ROB301 Final Design Project - The Galbraith Memorial Mail Robot

Yifei Zhou (1006878979) & Alyssa Wing (1006910755)

1. Introduction

Our Final Project consisted of programming the TurtleBot 3 Waffle Pi using Bayesian probability to represent an autonomous mail delivery robot. The Robot Operating System, or ROS, was programmed to localise based on detected colours and lines (see figure in appendix) [1]. This was built upon our previous lab in line-following using PID control. In this report, we will examine the deliverables related to Bayesian localization. The first of the four was to complete a simulation of Bayesian localization in an environment where our TurtleBot is travelling in a loop marked by black tape. Different "offices" were represented by yellow, green, blue, and red coloured paper on top of the tape. Second, we had to demonstrate that our robot could execute the full route completely without stopping and without localization. The third step was to integrate our Bayesian localization model with our TurtleBot and demonstrate that, after passing some offices, the robot could correctly converge to its actual position, i.e., which office it was at.

2. Robot Platform

The robotic platform used was the Robot Operating System. The TurtleBox is capable of differential steering due to its two separately-controlled wheels, allowing it to translate and rotate easily. These wheels are controlled by two Dynamixel actuators from an OpenCR board. The TurtleBot itself comes with an array of available sensors and actuators. The applicable features to our project are listed below [2]:

- Raspberry Pi 3B - it collects data from sensors, allowing it to be published. It is also used as a subcontroller from the PC to the relevant actuators.
- Raspberry Pi camera - permits RGB (red, green, blue) colour perception

3. Solution Strategy Overview

The first function was our path-following via PID control. Our first strategy was to decrease the linear speed of our robot and its angular velocity so that the colour detections could be more accurate. Our colour detection was reliant on our Raspberry Pi camera, which returned RGB values of the colour it detected. We converted these RGB values to HSV (hue, saturation, value) because it allowed the robot to differentiate the colours more easily; more specifically, there were greater differences in HSV values than RGB values.

Our first motor-control node, lab3.py, uses the camera to detect when the robot is on a line. The localization node, final_project.py, detects when the robot reaches the next office on the track and estimates the robot's position on the topological map of the route. Our Lab 3 code was modified to subscribe to the colour detected by the sensor, to convert this colour to HSV, and to use the result to differentiate between a line and not a line. If a line was detected, then our TurtleBot would perform line-following via our PID control. Conversely, if there was not a line detected, then the angular velocity was set to zero so that the robot would just travel straight.

The measurement model information detects whether or not the robot reaches the next office. If so, then it would estimate its current location using Bayesian localization based on its previous state estimation, control input, and latest measurement.

4. Design Methodology - Technical Details

First Deliverable: Simulation

For the simulation, there were three main variables to keep track of: the state, control, and measurement. The state at each time step *k* was represented as a numpy array of 11 elements with each element representing the probability of the robot being at that office (i.e. if the element at index 0 was 0.2 then there was a 20% chance of the robot being at the first office on the route, office 2). The state at each step was stored in the 2D array *x*. The control *u* and measurements *z* from the handout were stored in arrays.

To implement Bayesian localization in the preliminary simulation, the state and measurement models were represented as 2D arrays – see lines 16-20 and 22-27 in Bayesian.py of the appendix, respectively. These were taken directly from the project handout [1].

The state prediction step (see lines 42-53 of Bayesian.py) implemented equation (1), from the Bayesian Localization primer [3]: $p(x_{k+1}|z_{0:k}) = \sum_{x_k \in \Lambda} p(x_{k+1}|x_k, u_k)p(x_k|z_{0:k})$ (1). Line 47, for example, implements $p(X-1|X,-1)p(X|z_{0:k})$, i.e. the probability the robot has moved backwards from office *X* to office *X-1*, given a control input of -1 and the probability of having been at office *X*. Lines 44-53 perform the summation over the three possible states the robot could have moved to $x_k \in \Lambda$ and stores the state prediction array in *state_prediction*.

Lines 55-61 perform the state update step, implementing equation (2), also from the primer: $p(x_{k+1}|z_{0:k+1}) = \frac{p(z_{k+1}|x_{k+1})p(x_{k+1}|z_{0:k})}{\sum_{\xi_{k+1}\in\Lambda} p(z_{k+1}|\xi_{k+1})p(\xi_{k+1}|z_{0:k})}$ (2). Line 60 calculates the denominator of (2), where `p_measure[cmap[j]][z[k+1]]` retrieves $p(z_{k+1}|\xi_{k+1})$ from the measurement model and `state_prediction[j]` retrieves $p(\xi_{k+1}|z_{0:k})$ from the state prediction step. Similarly, line 61 calculates the entire expression for the probability of being at a particular office and stores it in the *state_update* array, which is appended in line 63 as the current state estimation for step *k* in array *x*.

To simulate, the first entry in array *x* is a uniform distribution of 1/(the number of offices). Then the rest of the array is filled by running the state prediction and state update steps *k* number of times, using the given *u* and *z* values for each step. At the end, bar graphs for the state estimation at each step are generated and displayed (see appendix).

Second Deliverable: Full-Route Execution & Colour Detections

Our line-following code from lab 3 had to be modified to allow our robot to quickly stabilise upon correcting itself on the black tape. We decreased the linear speed of our robot to 0.05 m/s to allow more accurate colour detections. As a result, we also decreased its angular velocity to prevent over-correcting, which could throw the robot off after crossing an office. To calibrate, we placed it on top of each colour and noted its average RGB values. Then, we converted them to an HSV scale as our new reference colour codes. The values were converted using a predefined function after importing “colorsys”.

Since we wanted to traverse the whole route without stopping, we created a function called “is_office” (line 080 in Lab3.py) to return “True” if the robot was at an office, and “False” otherwise, implemented using if statements - if the detected HSV was within a tolerance (each HSV had its own tolerance which we adjusted as we tested; see lines 83-85 of lab3.py) with the HSV colour codes from calibration, then the function would return the the corresponding colour as a string (equivalent to boolean True). We only compared the HSV values to the colour calibrated HSV values because if we are not at an office, then we must be on the line. If “is_office” returned “True”, then we set the angular velocity to 0 because we noticed that offices were only located on straight blocks; thus, if a colour was detected, if it just travelled straight over then it would remain on the path. If a colour was not detected, then the angular velocity would be nonzero if necessary.

During testing, we realised the “is_office” function would sometimes detect the office too late, or not at all, resulting in the robot trying to perform line-following even when it was no longer on the line. Since the robot was going slower, we inverted the logic of the is_office function. We created a similar is_line function (see lines 103-115) that would return ‘n’ for ‘no colour’ (logically “True”) if the robot was very likely on a line. If is_line returned True, then the robot would perform line-following using PID control as usual, otherwise, it would go straight.

Third Deliverable: Bayesian-Localization

Integrating the Bayesian Localization from the first deliverable with the travelling robot took multiple iterations adjusting gains and tolerances. The *state_model* function returned a 1x3 array of the appropriate probability distribution for the three possible locations the robot could be given its previous state and control input (nearly identical to Bayesian.py). The *measurement_model* function returned a similar 1x5 array of the probability distribution of the current colour measurement. These were calculated using the 2-norm of the difference between the current measurement’s RGB vector and the reference RGB vectors for each possible colour reading (initialised using `colour_codes` from line 169 in final_project.py).

The *state_predict* and *state_update* functions were modified to integrate it with the localizer. In *state_predict*, the input *u* was always set to +1 because we expected the robot to only move forwards. The *state_update* function directly updated the localizer’s current state, *self.probability*. The index of the value with the maximum probability, representing the robot’s estimated location, was published to a topic “/state”.

The colour_callback and line_callback functions simply retrieved the relevant message information from the subscribed topics for the current colour measurement and line index, respectively. The main iterations we went through are as follows:

1. Ran state predict and state update at the node rate: The model updated constantly thinking it had moved forwards. This worked terribly: the sensor was inaccurate, sometimes reading the wrong colour or reading the line as the colour, and so thought it had moved to the next office when really it had only inched forwards.
2. Added the line segments between offices as states: This was to allow the localizer to consider itself on a line so it would be less likely to think it had moved forward to the next office (as offices were the only possible states) when it hadn’t. We modified the state model so there was a higher probability of the robot staying in the same state even if *u*=1.

3. Kept track of the last *n* states so a "state change" only registered when all of them were the same. This was so any fluctuations/uncertainties due to rapidly-adjusting sensor readings would be ignored. When the state was "stable", we ran *state_predict* and *state_update*. However, this method did not work because the array of the last *n* states was updated with the current state estimation at the loop rate while the state estimation itself was only updated each time the recorded states were all the same. This was equivalent to the previous solution running at a slower rate.
4. We needed to use the current readings without updating the localizer's state. We used the measurement model function to estimate the robot's current position based on the latest reading and the last updated state probability (*self.probability*). We reset the state model back to the given matrix in the handout. If the measurement model predicted the robot to be at an office when in the previous loop iteration it predicted the robot to be between offices, then the robot had likely just arrived at an office so we ran *state_predict* and *state_update*. This gave us the best performance so far, but it was still suboptimal.

Fourth Deliverable Attempt: Mail Delivery

We were unfortunately unable to complete the fourth deliverable because our localizer was not accurate enough to reliably predict the robot's current state. Thus, it could be improved by making our localizer more accurate, which is also described in Section 6. The list of offices for the robot to visit would be added, and if the localizer determined that it is at an office it must deliver mail to, several steps must happen:

- Publish a "True" message to a new rostopic "/deliver_mail"
- Modify the motor-control node so it subscribes to "/deliver_mail" and stops the robot, rotates 90˚, waits for two seconds, and rotates back if it receives a "True" message.
- To prevent the robot from perpetually reading "True" and delivering mail to one forever, we would only have it stop if the previous message was "False" and the new one "True".

5. Demonstration Performance

After some iterations and debugging, our first simulation of Bayesian localization worked well, and the state graphs clearly showed that our model was able to localise. After some movements between states, the model converged to the correct office given a map of colours and control inputs. This is shown in Figure 1 to the right.

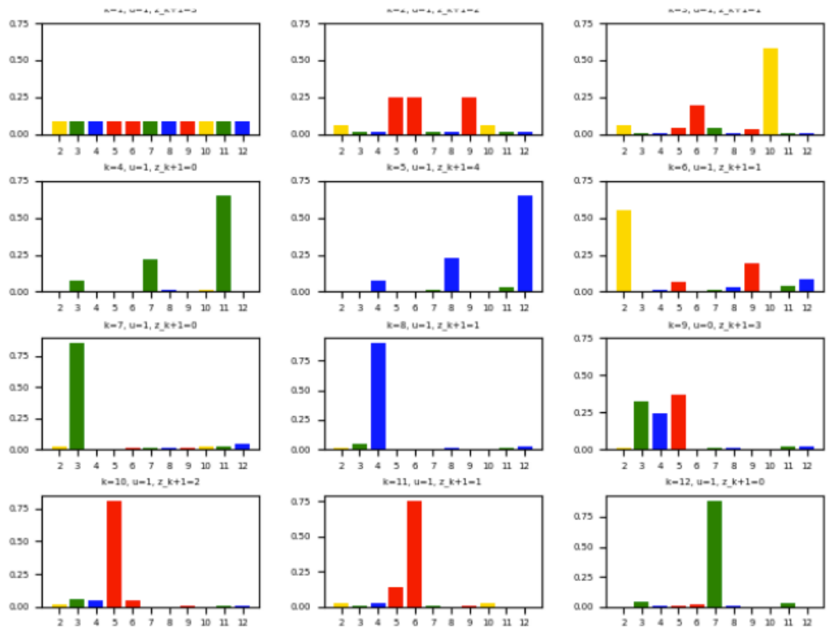


Figure 1: Probability distribution of robot's location at each step *k*

Next, we were correctly able to show that our TurtleBot could traverse the full route without stopping for the second deliverable. After some modifications to our PID line-following code, we were able to have the robot move at a slower speed to enable more accurate colour detections. By the end of our lab sessions, our TurtleBot was able to traverse multiple laps through the route without going off-track.

As mentioned above, the localizer was able to converge to the correct location, but only around 50% of the time. It was able to track correctly for about 6 offices in a row, but then it would jump to an office on the other side of the track with the same colour as the current office, or start predicting the robot was moving faster past the offices than it actually was. Since the localizer was so unreliable, we were unable to implement the mail-delivery mechanic in anything but theory.

6. Potential Improvements

There are many areas for improvements that could improve the accuracy of our robot as well as its localization and mapping abilities. First, our PID controller could be improved and more iteratively tuned without the presence of offices because there were still some slight oscillations when the robot overcorrected itself. As well, the PID control could be modified to increase the overall speed of our robot for a more efficient mail delivery service. This would go hand-in-hand with more accurate colour detection. The colour detection could be improved by more rigorously calibrating the colour codes. The measured colours could be more consistent if a flashlight were fixed to point where the camera would look in order to have our camera virtually indifferent from different room or window lighting depending on the time of day.

Furthermore, the line-following algorithm could be improved by modifying our code to only confirm that it is on a line and thus do line-following PID control after it has received multiple readings from the camera sensor proving that it is actually on a line. Previously, there were several instances where the sensor confused an office, usually blue, with the line. Consequently, the PID control was activated and caused overcorrection. Once past the office, the robot could no longer find where the line was. Similarly, we could keep track of the last 15-30 sensor measurements and compare them to the colours as well to be more certain if the TurtleBot was passing over an office. We could also require a fraction, rather than all, of the recorded preceding measurements be the same to confirm the robot's position to improve robustness.

7. Conclusion

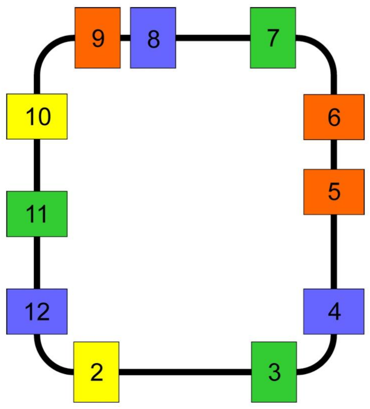
While our Galbraith Memorial Mail Robot was only partially successful, it left us with many lessons to be learned and many potential future improvements. This project emphasised the importance of not only theoretical but also practical implementation. While a simulation may work perfectly, there are a multitude of errors to be accounted for and minimised in practice. In our implementation of a PID controller, we learned the importance of patience and attention to detail over countless iterations to properly tune the different gain values according to the robot's performance in real life. We also learned the significance and effectiveness of Bayesian localization. While it worked in simulation, the colour calibration readings were too unreliable to obtain a proper localization convergence each time.

8. References

1. “Final Design Project: The Galbraith Memorial Mail Robot”, ROB301 Handout, 2022.
2. “Introduction to ROS”, ROB301 Handout, 2022.
3. “The Concept of Bayesian Localization”, ROB301 Handout, 2022.

9. Appendix

Topological map of the route



Bayesian.py

```
from random import random
import matplotlib.pyplot as plt
import numpy as np

# colour representations
b = 0
g = 1
y = 2
o = 3
n = 4

cmap = [y, g, b, o, o, g, b, o, y, g, b]

states = list(range(2, 12))

# state model p(x_k+1 | x_k = X, u_k)
p_state = [[0.85, 0.10, 0.05],  # u = -1
        [0.05, 0.90, 0.05],  # u =  0
        [0.05, 0.10, 0.85]]  # u = +1
        # X-1   X     X+1

# measurement model p(z_k | x_k) - certainty matrix
p_measure = [[0.60, 0.20, 0.05, 0.05, 0.10],     # blue state
              [0.20, 0.60, 0.05, 0.05, 0.10],     # green state
              [0.05, 0.05, 0.65, 0.15, 0.10],     # yellow state
              [0.05, 0.05, 0.20, 0.60, 0.10]]     # orange state
              # b   g     y     o     n


# actions
x = []
u = [1,1,1,1,1,1,1,1,0,1,1,1,None]
z = [n, o, y, g, b, n, g, b, g, o, y, g, b]

# initial state is random (uniform)
# x.append(states[int(random()*4)])
x.append(np.ones(11) / 11)

# Loop
for k in range(11):
  print(z[k+1])
  # state prediction
  state_prediction = np.zeros(11)
  for i in range(11):     # each state_prediction entry, x_k+1; looping for each state in one step
        s = 0
        # x_k+1 is X-1
        s += p_state[u[k]+1][0] * x[k][(i+1)%11]
        # x_k+1 is X (stays in the same state)
        s += p_state[u[k]+1][1] * x[k][i%11]
        # x_k+1 is X+1 (state is the next one)
        s += p_state[u[k]+1][2] * x[k][(i-1)%11]
        # put into state prediction
        state_prediction[i] = s

  # state update
  state_update = np.zeros(11)
  for i in range(11): #for each x_k+1 (11 offices)
        s = 0   # initialize sum for the denominator
        for j in range(11): # sum for denominator for each state
              s += p_measure[cmap[j]][z[k+1]] * state_prediction[j]
        state_update[i] = p_measure[cmap[i]][z[k+1]] * state_prediction[i] / s

  x.append(state_update)   # save into the state all 11 office measurement probabilities


fig, axes = plt.subplots(4, 3)
fig.tight_layout(pad=0.7)

colours = ['blue', 'green', 'gold', 'red']
bar_colours = [colours[i] for i in cmap]

print(bar_colours)

i = 0
for ax in axes.flat:
  ax.bar(range(2, 13), x[i], color=bar_colours)
  ax.set_xticks(np.arange(2, 13, step=1))
  ax.set_yticks(np.arange(0, 1, 0.25))
  ax.tick_params(axis='both', labelsize=5)
  ax.set_title(f"k={i+1}, u={u[i]}, z_k+1={z[i+1]}", size=5)
  i += 1

plt.show()

# fig, ax = plt.subplots(1, 1)
# ax.bar(range(2, 13), x[0], color=bar_colours)
# plt.show()
```

Lab3.py

```
#!/usr/bin/env python
import rospy
from geometry_msgs.msg import Twist
from std_msgs.msg import UInt32
from std_msgs.msg import Float64
import colorsys
from std_msgs.msg import String, Float64MultiArray

class Controller(object):
 line_index = 0
 x_estimate = 0
 rgb = [0, 0, 0]

 colour_codes = [
       [203.3, 109.76, 147.2],  # red
       [120, 139, 130],  # green
       [161.5, 152, 172],  # blue
       [186.7, 170.6, 150.5],  # yellow
       [175, 158, 162],  # line
 ]

 colour_codes_hsv = [
       [0.9332905708787685, 0.4601082144613871, 0.7972549019607844],
       [0.42105263157894735, 0.13669064748201432, 0.5450980392156862],
       [0.76, 0.149, 0.562],
       [0.09254143646408837, 0.19389394750937322, 0.732156862745098],
       [0.993, 0.0926, 0.684]
 ]

 # yellow read: 0.077, 0.183, 0.709
 # red read: 0.953, 0.705, 0.902

 colours = ['r', 'g', 'b', 'y', 'n']

 # convert rgb colour codes to hsv:
 # for colour in colour_codes:
 #     colour_codes_hsv.append(colorsys.rgb_to_hsv(colour[0]/255.0, colour[1]/255.0, colour[2]/255.0))

 # print(colour_codes_hsv)

 def __init__(self):
       # publish motor commands
       self.cmd_pub = rospy.Publisher("cmd_vel", Twist, queue_size=1)

       # subscribe to detected line index
       self.color_sub = rospy.Subscriber(
             "line_idx", UInt32, self.camera_callback, queue_size=1
       )

       # subscribe to estimated state
       self.state_sub = rospy.Subscriber(
             "state", Float64, self.state_callback, queue_size=1
       )

       # subscribe to rgb camera

```

```
      self.rgb_sub = rospy.Subscriber(
            "mean_img_rgb", Float64MultiArray, self.rgb_callback, queue_size=1
      )

      self.office_pub = rospy.Publisher("office", Float64, queue_size=1)

def rgb_callback(self, msg):
      """Callback for line index"""
      # access the value using msg.data
      self.rgb = msg.data

def camera_callback(self, msg):
      """Callback for line index."""
      # access the value using msg.data
      self.line_index = msg.data

def state_callback(self, msg):
      """Callback for state"""
      self.x_estimate = msg.data



# define true/false function if at an office or not
def is_office(self):
      r, g, b = self.rgb # detected rgb values
      h, s, v = colorsys.rgb_to_hsv(r/255.0, g/255.0, b/255.0)   # detected converting rgb values to hsv values
      h_tolerance = 0.2
      s_tolerance = 0.3
      v_tolerance = 0.14

      # for c in self.colour_codes_hsv[:4]:
      #     if abs(h - c[0]) < h_tolerance and abs(s - c[1]) < s_tolerance and \
      #           abs(v - c[2]) < v_tolerance:
      #           return True

      # return False    # no office (colour block) detected

      hsv = self.colour_codes_hsv
      for i in range(len(hsv[:-2])):
            if abs(h - hsv[i][0]) < h_tolerance and abs(s - hsv[i][1]) < s_tolerance and \
            abs(v - hsv[i][2]) < v_tolerance:
            return self.colours[i]

        return None


    def is_line(self):
        r, g, b = self.rgb # detected rgb values
        h, s, v = colorsys.rgb_to_hsv(r/255.0, g/255.0, b/255.0)   # detected converting rgb values to hsv values
        h_tolerance = 0.15
        s_tolerance = 0.05
        v_tolerance = 0.14

        hsv = self.colour_codes_hsv
        if abs(h - hsv[-1][0]) < h_tolerance and abs(s - hsv[-1][1]) < s_tolerance and \
            abs(v - hsv[-1][2]) < v_tolerance:
            return self.colours[-1]

        return None


    def bang_bang(self):
        """
        TODO: complete the function to follow the line
        """
        desired = 320
        twist = Twist()
        twist.linear.x=0.1 #moves at 0.2m/s
        rate = rospy.Rate(30)   # 10 Hz

        while not rospy.is_shutdown():
            # go until we shut down ros
            actual = self.line_index #actual position
            error = desired - actual
            if error < 0:
                   twist.angular.z = -0.8 #correction: steer towards the right (clockwise)
            elif error > 0:
                twist.angular.z = 0.8 #correction: steer towards the left (counterclockwise)
            else:
                twist.angular.z = 0 #correction: nothing
            self.cmd_pub.publish(twist) #go forwards + do angular twist correction
            rate.sleep()


    def p(self): #proportional (part 4.2)
        """
        TODO: complete the function to follow the line
        """
        desired = 320
        kp = 0.004 #proportional gain
        twist = Twist()
        twist.linear.x = 0.15 #moves at 0.1m/s
        rate = rospy.Rate(30)   # 30Hz

        while not rospy.is_shutdown():
            # go until we shut down ros
            actual = self.line_index #actual position
            error = desired - actual
            correction = kp * error
            max_correction = 1.4 #max angular velocity

            if correction > max_correction:
                twist.angular.z = max_correction
            elif correction < -1*max_correction:
                twist.angular.z = -1*max_correction
            else:
                twist.angular.z = correction

            self.cmd_pub.publish(twist) #go forwards + do angular twist correction
            rate.sleep()

    def pi_control(self): #integral (part 4.3)
        """
        TODO: complete the function to follow the line
        """
        desired = 320
        integral = 0
        kp = 0.004 #proportional gain
        ki = 0.00001 #integral gain

        twist = Twist()
        twist.linear.x = 0.15 #moves at 0.1m/s
        rate = rospy.Rate(30)   # 30Hz

        while not rospy.is_shutdown():
            # go until we shut down ros
            actual = self.line_index #actual position
            error = desired - actual
            integral = integral + error
            correction = (kp * error) + (ki * integral)
            max_correction = 1.5 #max angular velocity

            #rospy.loginfo(error)
            #rospy.loginfo(integral)
            print("error: ", error)
            print("integral: ", integral)

            if correction > max_correction:
                twist.angular.z = max_correction
            elif correction < -1*max_correction:
                twist.angular.z = -1*max_correction
            else:
                twist.angular.z = correction

                if abs(error) <  50: #very good position
                    integral = 0.7 * integral #reset integral

            self.cmd_pub.publish(twist) #go forwards + do angular twist correction
            rate.sleep()

    def pid_control(self): #derivative (part 4.4)
        """
        TODO: complete the function to follow the line
        """
        desired = 320
        integral = 0
        derivative = 0
        lasterror = 0

        # test:
        # kp = 0.0047
        # ki = 00.0000001
        # kd = 0.05

        kp = 0.003#47
        ki = 00.0000001
        kd = 0.003#5

        speed = 0.07
        midspeed = speed
        espeed = speed
        minspeed = speed

        # tolerance = 0.0005
        tolerance = 0.002


        twist = Twist()
        twist.linear.x = speed #moves at 0.1m/s
        r = 120
        rate = rospy.Rate(r)    # 30Hz

        while not rospy.is_shutdown():
              # go until we shut down ros
              actual = self.line_index #actual position
              r, g, b = self.rgb #rgb values
              h, s, v = colorsys.rgb_to_hsv(r/255.0, g/255.0, b/255.0) # converting rgb values to hsv values
              error = desired - actual

              if abs(error) > 100: #if error is greater than 150 pixels (?)
              twist.linear.x = minspeed #slows down when encounters sharper turns
              if abs(error) > 75:
              twist.linear.x = espeed
              elif abs(error) > 50:
              twist.linear.x = midspeed
              else:
              twist.linear.x = speed #go faster when more accurate


              integral = integral + error
              derivative = error - lasterror

              correction = (kp * error) + (ki * integral) + (kd * derivative)
              # max_correction = 1.82 #max angular velocity
              max_correction = 1.3

              if correction > max_correction:
                     twist.angular.z = max_correction
              elif correction < -1*max_correction:
                     twist.angular.z = -1*max_correction
              else:
                     twist.angular.z = correction

                     if abs(error) <  50: #very good position
                          integral = 0.7 * integral #reset integral

              # if at an office, go straight
              # office = self.is_office()
              # if office or not self.is_line():
              #     twist.angular.z = 0
              #     print("office:\t", office, "\thsv:\t", f"({h}, {s}, {v}")

              # else:
              #     print("at line,\t\thsv:\t", f"({h}, {s}, {v}")
              #     if abs(error) > 300:
              #           twist.angular.z = 0

              # if not at a line, go straight
              if not self.is_line():
              twist.angular.z = 0
              print("office:\t", self.is_office(), "\thsv:\t", f"({h}, {s}, {v}")
              else:
              print("at line,\t\thsv:\t", f"({h}, {s}, {v}")
              # if abs(error) > 300:
              #     twist.angular.z = 0

              # print("is_office:", office)
              # print("hsv: ", h, s, v)

```

```
             # print("self.x_estimate:", self.x_estimate)

             # labe 3 - stopping at predetermined points
             # if abs(self.x_estimate - 0.61) < tolerance or abs(self.x_estimate - 1.22) < tolerance or\
             #           abs(self.x_estimate - 2.44) < tolerance or abs(self.x_estimate - 3.05) < 0.001:
             #     # old_twist = twist
             #     for i in range(2*r):    # 2 * 120 Hz = 240 cycles
             #           twist.linear.x = 0
             #           twist.angular.z = 0
             #           self.cmd_pub.publish(twist)
             #           print("speed x: ", twist.linear.x)
             #           rate.sleep()
             #     # twist = old_twist

             twist.linear.x = speed
             self.cmd_pub.publish(twist) #go forwards + do angular twist correction
             # print("speed x: ", twist.linear.x)
             lasterror = error

             rate.sleep()


if __name__ == "__main__":
 rospy.init_node("lab3")
 controller = Controller()
 controller.pid_control()

```

Final_project.py

```
#!/usr/bin/env python
import rospy
import math
from geometry_msgs.msg import Twist
from std_msgs.msg import String, Float64MultiArray, Float64
import numpy as np
import colorsys


class BayesLoc:
 def __init__(self, p0, colour_codes, colour_map):
```

```python
        self.colour_sub = rospy.Subscriber(
            "mean_img_rgb", Float64MultiArray, self.colour_callback
        )
        self.line_sub = rospy.Subscriber("line_idx", String, self.line_callback)
        self.cmd_pub = rospy.Publisher("cmd_vel", Twist, queue_size=1)
        self.state_pub = rospy.Publisher("state", Float64, queue_size=1)

        self.num_states = len(p0)
        self.colour_codes = colour_codes    # reference rgb values
        self.colour_map = colour_map        # list of office colours
        self.probability = p0
        self.state_prediction = np.zeros(self.num_states)

        self.cur_colour = None  # most recent measured colour

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

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

    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)

        TODO: complete this function
        """

```

```
      # state model p(x_k+1 | x_k = X, u_k)
      p_state =  [[0.85, 0.10, 0.05],  # u = -1
                  [0.05, 0.90, 0.05],  # u =  0
                  [0.05, 0.10, 0.85]]  # u = +1
                  # X-1 X     X+1

      # p_state =  [[0.85, 0.10, 0.05],  # u = -1
      #           [0.05, 0.90, 0.05],  # u =  0
      #           [0.05, 0.30, 0.65]]  # u = +1
      #           # X-1 X     X+1

      return p_state[u+1]


def measurement_model(self, x):
      """
      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?
      """
      if self.cur_colour is None:
            self.wait_for_colour()

      prob = np.zeros(len(colour_codes))

      """
      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).
      """

      # given rgb values
      r, g, b = self.cur_colour
      # and reference rgb values
      # compute distance between reference and each rgb value
      distances = np.zeros(len(colour_codes))
      for i in range(len(colour_codes)):
            distances[i] = 1/(((r-self.colour_codes[i][0])**2 + (g-self.colour_codes[i][1])**2 + (b-self.colour_codes[i][2])**2)**0.5)
```

```
        # sum all the distances
        total = sum(distances)
        # return each distance divided by the total sum
        prob = distances/total
        return prob

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

        u = 1
        p_state = self.state_model(u)   # don't know what u is
        for i in range(self.num_states):
            s = 0
            # x_k+1 is X-1
            s += p_state[0] * self.probability[(i+1)%self.num_states]
            # x_k+1 is X (stays in the same state)
            s += p_state[1] * self.probability[i%self.num_states]
            # x_k+1 is X+1 (state is the next one)
            s += p_state[2] * self.probability[(i-1)%self.num_states]
            # put into state prediction
            self.state_prediction[i] = s

    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
        """

        state_update = np.zeros(self.num_states)
        p_measure = self.measurement_model(self.probability)
        for i in range(self.num_states): #for each x_k+1 (11 offices)
            s = 0   # initialize sum for the denominator
            for j in range(self.num_states): # sum for denominator for each state
```

```
             s += p_measure[self.colour_map[j]] * self.state_prediction[j]
             state_update[i] = p_measure[self.colour_map[i]] * self.state_prediction[i] / s

       self.probability = state_update
       state = np.argmax(localizer.probability)
       self.state_pub.publish(state/2)


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]
 colour_map = [3,4,1,4,2,4,0,4,0,1,4,2,4,0,4,3,4,1,4,2,4]

 colours = ['r', 'g', 'b', 'y', 'line']

 # 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 = [
 #     [203.3, 109.76, 147.2],  # red
 #     [120, 139, 130],  # green
 #     [161.5, 152, 172],  # blue
 #     [186.7, 170.6, 150.5],  # yellow
 #     [175, 158, 162],  # line
 # ]
```

```

colour_codes = [[203.3, 109.76000000000002, 147.20000000000002],
        [120.0, 139.0, 130.0],
        [133.91459640000002, 121.95681, 143.31],
        [186.7, 170.6, 150.5],
        [174.42000000000002, 158.268708, 158.94706226400004]]


colour_codes_hsv = [
        [0.9332905708787685, 0.4601082144613871, 0.7972549019607844],
        [0.42105263157894735, 0.13669064748201432, 0.5450980392156862],
        [0.76, 0.149, 0.562],
        [0.09254143646408837, 0.19389394750937322, 0.732156862745098],
        [0.993, 0.0926, 0.684]
]

# colour_codes_hsv = [[0.9332905708787685, 0.4601082144613871, 0.7972549019607844],
#                       [0.42105263157894735, 0.13669064748201432, 0.5450980392156862],
#                       [0.7458333333333332, 0.11627906976744193, 0.6745098039215687],
#                       [0.09254143646408837, 0.19389394750937322, 0.732156862745098],
#                       [0.9607843137254903, 0.09714285714285711, 0.6862745098039216]]

# initial probability of being at a given office is uniform
p0 = np.ones_like(colour_map) / len(colour_map)

localizer = BayesLoc(p0, colour_codes, colour_map)

rospy.init_node("final_project")
rospy.sleep(0.5)
rate = rospy.Rate(10)

last_cols = np.zeros(5)

last_state = 0  # 0 for line, 1 for not line
cur_state = 0

localizer.state_predict()
localizer.state_update()
```

```

while not rospy.is_shutdown():
      cols = localizer.measurement_model(localizer.probability)
      cur_state = np.argmax(cols) != 4
      if not last_state and cur_state:
            localizer.state_predict()
            localizer.state_update()
      last_state = cur_state

      print(np.around(localizer.probability, 2))
      rate.sleep()

# i = 0

# while not rospy.is_shutdown():
#     if i > 15:
#           localizer.state_predict()
#           localizer.state_update()
#           i = 0
#     print(np.around(localizer.probability, 2))

#     i += 1

#     rate.sleep()

# 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
#     """

#     np.append(last_cols, colour_map[np.argmax(localizer.probability)])
#     np.delete(last_cols, 0)

#     cur_state = np.argmax(localizer.probability)/2

#     if np.min(last_cols) == np.max(last_cols):
#           # confident we're at an office
#           print(cur_state)
#           print(last_state)
#           if cur_state % 1 == 0 and last_state % 1 != 0:
```

```
#           localizer.state_predict()
#           localizer.state_update()

#           last_state = cur_state
#           # print(colours[colour_map[np.argmax(localizer.probability)]])
#           # state = str(np.argmax(localizer.probability))
#           # localizer.state_pub.publish(state)

#     print(np.around(localizer.probability, 2))
#     rate.sleep()

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

```