Rob301 Final Project Report

Cameron Witowski (1005533166) and Mingshi Chi (1004881096) Group 12 PRA101

Introduction

The purpose of this robotics project was to simulate a robot delivering mail to different offices in the Faculty of Applied Science and Engineering's Galbraith Building. The offices to be simulated are situated in a closed-loop square with the doors on the inside of the loop as shown below in Figure 1.

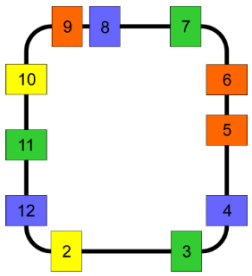


Figure 1: Map of the simulated path of delivery

The objective of the project is to design a control system based on Bayesian-localization techniques to navigate through the path when starting at a random location and simulate package delivery at the desired offices by rotating inwards 90 degrees, pausing, and rotating back to continue on its path on the black line.

Robot Platform

The platform used to prototype the Galbraith Memorial Mail Robot was the TurtleBot 3 Waffle Pi, running the Robotic Operating System (ROS) in a simulation environment called Gazebo. Under the sense-think-act framework, the platform is quite simplistic. There is one sensor used on the Turtlebot, a camera, there is Python running on ROS for the logic, and there is one means of actuation, velocity input to the robot's motors. The camera data is then broken down into two channels, or topics: (1) a color topic, carrying the RGB values of the camera's view, and (2) a line following topic, the latter of which returns the horizontal location of the line to follow. The velocity input is also broken down into linear forward velocity and angular velocity, essentially embodying the unicycle model. A flowchart illustrating this entire system is presented below.

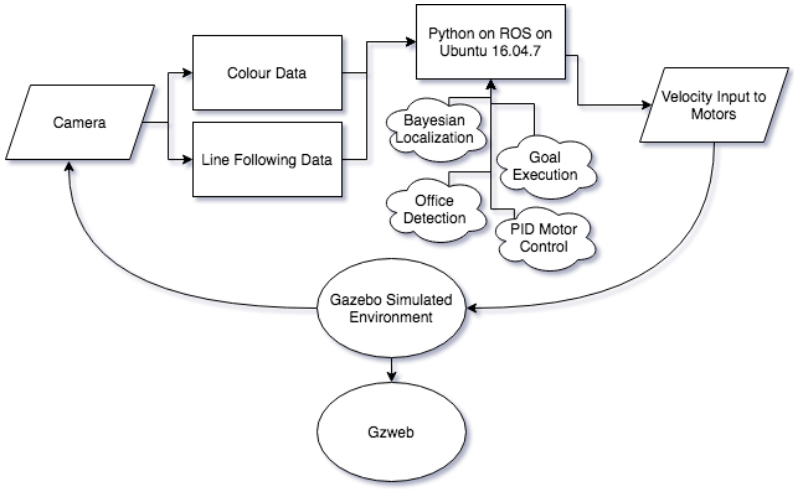


Figure 2: High Level System Design

Solution Strategy

The general approach is shown in Figure 2. For general movement around the closed loop, the idea was to use the PID control found in the follow_the_line() method in Appendix[1]. This method uses PID control with tuned parameters from Lab 3 to follow the line. Since the camera returned the RGB values for what it sees, the first attempt was to only use PID control if the color returned was within the same range as the color of what the camera saw when it sees only the black line. If it doesn't see the line, it would drive straight. However, this method resulted in the robot following any edge which included following the edge of the colored offices. To fix this, the strategy was changed to using PID if it doesn't see color as the color value of the line is fairly close to white. Some tuning was necessary to find the right threshold between the colored offices and the color of the line.

After identifying the color, the robot would pause in the middle of the line by driving straight without using the PID control by using a timer which was also tuned using trial and error.

Using Bayesian-localization method and a predefined map, the robot would use where it thinks it is to execute package delivery at the predefined target offices. It would turn 90 degrees counterclockwise and 90 degrees back and drive straight until it passed the second edge of the office to continue it's path using PID control to avoid following the office door edge.

For color detection, the color detected would be determined by finding the closest color code to the color value received from the sensor to account for noise.

Methodology

To spell out the specific technical implementation details of our design, we will go walk through each node on the flowchart in figure 2.

1) Camera, Colour Data and Line Following Data (Sense)

The Turtlebot 3 Waffle Pi is equipped with a Raspberry Pi 8MP camera which is used as the sole sensor for our mail delivery system. The ROS node for colour data and line following data was provided by the instructors, so we will not go into detail about them. What is important though is that as black boxes these nodes take in a camera image and output a single RGB data point representing the colour the camera sees and an integer from 0-640 representing the horizontal location of a detected line. In the following nodes this data is massaged to deal with edge cases, for instance if no line is detected.

2) ROS Python, running on Linux Ubuntu (Think).

We put this section as separate from the colour/line nodes since the following is entirely designed and implemented by us. The thinking processes of our Mail Delivery robot can be broken down into four components, going from low-level to high-level.

a) PID Motor Control.

Assuming the robot is on, or nearby a line, we invoke the following algorithm to follow it. Taking as input the horizontal location of the line as detected by the camera (an integer from 0-640), we define the error as the difference between this number and 320 (the centre of the camera's view). As the turtlebot proceeds, we compute the error, the derivative of the error (difference between the error of each step and the error of the previous step), and the integral error (sum of all errors so far). With this information in hand, we set a constant linear velocity = 0.10 m/s and an angular velocity some linear combination of these three errors. We found the best constants, or tuning gains, to be $K_p = 0.0055$, $K_i = 0.001$, and $K_d = 0.017$. The relevant section of code can be found in Appendix A, in the line_callback and follow_the_line functions.

b) Office Detection.

PID control works great when robust line measurements are available, but what happens when no line is detected? Through testing of the PID algorithm, it was found to be highly reliable until the Turtlebot reached an office. Then, the camera no longer sees a line to follow but instead sees a block of colour. With no modifications the Turtlebot twists and turns like a blind mouse upon reaching an office.

To remedy this, we found a reliable method through trial and error of detecting when the robot has reached an office, by using the colour information. We tested several logic schemes, and many turned out to fail. For instance, the colour output when the robot is following a line properly is about (223, 223, 223). We first tried to set a tolerance such that if a colour was detected that is more than 10 away from this line colour, in any of the RGB channels, we say the robot is at an office. The idea behind this is that when the Turtlebot reaches an office, it detects some colour that is not grey. While this worked on some trials, to our surprise, the Turtlebot would sometimes follow the outline of the office as though it were its intended path.

What worked best was the following: if any of R, G, or B is <= 210, then the robot is at an office. This proved robust in all cases.

When the Turtlebot is at an office, we stop the PID algorithm and set angular velocity to 0 and set linear velocity constant (unless a delivery is being made, which is explained later). We also set a timer such that even if an outlier measurement appears, the Turtlebot does not start the PID control back up until at least 9 seconds after reaching an office (longer if making a delivery). This value was determined experimentally as the time taken to pass through an office.

c) Bayesian Localization.

After developing a robust way to detect when the Turtlebot is at an office, the first step before running the localization algorithm is to measure a colour. To do this, we take the euclidean distance between the camera output RGB value and the 4 separate possible office colours.[1] The smallest distance is the measured colour. This strategy was found experimentally to be incredibly reliable, even with noise in the colour data. We were able make correct measurements around 90% of the time.

With office detection and measurements in hand, implementing the Bayesian localization algorithm is as simple as following a cookbook. We broke the algorithm down into 3 steps, the code for which can be found in Appendix A, in the measurement_model, statePredict, and stateUpdate functions. Each time an office is detected, stateUpdate is called, which itself calls measurement_model then statePredict.

The measurement model we implemented captures the confidence in our measurements we gained through experience. Put simply, we defined the probability that a given measurement is correct as 88%, and the probability that it is incorrect is 12%—4% for each of the other 3 colours. In other words, if the Turtlebot is at a green office, it has an 88% chance of detecting green, 4% for orange, 4% for purple/blue, and 4% for yellow. These numbers were chosen since they depict our strong measurement confidence, they can still handle the occasional incorrect measurement, and they are nice whole numbers.

The state predict function is included almost as a formality, since it is meant to capture the uncertainty between our control input and real world dynamics. In our simulation however, we are always moving forward (never staying still and never going back), and we have a robust way of detecting offices. Thus, at each subsequent call to statePredict, it is just about certain that the Turtlebot has moved forward one office. Thus, the forward = 1 if-clause is the only one that runs in our demo. In this clause we set the probability of going forward at 95% and the probability of staying still at 5%. This is to handle the odd case that the robot detects the same office twice, even though this seldom happens in our simulated trials. The statePredict function would have a much greater use if path planning is implemented, which is discussed in the Potential Improvements section.

Finally, the stateUpdate function simply multiplies the state prediction by the measurement model, then normalizes. More specifically, set each array index x of the output pdf = (the predicted probability of being in that position) times (the probability of getting the measurement we got if we were there). Note that this function references and relies on an accurate topological map of office locations.

The algorithm as stated proves highly robust, and quickly settles to over 99% positional confidence after just 3 or 4 measurements.

d) Goal Execution.

---

[1] We do not consider the possibility of the Turtlebot detecting a “line coloured” office, as suggested in the project outline pdf. This is due to the fact that our office detection was determined to be so robust that if we are making measurements, we are definitely at an office with some non-grey colour.

Goal execution is the analog of high level planning in our mail delivery system, albeit extremely simplistic in our proof of concept. The basic logic for delivering packages is illustrated in the flowchart below.

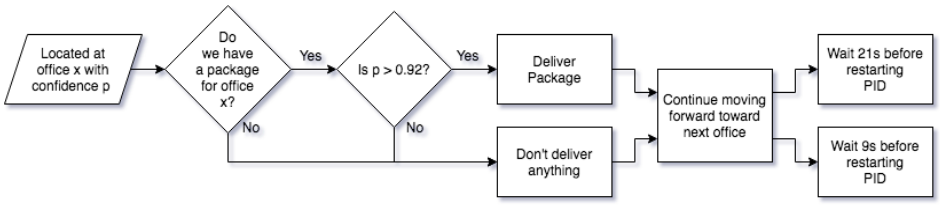


Figure 3: Flow of logic for package delivery.

To initialize this execution algorithm, all we must do is define a list of office indexes for which we have packages to deliver. The numbers 0.92, 21s, and 9s were all determined experimentally to work adequately. Note that the "Deliver Package" node is a motor control subroutine and can be found in Appendix A, in the deliver function. Improvements to our mail delivery robot centre around better ways to complete this goal execution, and are discussed later.

3) Velocity Input To Motors (Act).

Finally, the system will publish the desired velocities to the cmd_vel topic. These will be one of three kinds, determined by all of the above programming logic:

a) PID motor control.
b) Constant forward velocity through an office.
c) Delivery subroutine.

4) Gazebo Simulation Environment and Gzweb.

Under the hood, the Gazebo software is used to simulate all of the robot's behaviour and real world dynamics as well as the map of offices. This simulation is published to a server and can be viewed through any browser on the U of T network (or anywhere in the world with the U of T VPN). A picture of this simulation running is in Appendix B.

Demo Performance

During the first demonstration, the turtlebot came across the issue of following every edge including the edges of the office doors when it did not the day before during testing. This affected the movement of the rover significantly and the adjustments of color and line detection described in Methodology Part b was implemented before the second trial.

During the second demonstration, all deliveries were successful and the turtlebot did not run into any issues with movement or localization.

Screenshots of the demo can be seen in Appendix B.

Potential Improvements

If additional time, resources, and motivation were directed toward improving our mail delivery robot, the most wanting aspects of our design would be in goal execution.

1) Goal Execution

The mail delivery robot's planning is incredibly simplistic and could use an overhaul. Currently, the robot drives in a circle around the offices until all packages are delivered with 92% confidence. This scheme is linear, static, and riddled with potential improvements:

- If the Turtlebot is at position 2, and has a package for position 1, it would be much faster to turn around rather than going all the way around the circle.
- If the Turtlebot is at position 2, and has a package for position 2, but it's confidence is slightly below 92%, it should be able to look at a neighbouring office to better situate itself, then come back to deliver the package.

- All things equal, the Turtlebot should prefer unique paths over non unique ones, in the interest of increasing positional confidence as soon as possible. As an example, if you were lost in Toronto, seeing a 7-11 beside a Tim-Horton's would not help you become situated (since this sequence of buildings is all over the place), but seeing a second cup right beside a spicy mafia restaurant would tell you exactly where you are (the bottom of King's College Rd., in this case).

Finding the best path for the robot to take is actually implementable with relative ease as a discrete optimization problem as follows:

Minimize the expected value of the length of a vector **x**, corresponding to control inputs -1 or 1, such that for every office-to-deliver-to, the robot is there with positional confidence greater than 92%.

More specifically,

$$\min_x E(\|\mathbf{x}\|_2) \text{ subject to } \|\mathbf{P}\mathbf{d_i}\|_\infty \geq 0.92 \ \forall \ i \in [1, m]$$

Where $E(\|\mathbf{x}\|_2)$ is the 2 norm of delivery path, which works out to be equal to the length since all entries are -1 or 1. **P** is a matrix such that row j is the discrete locational pdf at timestep j, which is a function of **x** according to the Bayesian localization algorithm. $\mathbf{d_i}$ is a column vector of length 11, with a 1 in the index of package i's destination office. The infinity norm is essentially the max element of vector $\mathbf{P}\mathbf{d_i}$. **P** is a function of **x** according to the bayesian localization algorithm. The number of packages to deliver is m.

To properly run this optimization algorithm, we would also need to define some tolerance below which probabilities are ignored; the probability is nonzero that the Turtlebot records wrong measurements every time and is never confident enough to deliver a package—but we should ignore this in computing the expected value.

We would run this optimization at each time step, then move the robot in the direction of **x**[0]. Such an algorithm would be a massive improvement over the current brute force strategy of package delivery.

Conclusion

The final project was a success. The robot was able to simulate package delivery around the Galbraith Building. The robot successfully predicted it's position throughout time and it's confidence level in its current position was increased throughout time.

The difficulties encountered in each deliverable served as a learning experience as well as great opportunities to think outside the box in problem solving. Although there were many ways to approach each problem, we ultimately chose the one that was easiest to implement as well as create robustness in the robot's delivery of the package.

Appendix A: Final Code

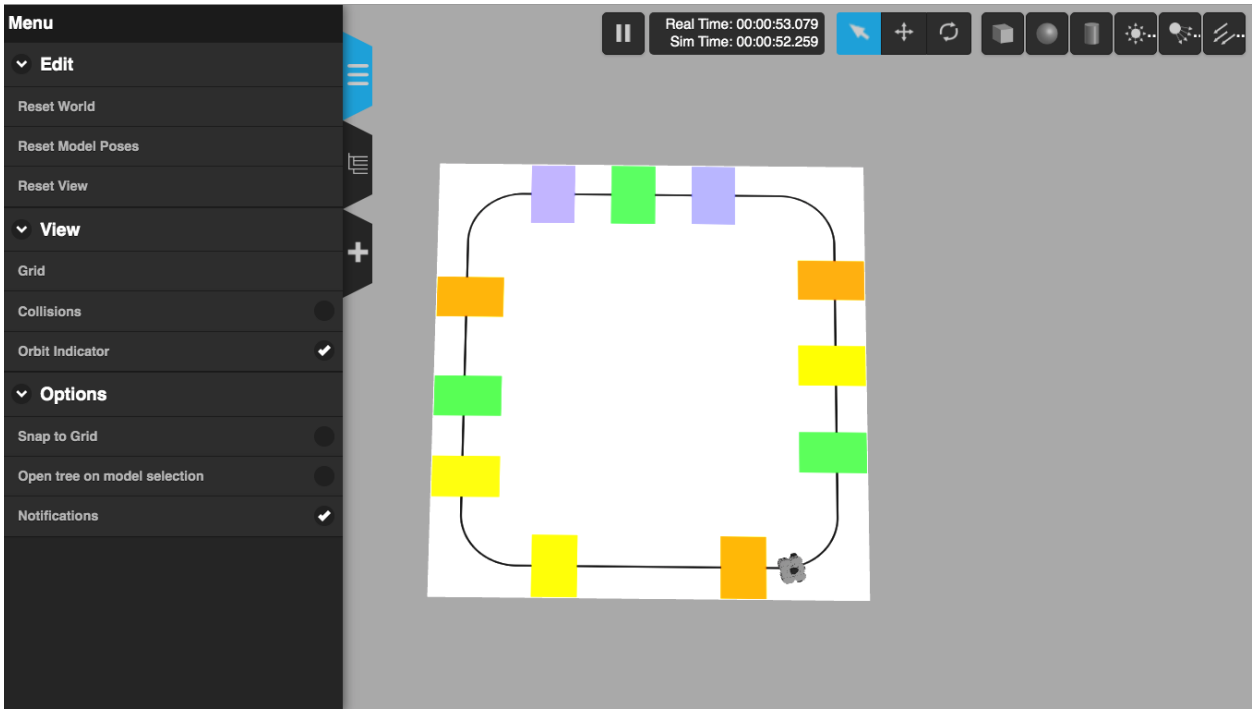
```
#!/usr/bin/env python
import rospy
import math
import time
from geometry_msgs.msg import Twist
from std_msgs.msg import String
import numpy as np
import re
import sys, select, os

if os.name == 'nt':
    import msvcrt
else:
    import tty, termios

def getKey():
    if os.name == 'nt':
      return msvcrt.getch()
    tty.setraw(sys.stdin.fileno())
    rlist, _, _ = select.select([sys.stdin], [], [], 0.1)
    if rlist:
        key = sys.stdin.read(1)
    else:
        key = ''
    termios.tcsetattr(sys.stdin, termios.TCSADRAIN, settings)
    return key


class BayesLoc:

    def __init__(self, P0, colourCodes, colourMap, transProbBack, transProbForward):
        self.colour_sub = rospy.Subscriber('camera_rgb', String, 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.probability = P0 ## initial state probability is equal for all states
        self.colourCodes = colourCodes
        self.colourMap = colourMap
        self.transProbBack = transProbBack
        self.transProbForward = transProbForward
        self.numStates = len(P0)
        self.statePrediction = np.zeros(np.shape(P0))

        self.CurColour = None ##most recent measured colour

        self.error = 0
        self.errori = 0
        self.lastError = 0
        self.twist = Twist()
```

```
        self.twist.linear.x = 0.10
        self.twist.angular.z = 0
        self.h = 0.15
        self.coolDown = 0

        self.colorCnt = 0
        self.rate = rospy.Rate(50)

    def colour_callback(self, msg):
        rgb = msg.data.replace('r:','').replace('b:','').replace('g:','').replace(' ','')
        r,g,b = rgb.split(',')
        r,g,b=(float(r), float(g),float(b))
        self.CurColour = np.array([r,g,b])

    def line_callback(self, msg):
        self.lastError = self.error
        self.error = -int(msg.data) + 320
        if (int(msg.data) != 0):
            self.errori += self.error*self.h
        self.follow_the_line()
        pass

    def follow_the_line(self):
        err = self.error
        erri = self.errori
        errd = self.error-self.lastError
        kp = 0.0055
        ki = 0.001
        kd = 0.017
        if (self.CurColour[0] >= 210 and self.CurColour[1] >= 210 and self.CurColour[2] >= 210):
            if (rospy.get_time() - self.coolDown >= 9):
                self.twist.angular.z = kp*err + ki*erri + kd*errd
        else:
            self.twist.angular.z = 0
            self.cmd_pub.publish(self.twist)
            if (rospy.get_time() - self.coolDown >= 13):
                rospy.loginfo("Colour Detected!!! Stopping line following temporarily")
                self.coolDown = rospy.get_time()
                self.twist.linear.x = 0
                rospy.sleep(3)
                self.cmd_pub.publish(self.twist)
                rospy.sleep(3)
                self.colorCnt = self.colorCnt + 1
                self.stateUpdate()
                maxIndex = 0
                for i in range(0,11):
                    if self.probability[i] >= self.probability[maxIndex]:
                        maxIndex = i
                rospy.loginfo("At position: " + str(maxIndex+1) + ", with prob " + str(self.probability[maxIndex]))
                if maxIndex in (7,9,0) and self.probability[maxIndex] > .92:
                    self.deliver()
                    self.coolDown += 12
                self.twist.linear.x = 0.10
        self.cmd_pub.publish(self.twist)
        self.rate.sleep()
        return
```

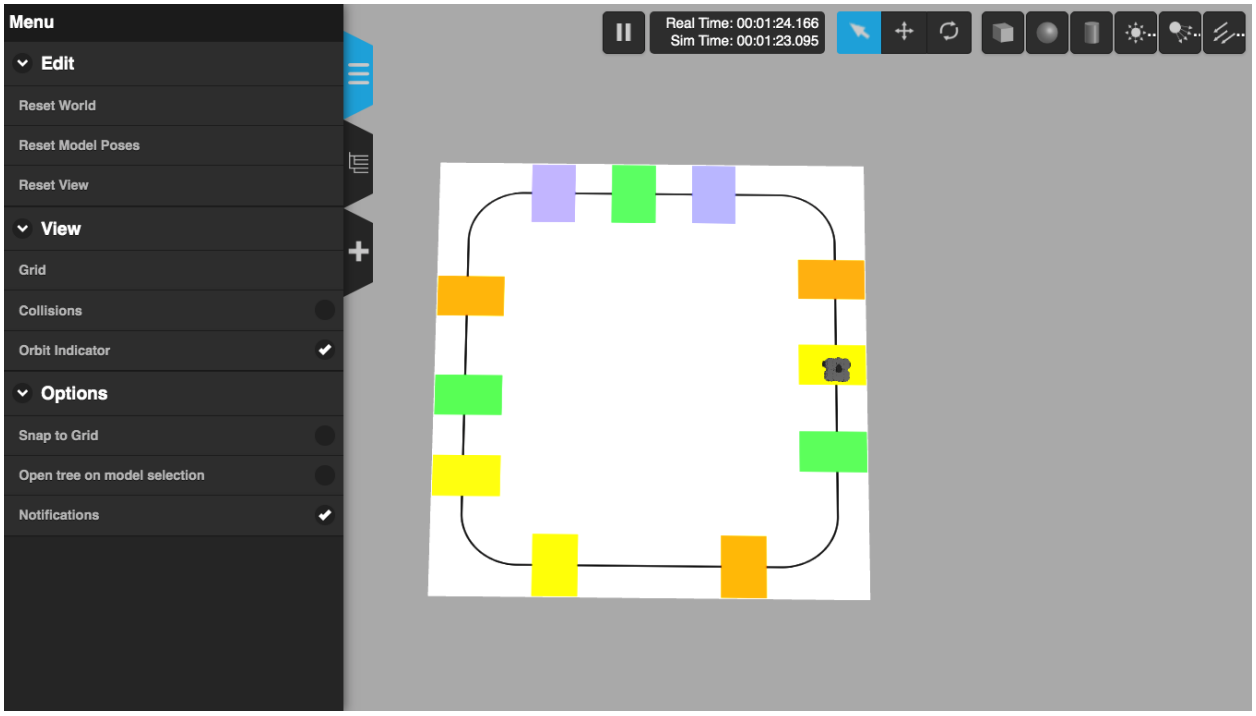
```
    def deliver(self):
        self.twist.linear.x = 0
        self.twist.angular.z = math.pi/12
        self.cmd_pub.publish(self.twist)
        rospy.sleep(6)
        self.twist.angular.z = 0
        self.twist.angular.z = -math.pi/12
        self.cmd_pub.publish(self.twist)
        rospy.sleep(6)
        self.twist.angular.z = 0
        self.twist.linear.x = 0.1
        self.cmd_pub.publish(self.twist)

    def waitforcolour(self):
        while(1):
            if self.CurColour is not None:
                break

    def measurement_model(self):
        if self.CurColour is None:
            self.waitforcolour()
        prob=np.zeros(11)
        dist=np.zeros(4)
        minIndex = 0

        for i in range(0,4):
            dist[i] = (self.colourCodes[i][0]-self.CurColour[0])**2 + (self.colourCodes[i][1]-self.CurColour[1])**2 +
(self.colourCodes[i][2]-self.CurColour[2])**2
            if (dist[i] <= dist[minIndex]):
                minIndex = i
        rospy.loginfo(minIndex)
        for i in range(0,4):
            if (i == minIndex):
                prob[i] = 0.88
            else:
                prob[i] = 0.04
        return prob

    def statePredict(self,forward):
        self.statePrediction = np.zeros(11)
        for i in range(0,11):
            if forward == 1:
                self.statePrediction[(i+1)%11] +=  0.95*self.probability[i]
                self.statePrediction[i] += 0.05*self.probability[i]
                self.statePrediction[(i-1)%11] += 0*self.probability[i]
            elif forward == 0:
                self.statePrediction[(i+1)%11] += 0.05*self.probability[i]
                self.statePrediction[i] += 0.9*self.probability[i]
                self.statePrediction[(i-1)%11] += 0.05*self.probability[i]
            else:
                self.statePrediction[(i+1)%11] += 0.05*self.probability[i]
                self.statePrediction[i] += 0.1*self.probability[i]
                self.statePrediction[(i-1)%11] += 0.85*self.probability[i]

    def stateUpdate(self):
        self.statePredict(1)
        prob = self.measurement_model()
```

```
        norm = 0
        for i in range(0,11):
            norm += self.statePrediction[i]*prob[self.colourMap[i]]
        for i in range(0,11):
            self.probability[i] = self.statePrediction[i]*prob[self.colourMap[i]]/norm

if __name__=="__main__":
    if os.name != 'nt':
        settings = termios.tcgetattr(sys.stdin)

    # 0: Green, 1: Orange, 2: Purple, 3: Yellow, 4: Line
    color_maps = [3, 1, 0, 3, 1, 2, 0, 2, 1, 0, 3] ## current map starting at cell#2 and ending at cell#12
    color_codes = [[72, 255, 72], #green
                   [255, 174, 0], #orange
                   [145,145,255], #purple
                  [255, 255, 0], #yellow
                   [133,133,133]] #line

    trans_prob_fwd = [0.1,0.9]
    trans_prob_back = [0.2,0.8]

    rospy.init_node('final_project')
    bayesian=BayesLoc([1.0/len(color_maps)]*len(color_maps), color_codes, color_maps, trans_prob_back,trans_prob_fwd)
    prob = []
    rospy.sleep(0.5)
    state_count = 0

    prev_state=None
    try:

        while (1):
            key = getKey()
            if (key == '\x03'):
                rospy.loginfo('Finished!')
                rospy.loginfo(prob)
                break

    except Exception as e:
        print("comm failed:{}".format(e))

    finally:
        rospy.loginfo(bayesian.probability)
        cmd_publisher = rospy.Publisher('cmd_vel', Twist, queue_size=1)
        twist = Twist()
        cmd_publisher.publish(twist)
```

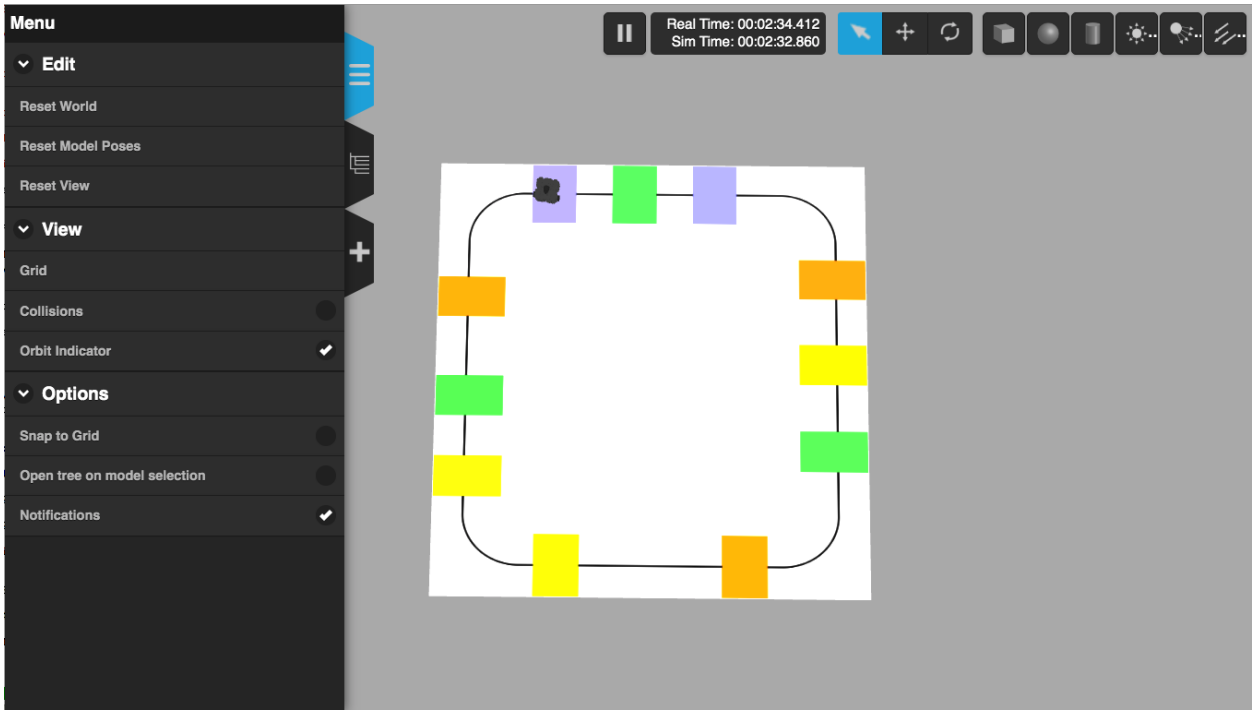
Appendix B: Simulation Screenshots and Demo



The Turtlebot follows the line by using PID control.



The Turtlebot has no trouble passing through offices.



The Turtlebot is making a delivery at office 9!

```
[INFO] [1606965397.574247, 40.209000]: Colour Detected!!! Stopping line following te
mporarily
         [INFO] [1606965403.636580, 46.211000]: 1
                                                  [INFO] [1606965403.636889, 46.21100
0]: At position: 9, with prob 0.297297297297
                                                [INFO] [1606965422.322293, 64.755000]: C
olour Detected!!! Stopping line following temporarily
                                                       [INFO] [1606965428.355549, 70.7
56000]: 0
         [INFO] [1606965428.355858, 70.756000]: At position: 10, with prob 0.4727664
37583
     [INFO] [1606965435.518768, 77.815000]: Colour Detected!!! Stopping line followi
ng temporarily
              [INFO] [1606965441.527379, 83.817000]: 3
                                                       [INFO] [1606965441.527694, 83.
817000]: At position: 11, with prob 0.496902966849
                                                     [INFO] [1606965448.638368, 90.9020
00]: Colour Detected!!! Stopping line following temporarily
                                                          [INFO] [1606965454.678211
, 96.902000]: 1
               [INFO] [1606965454.678516, 96.902000]: At position: 5, with prob 0.94
7776001153
           [INFO] [1606965473.266329, 115.326000]: Colour Detected!!! Stopping line f
ollowing temporarily
                  [INFO] [1606965479.279350, 121.326000]: 2
                                                            [INFO] [1606965479.2796
65, 121.326000]: At position: 6, with prob 0.994972921793
                                                          [INFO] [1606965486.452435,
128.419000]: Colour Detected!!! Stopping line following temporarily
                                                                [INFO] [160696549
2.523356, 134.420000]: 0
                       [INFO] [1606965492.523653, 134.420000]: At position: 7, with
 prob 0.995389894178
                  [INFO] [1606965499.680449, 141.543000]: Colour Detected!!! Stopp
ing line following temporarily
                            [INFO] [1606965505.701114, 147.543000]: 2
                                                                    [INFO] [16069
65505.701379, 147.543000]: At position: 8, with prob 0.997261654386
```

Console printouts from the above run. Some notes:

1) Apologies that the indentation is indented weird.
2) The position numbering is one less than the defined numbers in project.pdf. So the bottom left yellow office is position 1, the bottom right orange office is position 2, etc.
3) The start position for this run was at the bottom middle (after *position* 1).
4) Three things are printed each time an office is reached:
    a) "Colour detected! Stopping line following"
    b) An index representing the measured colour:
        i) 0 = Green.
        ii) 1 = Orange.
        iii) 2 = Purple/blue.
        iv) 3 = Yellow.
    c) The position with the highest probability, and the corresponding probability.
5) We can see that by the fourth office the positional confidence is already at 94%, and at position 6 settles to >99%.
6) The pdf evolution is highly logical and expected. After detecting orange first, the robot is equally likely to be at 2, 5, or 9 (so 29% confidence makes sense). After detecting the sequence orange-green, the robot is equally likely at position 3 or position 10 (so 47% confidence makes sense). After detecting the sequence orange-green-yellow, it's still equally likely between positions 4 and 11. But after the 4 color sequence orange-green-yellow-orange is detected, the robot must be at position 5, unless there was an incorrect measurement or an office was measured twice (both of which are experimentally unlikely, so 94% makes perfect sense here).