

Final Lab Report
ROB301 Introduction to Robotics

Eric Lefort – 1006896245
Gurpreet Mukker – 1005985971

5 December 2022

Introduction

The objective of the lab was to design a mail delivery robot. The robot is provided with a topological map of the circuit and a list identifying specific offices at which it needs to stop and deliver mail. The robot should autonomously navigate the circuit and stop at the specified offices. The exact path of the robot is defined by a black tape on the floor and the offices are marked by colored tiles.

Robot Platform

Turtlebot3 Waffle Pi, a four wheeled robot was used for this project. The main components of this robot are as follows

1. Raspberry Pi - This onboard computer runs ROS nodes and communicates with the PC through wifi network. The onboard ROS nodes collected data from all the sensor and sent to the host PC for processing
2. Lidar - This sensor shoots a laser that spins around to make range measurements of the environment. This sensor is especially useful for making 2D Map of the environment
3. IMU - This sensor contains an accelerometer and gyroscope that provides acceleration and angular speed in all the axis directions. This data is also used in odometry
4. DYNAMIXEL XM430 Actuators - Integrated DC motors with control, sensors, drivers etc. The sensor/encoders provide odometry data as well.
5. RPi Camera - 8MP camera to detect black tape and colored tiles
6. OpenCR - Open source motor controller that interfaces between Raspberry Pi and the motors

Strategy

The strategy was based on the objectives that can be summarized as below

1. Line detection and line following using PID Control
2. Color detection using the camera (Color Detection)
3. Estimating its position on the map (Bayesian Localization)

To make the project more manageable, it was divided into the following subtasks which were tackled individually and later integrated together.

Bayesian Localisation Module

Our bayesian localization algorithm (see Code > bayesian.py) was designed and tested separately from the rest of the project. Since the topological map, state model and measurement model were already given, we were able to design the algorithm and run a simulation of the actual course to measure its performance. This testing involved:

- Incorrect detection of delivery locations
- Incorrect color measurements. When providing erroneous color measurements, the algorithm was robust enough to maintain a good prediction with one wrong measurement but began to lose accuracy with two wrong measurements in a row.

Line Following Module

Our control loop from the previous lab was designed to make the robot follow the black line. See implementation details in Design Methodology. In addition, the following new logic was required:

- Going in a straight line ($\omega = 0$) when the robot is above a colored tile.
- When the black line and color tile are in the frame at the same time, measurements are unreliable. We applied a low-pass filter which only accepted measurements if they are repeated 8 times in a row.
- Stopping at the specified color tiles for delivery.

Design Methodology

Line Detection and Line Following

We use the onboard camera to identify the lateral position of a black tape(appendix - A). Our system measures the error term $e(t)$ as the distance of the black line from the center of the frame. We feed back this error term to a PID control loop (See appendix - B) to regulate the position of the robot such that it stays centered on this line while it moves. The control variable is angular velocity ω .

$$\omega = K_p e(t) + K_i \int_0^t e(\tau) d\tau + K_d \frac{de(t)}{dt}$$

To set the parameters, K_I , K_D and K_P we began by using the Ziegler-Nichols method, but needed to do much more tuning using trial and error methods to get the best control.

The linear velocity was controlled by a simple two-state system based on the error term $e(t)$.

- State 1 - Velocity set to V_{Max}
 - $|e(t)| < Threshold$ i.e. robot is on a relatively straight path
- State 2 - Velocity set to $0.7 \times V_{Max}$
 - $|e(t)| \geq Threshold$ i.e. robot is on curvy path

Notable observations during implementation

We noticed that K_p alone was sufficient to follow the line on the course although at a relatively low speed but the motion of the robot oscillated without settling on a straight line.

Addition of K_d removed the jerky motions and made it smooth because it resists any input that causes sudden movements in the robot. Small values of K_i parameters provided a small improvement to the control, while larger values made the robot oscillate. We kept this value very small in the final design.

Color Detection

Color similarity using euclidean distance formula

Ros node “mean_img_rgb” published the color measurement as a vector $\vec{v} = [r, g, b]$ where $r, g, b \in [0, 255.0]$.

Initially we averaged multiple measurements of the color tiles to find the corresponding fixed baseline colors. (See Table 4) To take a measurement, we found the L2 norm (distance) between each of the baseline color values and the current color reading from the camera. We classified the new measurement as the closest baseline color.

Unfortunately, this method proved to be highly sensitive to lighting conditions, making our measurements inaccurate.

Cosine Similarity

To mitigate errors due to variable lighting, we decided to find the cosine similarity between the current and average color vector according to the following formula. Using this scheme, the angle is invariant to vector magnitude changes(see Appendix - D). (full spectrum image brightness) For color i :

$$s_i = \frac{\vec{v}_m \vec{v}_i}{\|\vec{v}_m\| \|\vec{v}_i\|}$$

RGB values with tuning parameters

Given that we were working in an RGB color space, we decided to simply attempt to classify images as being red, green or blue. To do this we set up the problem as follows

$$R = k_r \cdot r \quad G = k_g \cdot g \quad B = k_b \cdot b$$

Then we would classify the image as red if $R == \max(R, G, B)$ and so on for green and blue. Here, the parameters k_r, k_g, k_b allowed us to tune the likelihood of measuring red, green and blue respectively.

While this method totally ignored the yellow readings, we assumed that the Bayesian model would be robust enough to compensate for this lack of measurement accuracy. In particular, the localizer usually classifies yellow as red. When getting a red reading, the Bayesian model considers there to be a higher chance of the true color being yellow. This is what allows our model to handle these tiles.

Localization

For the robot to stop at the correct location, we required proper localization of the robot on the provided topological map (see Appendix C) which was accomplished using Bayesian Localization techniques.

Bayesian localization is a two step process

1. State Prediction - This step predicts the probability distribution function of the robot's state based on the current state $p(x_k | z_{0:k})$ and next input u_k

$$p(x_{k+1} | z_{0:k}) = \sum_{x_k \in \Lambda} p(x_{k+1} | x_k, u_k) p(x_k | z_{0:k})$$

2. State Update - This step takes the state prediction $p(x_{k+1} | z_{0:k})$ and corrects it by incorporating the new measurements z_{k+1} .

$$p(x_{k+1} | z_{0:k+1}) = \frac{p(z_{k+1} | x_{k+1}) p(x_k | z_{0:k})}{\sum_{x_k \in \Lambda} p(z_{k+1} | x_{k+1}) p(x_k | z_{0:k})}$$

Since the system is discrete, we use discrete PDF in a matrix form. The probability $p(x_{k+1} | x_k)$ is looked up from Table 1(appendix) and the probability $p(z_{k+1} | x_{k+1})$ is looked up from Table -2 (appendix) depending on the current position(x) and the color measurement(z) at the current time step.

Mail delivery

As the robot navigates the course and the localizer receives measurements that are consistent with the map, the confidence will increase. We take the maximum likelihood estimate and check if it is one of the required stops. If it is, we check that our maximum likelihood estimate is above the confidence threshold,

in this case, 60%. We also check that the next most likely state is at least 30% lower in likelihood. If these conditions are met, then we stop at the office.

Demonstration Performance

For the project demonstration, we were provided with a list of 3 offices at which our robot was required to stop. We encoded these as a binary vector. We were also given a randomized starting point, which our robot would have to deduce after making some observations. In our case:

Offices requiring a stop: [0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1]

Initial position: between offices 4 and 5

During our first attempted demonstration, our controller was producing predictions that were unexpected. The estimates appeared uncorrelated to the map. After some troubleshooting, we identified the errors as being caused by an incorrect map. Our map consisted of an ordered list of the tile colors. (0: red, 1: green, 2: blue, 3: yellow)

Previous map: [3, 0, 1, 2, 2, 0, 1, 2, 3, 0, 1]

Fixed map: [3, 1, 2, 0, 0, 1, 2, 0, 3, 1, 2]

After inputting the correct map, the localization performed very well. The robot successfully stopped at tile 6 during its first pass around the track, but with relatively low confidence due to having only taken 2 measurements at that point. Successive measurements improved our localizer's confidence. Our robot successfully performed all required stops, though it did require a few manual corrections to its trajectory while passing over a colored tile.

Future Improvements

Despite our localizer having been able to successfully control the robot throughout the demonstration, stopping at all the correct offices, there remain many aspects of our system that can be improved to make it more accurate and robust.

Improve line detection / no longer on line

Occasionally, when passing over a colored tile, the robot rotates erratically, throwing it off course. This is caused by the fact that it struggles to track the line when it is moving over the edge of the tile. To counteract this, we could devise a system where we have a rolling average of line error measurements (over the past 10-20 measurements for example) and compare that to the new measurements we are receiving. If the error is changing too rapidly, then we could opt to avoid changing the input until we have received many such readings in a row.

Improve color detection

The measurement model given in project document (appendix F) which we used to construct Table-2(appendix) was just a general guideline on how to construct the $p(z|x)$ terms. In our design we used the same values. We greatly simplified the color classification system, thus, we should modify the Bayesian probabilistic model to reflect our system's behavior. For example, we never take yellow

measurements and yellow tiles are usually classified as red. An example of a measurement model that could work is provided in the appendix (Table 3)

Converting image to color measurement

It was found that the ROS node “mean_img_rgb”(provided to us in the lab) used the mean color of the first 200 rows of the image to create a color measurement(appendix G). This is a very unreliable method since it could include the gray floor in the window. Instead, a smaller window, centered at the top of the frame should have been used to avoid getting unwanted objects and shadows in the frame. (perhaps 100 x 100 or 200 x 200, could be determined using trial and error)

HSL

A method for color detection that we have not yet tried is modeling our colors in the HSL coordinate space and comparing only the hues. When we map the rgb color to HSL, the new vector will have a hue in the range [0,360). This system makes a lot of sense in our case, because it separates the saturation and lightness from the hue, which is the only part we are interested in measuring.

Color	red	green	blue	yellow
hue	0	120	240	60

For our implementation, we can simply classify the new measurement as the color which minimizes the error between the hue values.

Better camera

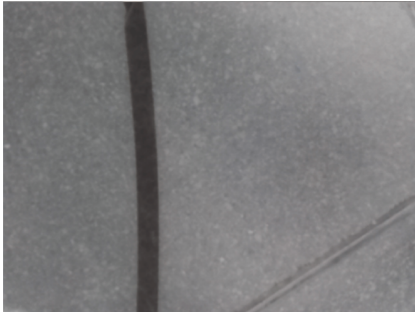
One of the major obstacles to getting highly accurate and consistent measurements was the camera. This could be improved in a few ways. First, adding a form of built-in lighting to the vision system could allow it to operate consistently in different light levels and in environments where there are moving shadows. Second, acquiring a camera with a higher quality sensor and larger color space could allow the system to better distinguish between colors. Observing Table 4, (see Appendices) we see that the robot has a very hard time telling certain colors apart, particularly with yellow.

Conclusion

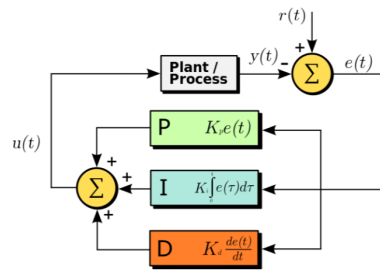
In this project the robot was able to deliver the mail on the designated spots successfully. PID control was used to keep the robot on the track using images from the camera and it worked very well. Localization was established using Bayesian methods which worked really well. Despite very poor color measurements from the camera, the robust Bayesian localization algorithm still allowed the robot to navigate accurately most of the time. To improve color measurements, we suggest filtering the error values to avoid erratic movement while moving over color tiles. To improve the color measurements, we recommend using HSL color space where we could classify the color based on the hue values while ignoring saturation and lightness which are prone to the noise of the robot. Further small improvement can be achieved through methods described above.

Appendices

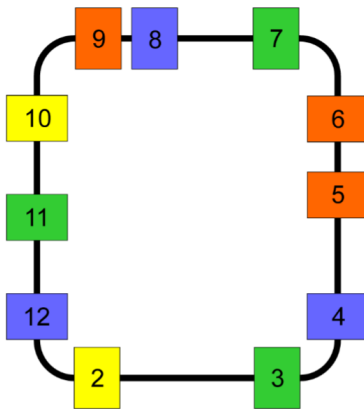
Appendix A



Appendix B

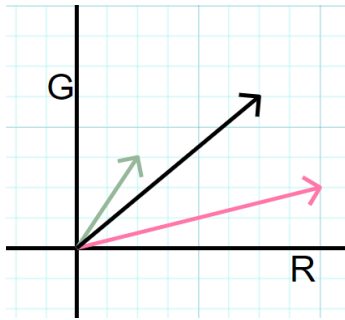


Appendix C



Appendix D

In this example, the black measurement vector is mapped to green as the angle between the tuned green vector and measurement is the smallest.



Appendix E

```
if not abs(self.error_x) > 280: # Line detected
    self.cur_colour = [False, False, False, False, True]
    self.conseq_line_detec += 1
    if self.conseq_line_detec > 8: #####
        return
else:
    self.conseq_line_detec = 0
```

Appendix F

z_k	$x_k \sim$	blue	green	yellow	orange
blue		0.60	0.20	0.05	0.05
green		0.20	0.60	0.05	0.05
yellow		0.05	0.05	0.65	0.20
orange		0.05	0.05	0.15	0.60
nothing		0.10	0.10	0.10	0.10

Appendix G

```
def camera_callback(self, msg):
    try:
        rgb_cv_img = self.bridge.imgmsg_to_cv2(msg, "rgb8")
    except CvBridgeError as e:
        print(e)

    # publish the color sensor reading
    color_array = rgb_cv_img[:200]
    intermediate = np.mean(color_array, axis=0)
    color = np.mean(intermediate, axis=0)

    color_msg = UInt32MultiArray()
    color_msg.data = color
    self.color_sensor_publisher.publish(color_msg)
```


Table 1.

		Next Position $X_{(K+1)}$										
Current Position X_K		2	3	4	5	6	7	8	9	10	11	12
	2	0.10	0.05	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.85
	3	0.85	0.10	0.05	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	4	0.00	0.85	0.10	0.05	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	5	0.00	0.00	0.85	0.10	0.05	0.00	0.00	0.00	0.00	0.00	0.00
	6	0.00	0.00	0.00	0.85	0.10	0.05	0.00	0.00	0.00	0.00	0.00
	7	0.00	0.00	0.00	0.00	0.85	0.10	0.05	0.00	0.00	0.00	0.00
	8	0.00	0.00	0.00	0.00	0.00	0.85	0.10	0.05	0.00	0.00	0.00
	9	0.00	0.00	0.00	0.00	0.00	0.00	0.85	0.10	0.05	0.00	0.00
	10	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.85	0.10	0.05	0.00
	11	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.85	0.10	0.05
	12	0.05	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.85	0.10

Table 2.





		Position of the robot										
Measurement Colour		2	3	4	5	6	7	8	9	10	11	12
	Blue	0.03	0.07	0.20	0.02	0.02	0.07	0.20	0.02	0.03	0.07	0.20
	Green	0.03	0.20	0.07	0.02	0.02	0.20	0.07	0.02	0.03	0.20	0.07
	Yellow	0.33	0.02	0.02	0.07	0.07	0.02	0.02	0.07	0.33	0.02	0.02
	Red	0.07	0.02	0.02	0.20	0.20	0.02	0.02	0.20	0.07	0.02	0.02
	Nothing	0.05	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.05	0.03	0.03

Table 3. Measurement model $p(z_k | x_k)$

$Z_k x_k$	red	green	blue	yellow
red	0.7	0.05	0.05	0.6
green	0.15	0.7	0.25	0.3

blue	0.15	0.25	0.7	0.1
yellow	0	0	0	0

Table 4. Obtained parameters after tuning

Color	Red	Green	Blue	Yellow
[R G B]	[213,73,132]	[143,169,147]	[148,139,171]	[148,149,147]
				

Code

final_project.py

```
#!/usr/bin/env python
import rospy
import math
from geometry_msgs.msg import Twist
from std_msgs.msg import String, UInt32MultiArray, UInt32,
Float64MultiArray
import numpy as np
import colorsys

from bayesian import *

class BayesLoc:
    def __init__(self, p0, colour_codes, colour_map):
        self.colour_sub = rospy.Subscriber(
            "mean_img_rgb", Float64MultiArray, self.colour_callback
        )
        self.line_sub = rospy.Subscriber("line_idx", UInt32,
            self.line_callback)
        self.cmd_pub = rospy.Publisher("cmd_vel", Twist,
            queue_size=1)

        self.num_states = len(p0)
        self.colour_codes = colour_codes
        self.colour_map = colour_map
        self.probability = p0
        self.state_prediction = np.zeros(self.num_states)

        self.clr_idx = np.array(['r', 'g', 'b', 'y', 'l'])
        self.cur_colour = None # most recent measured colour
        self.rate = rospy.Rate(10)

        self.max_v = 0.1
        self.v = 0.1 #0.26
        # max angular z 1.82
        self.omega_max = 0.5
        self.omega_f = 0
        self.dt = 1/10

        # controller parameters
        self.desired_x = 320
        self.kp = 0.004
        self.ki = 0.004
        self.kd = 0.001

        self.error_x = 0
        self.error_integral = 0 # freq = 30 hz
        self.error_derivative = 0

        self.checkpoints=[0.61,1.22,2.44,3.05]
        self.cur_checkpoint = 0
        self.stopped = False
        self.stop_time = 0

        self.conseq_line_detec = 0
        self.new_waypoint_detec = 0
        # Colour calibration
        # self.colour_avg = 0
        # self.num_reading=0
        # self.colour_tot = 0

    def colour_callback(self, msg):
        """
        callback function that receives the most recent colour
        measurement from the camera.
        """
        self.prev_colour = self.cur_colour
        rgb_read = np.array(msg.data) # [r, g, b]
        rgb_read[0] *= 0.94
        rgb_read[1] *= 1.08

        print(rgb_read)
        line = np.array([False, False, False, False, True])
        if not abs(self.error_x) > 220: # Line detected
```

```
        self.cur_colour = line
        self.conseq_line_detec += 1
        if self.conseq_line_detec > 8: #####
            return
        else:
            self.conseq_line_detec = 0

        # colour readings / calibration
        # self.num_reading+=1
        # self.colour_tot += rgb_read
        # self.colour_avg=self.colour_tot / self.num_reading

        # print(self.colour_avg)

        # errors = np.zeros((1,5))
        # for i in range(5):
        #     code = self.colour_codes[i]
        #     errors[i] = (np.dot(rgb_read,
        code)/(np.linalg.norm(rgb_read)*np.linalg.norm(code)))
        #     print(errors[i])
        # errors = [(np.dot(rgb_read,
        code)/(np.linalg.norm(rgb_read)*np.linalg.norm(code))) for code in
        self.colour_codes]
        errors = [np.linalg.norm(rgb_read - code) for code in
        self.colour_codes]

        clr = np.max(rgb_read) == rgb_read
        clr = np.append(clr, [False, False])

        # camera_reading = (errors == np.min(errors))
        # print(f'camera_reading {camera_reading}')
        # self.cur_colour = camera_reading

        self.cur_colour = clr

        # print(f'{self.prev_colour},\t {self.prev_colour[line]},
        \t{self.cur_colour[line]}')

        if not self.prev_colour is None and (self.prev_colour[line])
        and (not self.cur_colour[line]):
            self.new_waypoint_detec = True

    def line_callback(self, msg):
        """
        TODO: Complete this with your line callback function from
        lab 3.
        """

        error_prev = self.error_x
        self.error_x =self.desired_x- msg.data

        if self.stopped:
            self.v = 0
            self.omega_f = 0
        else:
            self.v=self.max_v
            self.omega_f = 1

        self.error_integral = self.error_x * self.dt
        self.error_derivative = (self.error_x - error_prev) /
        self.dt # divide by dt
        return

    def wait_for_colour(self):
        """Loop until a colour is received."""
        rate = rospy.Rate(100)
        while not rospy.is_shutdown() and self.cur_colour is None:
            rate.sleep()

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

    def measurement_model(self, x):
        """
```

```

        Measurement model  $p(z_k | x_k = \text{colour})$  - given the pixel
        intensity,
        what's the probability that of each possible colour  $z_k$ 
        being observed?
        """
        if self.cur_colour is None:
            self.wait_for_colour()

        prob = np.zeros(len(self.colourCodes))

        """
        TODO: You need to compute the probability of states. You
        should return a 1x5 np.array
        Hint: find the euclidean distance between the measured RGB
        values (self.cur_colour)
        and the reference RGB values of each colour
        (self.ColourCodes).
        """

        # print("Colour Probability (Non-Normalized):")
        # {}.format((self.cur_colour - self.colour_codes)**2))

        return prob

    def state_predict(self, u):
        rospy.loginfo("predicting state")
        """
        TODO: Complete the state prediction function: update
        self.state_prediction with the predicted probability of
        being at each
        state (office)
        """
        # X_K_1_H = self.state_prediction

        # for i in range(len(X_K_1_H)):
        #     x_n=0
        #     for j in range(len(X_K_1_H)):
        #         if(u == CLOCKWISE): # -1
        #             # state_model(u)
        #             x_n = x_n + np.transpose(XK_XK_1)[i][j]*
X_K[j]

        #         elif(u == ACLOCKWISE):
        #             x_n = x_n + XK_XK_1[i][j] * X_K[j]
        #         else:
        #             x_n = x_n + XK_XK_0[i][j] * X_K[j]

        #     X_K_1_H[i] = x_n
        # self.state_prediction = X_K_1_H

        state_prediction(u)
        self.state_prediction = np.copy(X_K_1_H)

    def state_update(self):
        rospy.loginfo("updating state")
        """
        TODO: Complete the state update function: update
        self.probabilities
        with the probability of being at each state
        """
        # clr_msmnt = self.clr_idx[self.cur_colour]

        print("self.cur_colour---"+str(self.cur_colour))

print("self.convert_colour_code(self.cur_colour)---"+str(self.conver
t_colour_code(self.cur_colour)))

        state_updatel(self.convert_colour_code(self.cur_colour))
        self.state_curr = np.copy(X_K)

        # def state_update(self):
        #     rospy.loginfo("updating state")
        #     """
        #     TODO: Complete the state update function: update
        self.probabilities
        #     with the probability of being at each state
        #     """
        #     # clr_msmnt = self.clr_idx[self.cur_colour]

        #     for i in range(len(X_K_1_UPD)):
        #         numer =
Z_XK[clr_msmnt][map_[i]]*self.state_prediction[i]
        #         denom = 0
        #         for j in range(len(X_K_1_UPD)):

#
#
denom=denom+Z_XK[clr_msmnt][map_[j]]*self.state_prediction[j]
#         X_K[i] = numer/denom

    def convert_colour_code(self, clr_array):
        blu = 0
        grn = 1
        yel = 2
        org = 3
        nothing = 4
        clr_msmnt = self.clr_idx[self.cur_colour]
        if (clr_msmnt == 'r'):
            return org

        if (clr_msmnt == 'b'):
            return blu

        if (clr_msmnt == 'g'):
            return grn

        if (clr_msmnt == 'y'):
            return yel

        if (clr_msmnt == 'l'):
            return nothing

    def follow_the_line(self):
        if self.cur_colour is None:
            self.wait_for_colour()
        tmp = np.array([0, 0, 0, 0, 1])
        if tmp[self.cur_colour]:
            omega = self.omega_f * ( \
                self.kp * self.error_x + \
                self.ki * self.error_integral + \
                self.kd * self.error_derivative)
            # print('LINE DETECTED')
        else:
            omega = 0
            # print('LINE NOT DETECTED')

        twist=Twist()
        twist.linear.x=self.v
        # twist.linear.x=0
        twist.angular.z=omega * 0.6
        self.cmd_pub.publish(twist)

    # def route_execution(self):

    def move_to_next(self):
        conseq_l = 0
        conseq_c = 10
        if self.cur_colour is None:
            self.wait_for_colour()
        print("CUR_CLR: {}, CLR_IDX: {}".format(self.cur_colour,
self.clr_idx[self.cur_colour]))
        # while self.cur_colour == None:
        #     self.rate.sleep()
        while self.clr_idx[self.cur_colour] != 'l' and not
rospy.is_shutdown(): # MOVE TO END OF CURRENT TILE
            self.follow_the_line()
            self.rate.sleep()
        while self.clr_idx[self.cur_colour] == 'l' and not
rospy.is_shutdown(): # MOVE TO NEXT TILE
            self.follow_the_line()
            self.rate.sleep()

    # def deliver(self):

    def convert_colour_map(colour_map):
        blu = 0
        grn = 1
        yel = 2
        org = 3
        nothing = 4
        for i in range(len(colour_map)):
            if colour_map[i] == 0:
                map_[i] = org
            if colour_map[i] == 1:
                map_[i] = grn
            if colour_map[i] == 2:

```

```

        map_[i] = blu
        if colour_map[i] == 3:
            map_[i] = yel

if __name__ == "__main__":

    # This is the known map of offices by colour
    # 0: red, 1: green, 2: blue, 3: yellow, 4: line
    # current map starting at cell #2 and ending at cell #12
    # colour_map = [3, 0, 1, 2, 2, 0, 1, 2, 3, 0, 1]
    colour_map = [3, 1, 2, 0, 0, 1, 2, 0, 3, 1, 2]

    convert_colour_map(colour_map)
    init_bayesian()

    # TODO calibrate these RGB values to recognize when you see a
    colour
    # NOTE: you may find it easier to compare colour readings using
    a different
    # colour system, such as HSV (hue, saturation, value). To
    convert RGB to
    # HSV, use:
    # h, s, v = colorsys.rgb_to_hsv(r / 255.0, g / 255.0, b / 255.0)
    # colour_codes = [
    #     [167, 146, 158], # red
    #     [163, 184, 100], # green
    #     [173, 166, 171], # blue
    #     [167, 170, 117], # yellow
    #     [150, 150, 150], # line
    # ]

    # colour_codes = [
    #     [205.00777344, 71.82074219, 118.78730469], # red
    #     [141.92238281, 175.46325781, 113.37578906], # green
    #     [163.42664063, 154.76842188, 172.66335937], # blue
    #     [163.48789844, 154.51808594, 146.96142969], # yellow
    #     [1,1,1], # line
    # ]

    # colour_codes = [
    #     [222.60746528, 80.23368774, 149.94774455],
    #     [153.343846, 181.10858058, 153.98644996],
    #     [177.98156326, 162.78980768, 185.73809754],
    #     [163.54012709, 164.45835603, 158.94309152],
    #     [1,1,1],
    # ]

    colour_codes = [ # 'r', 'g', 'b', 'y', 'l'
        [212.7521484375, 72.7033359375, 132.1031796875], # r
        [143.340703125, 168.721421875, 147.481078125], # g
        [148.301296875, 139.303796875, 171.307140625], # b
        [147.729875, 149.130125, 146.59459375], # y
        [1,1,1],
    ]

    # colour_codes = [
    #     [222.60746528, 80.23368774, 149.94774455],
    #     [153.343846, 181.10858058, 153.98644996],
    #     [177.98156326, 162.78980768, 185.73809754],
    #     [163.54012709, 164.45835603, 158.94309152],
    #     [168.17824336, 157.40005376, 170.44722016],
    # ]

#FlashLight
    # colour_codes = [
    #     [205.21265486, 78.93533812, 125.58477038],
    #     [130.42153719, 162.00865726, 133.19120532],
    #     [159.61547117, 147.40164879, 161.31224496],
    #     [144.7171216, 147.09664504, 142.75635632],
    #     [43.30992359, 139.01833841, 139.87589367],
    # ]
    # initial probability of being at a given office is uniform
    p0 = np.ones_like(colour_map) / len(colour_map)

    rospy.init_node("final_project")
    rospy.sleep(0.5)
    rate = rospy.Rate(10)
    localizer = BayesLoc(p0, colour_codes, colour_map)

    direction = 1
    # localizer.move_to_next()

    # mail_deliver = True
    stop_points = np.array([0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1])

    while not rospy.is_shutdown():
        """
        TODO: complete this main loop by calling functions from
        BayesLoc, and
        adding your own high level and low level planning + control
        logic
        """
        localizer.follow_the_line()

        if(localizer.new_waypoint_detec):
            localizer.new_waypoint_detec = 0
            print('WAYPOINT DETECTED')
            localizer.state_update()
            print(X_K)

        loc = X_K == np.max(X_K)
        print(loc)

        if np.count_nonzero(loc) == 1 and stop_points[loc]:
            """ STOP
            plt.plot(np.arange(2,13),X_K)
            plt.title("Update Color:
            {}".format(localizer.clr_idx[localizer.cur_colour]))
            plt.show()

            # plt.plot(np.arange(2,13),X_K)
            # plt.title("Update Color:
            {}".format(localizer.clr_idx[localizer.cur_colour]))
            # plt.show()
            print(X_K_1_H)
            localizer.state_predict(direction)
            # plt.plot(np.arange(2,13),X_K_1_H)
            # plt.title("Prediction")
            # plt.show()

            # if mail_deliver:
            #     localizer.deliver()

        rate.sleep()

    rospy.loginfo("finished!")
    rospy.loginfo(localizer.probability)

```

Bayesian.py

```

import numpy as np
import matplotlib.pyplot as plt

```

```

blu = 0
grn = 1
yel = 2
org = 3
nothing = 4

```

```

CLOCKWISE = -1
ACLOCKWISE = 1
NO_MOTION = 0

```

```

col=np.array([blu, grn, yel, org])

```

```

map_ = [2, 1, 0, 3, 3, 1, 0, 3, 2, 1, 0]

```

```

num_colors =[0,0,0,0]

```

```

def init_num_colors():
    num_colors[blu] = map_.count(blu)
    num_colors[grn] = map_.count(grn)
    num_colors[yel] = map_.count(yel)
    num_colors[org] = map_.count(org)
    # num_colors = [3, 3, 2, 3] # blu, grn, ylw, org

```

```

X_K = np.full((len(map_)),1/len(map_))
X_K_1_H = np.full((len(map_)),1/len(map_))
X_K_1_UPD = np.zeros((len(map_)))

```

```

'''Orange, Blue, Green, Yellow'''

```

```

Z_XK = [
    [0.60, 0.20, 0.05, 0.05],
    [0.20, 0.60, 0.05, 0.05],
    [0.05, 0.05, 0.65, 0.20],
    [0.05, 0.05, 0.15, 0.60],
    [0.10, 0.10, 0.10, 0.10]]

clr_codes = {
    'blue':0,
    'green':1,
    'yellow':2,
    'orange':3,
    'nothing':4}

def fun_X_Z(clr_measmnt):
    return np.array([Z_XK[clr_measmnt][i]/num_colors[i] for i in
map_] )

X_Z = np.zeros((5, 11))

def init_X_Z():

    for msmnt in range(5):
        X_Z[msmnt] = fun_X_Z(msmnt)

# np.array(fun_X_Z(i) for i in range(5))

# ([
# [0, 0, 0, 1/3,1/3,0, 0, 1/3,0, 0, 0],
# [0, 0, 1/3,0, 0, 0, 1/3,0, 0, 0,1/3],
# [0, 1/3,0, 0, 0, 1/3,0, 0, 0, 1/3,0],
# [1/2,0, 0, 0, 0, 0, 0, 0, 0, 1/2,0, 0]])

XK_XK_1 = np.zeros((len(map_),len(map_)))
XK_XK_0 = np.zeros((len(map_),len(map_)))

def setup_XK_XK_1():
    for i in range(len(map_)):
        XK_XK_1[(i-1)%11][i] = 0.05
        XK_XK_1[i][i] = 0.1
        XK_XK_1[(i+1)%11][i] = 0.85
    # print(XK_XK_1)

for i in range(len(map_)):
    XK_XK_0[(i-1)%11][i] = 0.05
    XK_XK_0[i][i] = 0.9
    XK_XK_0[(i+1)%11][i] = 0.05

def state_prediction(input):
    for i in range(len(X_K_1_H)):
        x_n=0
        for j in range(len(X_K_1_H)):
            if(input == CLOCKWISE): # -1
                x_n = x_n + np.transpose(XK_XK_1)[i][j]* X_K[j]
            elif(input == ACLOCKWISE):
                x_n = x_n + XK_XK_1[i][j] * X_K[j]
            else:
                x_n = x_n + XK_XK_0[i][j] * X_K[j]

        X_K_1_H[i] = x_n
        # print("X_K_1_H " +str(X_K_1_H))

def state_update1(measurement_col):

    for i in range(len(X_K)):
        numer = Z_XK[measurement_col][map_[i]]*X_K_1_H[i]
        denom = 0
        for j in range(len(X_K)):
            denom = denom +
Z_XK[measurement_col][map_[j]]*X_K_1_H[j]
        # print("denom--"+str(denom))
        X_K[i] = numer/denom
        # print('A POSTERIORI ESTIMATE: {}'.format(X_K))

def test1():
    state_prediction(ACLOCKWISE)

plt.plot(np.arange(2,13),X_K_1_H)
plt.title("Prediction")
plt.show()

state_update(col[yel])
plt.plot(np.arange(2,13),X_K)
plt.title("Update yellow")
plt.show()

state_prediction(ACLOCKWISE)
plt.plot(np.arange(2,13),X_K_1_H)
plt.title("Prediction")
plt.show()

state_update(col[grn])
plt.plot(np.arange(2,13),X_K)
plt.title("Update green")
plt.show()

state_prediction(ACLOCKWISE)
plt.plot(np.arange(2,13),X_K_1_H)
plt.title("Prediction")
plt.show()

state_update(col[blu])
plt.plot(np.arange(2,13),X_K)
plt.title("Update blue")
plt.show()

state_prediction(ACLOCKWISE)
plt.plot(np.arange(2,13),X_K_1_H)
plt.title("Prediction")
plt.show()

state_update(col[yel])
plt.plot(np.arange(2,13),X_K)
plt.title("Update yel")
plt.show()

def test2():
    u = [1,1,1,1,1,1,1,1,0,1,1,1]
    clr = [org, yel, grn, blu, nothing, grn, blu, grn, org, yel,
grn, blu]

    for i in range(len(clr)):
        state_prediction(u[i])
        plt.plot(np.arange(2,13),X_K_1_H)
        plt.title("Prediction")
        plt.show()

        state_update(clr[i])
        plt.plot(np.arange(2,13),X_K)
        plt.title("Update color code: {}".format(clr[i]))
        plt.show()

def init_bayesian():
    setup_XK_XK_1()
    init_num_colors()
    init_X_Z()

if __name__ == "__main__":
    init_bayesian()
    print(X_Z)
    # print()

    # setup_XK_XK_1()
    print(XK_XK_1)

    test2()

```