

        # Set current detected colour based on Colour Filter
        self.cur_colour = int(statistics.median(self.colour_filter) + 0.5)

        # self.cur_colour = color
        # print(color)

    def state_model(self):
        """
        State model: p(x_{k+1} | x_k, u)
        """

        state_predict = np.roll(self.probability, 1)

        return state_predict
```

```
    def measurement_model(self):
        """
        Measurement model p(z_k | x_k = colour) - given the pixel intensity,
        what's the probability that of each possible colour z_k being observed?
        """

        measurement_model = np.zeros(len(self.colour_codes)-1)

        for i in range(len(self.colour_codes)-1):
                    measurement_model[i]  =  1/math.sqrt((self.colour_codes[i][0]  -
self.rgb[0])**2     +     (self.colour_codes[i][1]     -     self.rgb[1])**2     +
(self.colour_codes[i][2] - self.rgb[2])**2)

        return measurement_model

    def state_predict(self):
        """
        update self.state_prediction with the predicted probability of being at each
        state (office)
        """

        self.state_prediction = self.state_model()

    def state_update(self):
        """
        update self.probabilities with the probability of being at each state
        """

        measurement_model = self.measurement_model()

        for i in range(len(self.colour_map)):
                            self.probability[i]   =   self.state_prediction[i]   *
measurement_model[self.colour_map[i]]

        # Normalize
        prob_sum = sum(self.probability)
        for i in range(len(self.colour_map)):
            self.probability[i] = self.probability[i] / prob_sum


if __name__ == "__main__":

    # Initialize Localization Parameters
    colour_map = [3, 1, 2, 0, 0, 1, 2, 0, 3, 1, 2] # (0: orange, 1: green, 2: blue, 3:
yellow, 4: line)
    p0 = [1/11] * 11 # Initial probability of being at a given office is uniform
    offices = [0, 3, 6  ] # Office delivery locations


    rospy.init_node("final_project")
    localizer = BayesLoc(p0, colour_map)
    rospy.sleep(0.5)
    rate = rospy.Rate(60) # Increased from 30
```

```
    # Initialize Control Parameters
    last_state = ""
    color_timer = rospy.Time.now().to_sec() - 2
    color_registered = False

    # Initialize Plotting Parameters
    total_steps = 0
    steps_per_plot = 6 # Number of steps to visualize in each plot
    office_indices = np.arange(2, 13)
    plot_colour_map = {0: 'orange', 1: 'green', 2: 'blue', 3: 'yellow'}  # RGB values
for the colours
    plot_prob = []

    while not rospy.is_shutdown():
        if localizer.cur_colour == 4: # Line

            # PID Line Following
            localizer.controller.follow_the_line()

            # Set State to Line
            last_state = "line"

        else: # Colour

            # If we are entering a new Colour from a Line
            if last_state == "line":
                localizer.controller.correct_left() # Apply correction if necessary
                color_registered = False

            # Straight Line
            localizer.controller.straight_line()

            # State Predict and Update for new Office Location
             if rospy.Time.now().to_sec() - color_timer > 3 and not color_registered
and all(el in [0,1,2,3] for el in localizer.colour_measurments):

                # State Predict and State Update
                localizer.state_predict()
                localizer.state_update()

                rospy.loginfo(localizer.probability)
               rospy.loginfo(localizer.probability.index(max(localizer.probability)))

                # If confident of state, deliver package at desired office
                if max(localizer.probability) > 0.7:
                                                                curr_office      =
localizer.probability.index(max(localizer.probability)) # Currently perceived office
index

                    if curr_office in offices:
                        localizer.controller.deliver_package() # Deliver package
                         offices.pop(offices.index(curr_office)) # Remove office from
list to deliver after delivering

                # Store state to plot
```

```
                total_steps += 1
                plot_prob += [np.copy(localizer.probability)]

                color_registered = True
                color_timer = rospy.Time.now().to_sec() # Reset timer

            # Set State to Colour
            last_state = "colour"

        rate.sleep()

    rospy.loginfo("finished!")

# Visualization of results
for plot_index in range(0, total_steps, steps_per_plot):
    plt.figure(figsize=(10, 8))
    for i in range(steps_per_plot):
        step = plot_index + i
        if step >= total_steps:
            break
        plt.subplot(steps_per_plot, 1, i + 1)

        # Assigning colors to bars based on colour_map
        bar_colours = [plot_colour_map[colour] for colour in colour_map]

        plt.bar(office_indices, plot_prob[step], color=bar_colours)
        plt.xticks(office_indices)  # Ensure each office number is a tick mark
        plt.ylim(0, 1)  # Set y-axis limit for better comparison
        plt.axhline(y=0.7, color='r', linestyle='-') # Add Confidence Threshold Line
        plt.ylabel('Probability')
        plt.title(f'Step {step + 1} Probability Distribution')

    plt.xlabel('Office Number')  # Only set x-label for the bottom subplot
    plt.tight_layout()  # Adjust layout for neatness
    plt.show()
```