The Final Design Project Report

The Galbraith Memorial Mail Robot

Group 13: Rayhaneh Behravesh, Antoine Minjon

Introduction

The aim of this project is to develop a control system based on the bayesian localization techniques to deliver mail to arbitrarily chosen offices (coloured and numbered nodes) on a closed loop path, starting at an arbitrarily chosen initial location. The turtlebot should stop for 2 seconds on the chosen offices. A state/office is represented by a coloured node (red, blue, green, yellow) in the topological map in figure 1. The black line is the path the turtlebot should line-follow. To achieve this task we first had to achieve full-route execution without bayesian localization and then implement localization and determine the convergence of the state estimation. Finally, we stop at the desired nodes given high confidence rate in estimation, which was ensured after a full traversal of the route.

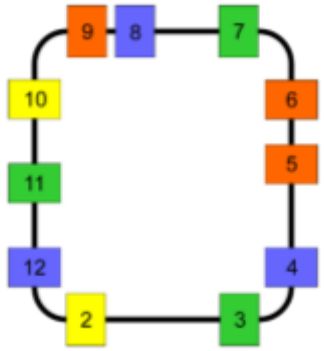

Fig 1 : topological map used for testing

Hardware

We used a Turtlebot 3 Waffle Pi (Fig 2) for this project, this is a holonomic robot. We used the two motors attached to the wheels to control the robot's movement. As for the sensors, we only used the Raspberry Pi camera which permits RBG colour perception. The camera is located in front of the robot and points downwards, used to identify the black line on the ground and the colour of the node. The OpenCR 1.0 control board controls the motion and wheels of the turtlebo and contains the Inertial Measurement Unit (IMU) and gyroscope. To establish communication between the turtlebot and the PC, the bot has a bluetooth module for remote controller.

Fig 2 : Turtlebot 3 Waffle Pi

Strategy

To solve this problem we have two main issues to solve:

- The robot has to know where it is on the topological map. So we need to implement a program to localize the robot on the topological map using the camera and the input given to the motors.
- The robot has to follow the predetermined path, so we need to implement a controller to follow the line using the image given by the camera.

Implementation

For localizing the robot we used bayesian localization. We represented the probability to be at each node as a vector. We used a matrix for the state model and a vector for the measurement model to make an easier calculation to update the state estimate.
For the controller, we used a PID controller because it's a good and easy way to make the robot follow the predetermined path. We use the camera to find the distance between the trajectory of the robot and the predetermined path.

There were many considerations before the demo. Initially, the biggest challenge we had was correct colour interpretation from the camera reading. At first, we didnt convert to HSV (hue, saturation, value) values and implemented measurement update using the RGB readings. However, the robot was not able to distinguish the colours from the line, meaning our PID controller would cause a disturbance in the trajectory when the robot arrived at a coloured node. This challenge became further complicated when we were faced with wifi issues, causing delayed communication with the robot. This caused the robot to not perform as needed, and spin around itself when encountering a coloured node. To solve this problem we converted the values to HSV and decreased the speed of the robot to 0.05 which somewhat compensated for the slow wifi connection and better implementation of the PID control. We found that it was easier to tune the P, I &D values for slower speeds from lab 3. The method we used to determine what colour were on was taking the norm of the difference between the detected colour and all the colours in the `colour_code` matrix:

```
diff =  np.linalg.norm(colour_codes[i]-self.cur_colour)
```

The colour with the smallest norm would be chosen as the detected colour.

Fig 3 : State model matrix

Once we were able to achieve the desired accuracy in line following and colour detection, we proceeded to develop code to have the robot execute the full route. We did this by setting `twist.angular.z=0` every time a colour was detected. Additionally, we set the integral gain to 0 to prevent the long-term build-up of error that could disturb line following. We had a lot of challenges in this section and spent multiple sessions to tweak PID, colour codes and the ROS code.

After getting the robot to traverse the entire path, we implemented the Bayesian localization node. Having developed the python code to implement bayesian localization in part 1 of the project, we modified and implemented it into our ROS code. We performed matrix multiplication between the state model and the probability array to find the state prediction before any measurements. The state model consisted of three matrices (fig. 3) with each matrix representing the state model for speeds 1,0 and -1 respectively. As the robot was always going forward we used the 1st matrix (note that we realized that we must multiply by the transpose of the matrices and hence numbered them backward: 1,0,-1 opposed to

-1,0,1 to prevent additional changes). Multiplying this with our state prediction was a quick way to determine the new predicted state. Note that the initial state prediction is the array `np.array( [1/11]*11)` which holds a uniform distribution for the prediction due to no initial knowledge of the state.

For our measurement update, first we find the distance between the colour measured by the camera and each theoretical colour for each node. After normalization, we get the probability to be on each colour in a vector. To get the measurement vector, we multiply the probability vector by a matrix M (fig 4), the ith element represents the probability to be at node *i,* knowing only the measurement made by the camera. To update our state prediction we just need to make the Hadamard product between the measurement vector and our state prediction and then normalize it. Note that multiplying by M allows for matrix multiplication since without it the dimensions dont match.

Fig 4 : Matrix M

```
self.state_prediction=np.multiply(np.matmul(M,self.measure),self.state_prediction)
self.state_prediction=self.state_prediction/(sum(self.state_prediction))
```

An important consideration we made was to ensure that the state is not constantly updated as the robot travels on the coloured nodes. To do that, we used an if statement which checked the following two conditions:

1. The previously measured colour is not equal to the current measured colour
2. The previously measured colour is a line

We only performed the state prediction and update if the conditions of the if-statement was met. This way, we ensured one update per coloured node (condition 1). We recognized that there is a probability that the robot may misinterpret different colours while traversing it and so condition 2 ensures that state prediction is not performed if there are two back-to-back readings of different colours.

To stop the robot at the locations s1, s2, s3 we use our state prediction model. We determine the most probable state by choosing the maximum probability in state prediction model:

```
max_prob = np.max(self.state_prediction)
```

We can further associate that with the office numbers by using the function `np.argmax()` to determine the index of the maximum element and add 2 as the topological map starts from node #2. We further check two conditions which determine if we should stop or not:

1. If the most probable index is in the list [s1,s2,s3]
2. If `max_prob` is greater than 80%

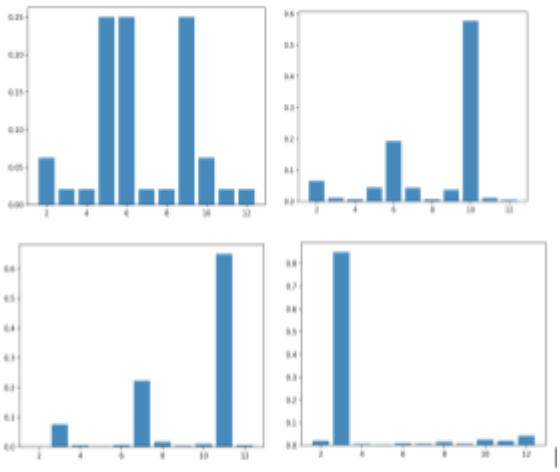

Fig 5 - Simulation of Bayesian Localization in Python

Condition 2 ensures that we dont stop at an office that we have identified as either s1, s2, s3 without being confident in it and hence prevent wrong stops. We chose the probability of 80% by looking at the probability distribution acquired from the python code from part 1 of the assignment fig (5). We saw that this probability was achieved after a few state updates and hence could be acquired before the end of the second traversal. We also tested out different probabilities and initially would stop at the incorrect offices, however, that was not an error from our prediction model, rather from stopping despite having low confidence.

Demonstration Performance

Upon demonstrating the lab, the robot was required to start at office 12 and stop at offices 9, 8, and 3. Due to lower confidence levels, the robot surpassed office 3 as the state estimation had yet to converge. The robot stopped at offices 9 and 8 in the first round as they were positioned towards the end of the traversal and stopped at office 3 in the second round. The state prediction converged by the time the robot had traversed the first half of the map.

List of potential improvement

To increase the efficiency of the robot, severals improvements could be made :

- We could change the formula to get the measurement model. The weight of RGB pixels could be different to make more difference between the colour of the line and blue/green nodes.
- We could change the colour of some nodes to spread the range of RGB values to be less likely to get a wrong colour from the camera.
- We could change the state prediction at the beginning if the robot starts at a known node (or known area) on the map. So the state prediction could converge faster and the robot can know its position faster on the map (especially on a bigger and more complex map).
- We were able to achieve the desired performance by decreasing the speed to 0.05 this caused the robot to traverse the path in a long time which made the process very inefficient. By tuning PID to perform well at higher speeds and colour detection to be done faster (higher connection speed), we can increase the speed of the robot and perform delivery tasks at a faster time.

Conclusion

For this project, we created a PID controller, we found that getting the values for each gain was a difficult task that needed a lot of trial and error to make a robot follow a line. This was even harder to tune because any small error at the beginning of a node was amplified at the end of the node and because we had a small length of line between each node. Now we can create a better PID much faster than before our project. Also we created a bayesian localizer during this project, in part 5.1 we found many different ways to implement this method and each one of them had pros and cons. We chose to use matrix in measurement model and state model because it seemed to be the most straightforward way to do it because we used vectors but it isn’t the most intuitive way to do it especially for the measurement model.

Appendix

Bayes.py

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

Px=np.array([1/11,1/11,1/11,1/11,1/11,1/11,1/11,1/11,1/11,1/11,1/11])



model=np.zeros(shape=(3,11,11))
for i in range (11):
   model[0][i][i-1]=0.85
   model[0][i][i]=0.1

   model[1][i][i-1]=0.05
   model[1][i][i]=0.9

   model[2][i][i-1]=0.05
   model[2][i][i]=0.1
   if i==10:
       model[0][i][0]=0.05
       model[1][i][0]=0.05
       model[2][i][0]=0.85
   else:
       model[2][i][i+1]=0.85
       model[1][i][i+1]=0.05
       model[0][i][i+1]=0.05

print(model)

b=np.array([0.05,0.2,0.6,0.05,0.05,0.2,0.6,0.05,0.05,0.2,0.6])
g=np.array([0.05,0.6,0.2,0.05,0.05,0.6,0.2,0.05,0.05,0.6,0.2])
y=np.array([0.65,0.05,0.05,0.2,0.2,0.05,0.05,0.2,0.65,0.05,0.05])
o=np.array([0.15,0.05,0.05,0.6,0.6,0.05,0.05,0.6,0.15,0.05,0.05])

x= [2,3,4,5,6,7,8,9,10,11,12]

Px=np.matmul(model[0],Px)

Px=np.matmul(model[0],Px)
Px=np.multiply(o,Px)
Px=Px/(sum(Px))
print(Px)
```

```
plt.bar(x,Px)
plt.show()

Px=np.matmul(model[0],Px)
Px=np.multiply(y,Px)
Px=Px/(sum(Px))
print(Px)
plt.bar(x,Px)
plt.show()

Px=np.matmul(model[0],Px)
Px=np.multiply(g,Px)
Px=Px/(sum(Px))
print(Px)
plt.bar(x,Px)
plt.show()

Px=np.matmul(model[0],Px)
Px=np.multiply(b,Px)
Px=Px/(sum(Px))
print(Px)
plt.bar(x,Px)
plt.show()

Px=np.matmul(model[0],Px)
Px=Px/(sum(Px))
print(Px)
plt.bar(x,Px)
plt.show()

Px=np.matmul(model[0],Px)
Px=np.multiply(g,Px)
Px=Px/(sum(Px))
print(Px)
plt.bar(x,Px)
plt.show()

Px=np.matmul(model[0],Px)
Px=np.multiply(b,Px)
Px=Px/(sum(Px))
print(Px)
plt.bar(x,Px)
```

```
plt.show()

Px=np.matmul(model[1],Px)
Px=np.multiply(g,Px)
Px=Px/(sum(Px))
print(Px)
plt.bar(x,Px)
plt.show()

Px=np.matmul(model[0],Px)
Px=np.multiply(o,Px)
Px=Px/(sum(Px))
print(Px)
plt.bar(x,Px)
plt.show()

Px=np.matmul(model[0],Px)
Px=np.multiply(y,Px)
Px=Px/(sum(Px))
print(Px)
plt.bar(x,Px)
plt.show()

Px=np.matmul(model[0],Px)
Px=np.multiply(g,Px)
Px=Px/(sum(Px))
print(Px)
plt.bar(x,Px)
plt.show()


Px=np.multiply(b,Px)
Px=Px/(sum(Px))
print(Px)
plt.bar(x,Px)
plt.show()
```

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


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.line_index=0
       self.num_states = len(p0)
       self.colour_codes = colour_codes
       self.colour_map = colour_map
       self.probability = p0
       self.state_prediction = np.array([1/11]*11)

       self.cur_colour = np.array([0,0,0])  # most recent measured colour
       self.model=np.zeros(shape=(len(p0),len(p0)))
       self.measure=np.zeros(len(colour_codes))
       self.traverse = [6]
       self.prev_col = None

   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]
       colour_list = ["red","green","blue","yellow","line","None"]
       #change to hsv
 h_cur, s_cur, v_cur =
colorsys.rgb_to_hsv(self.cur_colour[0]/255,self.cur_colour[1]/255,self.cur_colour[2]/2
55)
       #put in array
```

```
        self.cur_colour = np.array([h_cur, s_cur,v_cur])
        min = 1000
        val = 0
        for i in range(5):
           diff =  np.linalg.norm(colour_codes[i]-self.cur_colour)
           if diff<min:
            min = diff
            val = i
        # loop ended we can now detect what colour were on
        col_meas = colour_list[val]
        self.traverse.append(col_meas)
        self.prev_col = self.traverse[0] #initially index 6
        self.traverse.pop(0) # [6,new] ->> [new] - next round ->>
[new,newnew]->>[newnew]

        if col_meas != self.prev_col and self.prev_col == 'line':
            print(self.state_prediction)
            self.state_predict()
            #we update the state based on the colour we measured
            self.state_update(col_meas)
            max_prob = np.max(self.state_prediction)
            #argmax returns indeces of max element in array. self.state_pred is
probability. our path starts at 2 so every index (from 0) is added two to it
            max_prob_ind = np.argmax(self.state_prediction)+2
            print(max_prob_ind)
            print(" ")

            if int(max_prob_ind) in [2,8,3] and max_prob>0.6: #checking to see if the
value of the max index is of the 3 stops we want to pause at
                rospy.sleep(2)
        self.prev_col = col_meas
        print(col_meas)#print detected colour

        #print(self.cur_colour)

    def line_callback(self, msg):

        self.line_index= msg.data
        error=self.line_index-320
        d=(error-Xk)
        Xk=error
        inte+=error
```

```
    correction=-error*0.003 - d*0.007 - inte*0.0001
    twist.linear.x=0.05
    c=0
    if self.prev_col != 'line':
        twist.angular.z=0
        inte=0
        self.cmd_pub.publish(self.twist)

    else:
        twist.angular.z=correction
        self.cmd_pub.publish(self.twist)
```

```
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):
    p0 = np.ones_like(colour_map) / len(colour_map)
    model=np.zeros(shape=(3,len(p0),len(p0)))
    for i in range (len(p0)):
        model[0][i][i-1]=1
        model[1][i][i]=1

        if i==len(p0)-1:
            model[2][i][0]=1
        else:
            model[2][i][i+1]=1
    if u<0:
        a=0
    elif u>0:
        a=2
    else:
        a=1
    self.model=model[a]
```

```
    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).
        """
        s=0
        for i in range(len(colour_codes)):
            prob[i]=np.linalg.norm(x-colour_codes[i])
            s+=prob[i]
        prob=prob/s
        self.measure=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)
        """



        self.state_prediction=np.matmul(self.model,self.state_prediction)

    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 R G B Y
        """
```

```
M=[[0,0,0,1,0],[0,1,0,0,0],[0,0,1,0,0],[1,0,0,0,0],[1,0,0,0,0],[0,1,0,0,0],[0,0,1,0,0]
,[1,0,0,0,0],[0,0,0,1,0],[0,1,0,0,0],[0,0,1,0,0]]

self.state_prediction=np.multiply(np.matmul(M,self.measure),self.state_prediction)  #to
solve dimension problems
       self.state_prediction=self.state_prediction/(sum(self.state_prediction))



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]

   # 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 = np.array([
       [235, 73, 129],  # red
       [157, 181, 164],  # green
       [179, 166, 184],  # blue
       [176, 162, 153],  # yellow
       [172, 161, 167],  # line
   ])
    colour_codes = np.array([
   [92, 73, 89],  # red
   [26, 10, 66],  # green
   [80, 13, 69],  # blue
   [9, 10, 62],  # yellow
   [89, 11, 63],  # line
   ])


   # 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)

    twist=Twist()
    c=-100

    inte=0
    d=0
    Xk=0
    print(np.linalg.norm(colour_codes[0]-colour_codes[4]))
    print(np.linalg.norm(colour_codes[1]-colour_codes[4]))
    print(np.linalg.norm(colour_codes[2]-colour_codes[4]))
    print(np.linalg.norm(colour_codes[3]-colour_codes[4]))

    for i in range(300):
        """
        TODO: complete this main loop by calling functions from BayesLoc, and
        adding your own high level and low level planning + control logic
        """









        '''
                        if   np.linalg.norm(localizer.cur_colour-colour_codes[0])<50   or
np.linalg.norm(localizer.cur_colour-colour_codes[1])<12                                or
np.linalg.norm(localizer.cur_colour-colour_codes[2])<10                                or
np.linalg.norm(localizer.cur_colour-colour_codes[3])<7 and c==10:
            if c==10:
                twist.angular.z=0
                inte=0
                print('color')
                localizer.state_model(0.075)
                localizer.measurement_model(localizer.cur_colour)
                localizer.state_predict()
                localizer.state_update()
                c+=1
```

```
                """ to make it stop at K"""
                if localizer.state_prediction[k]>0.6:
                    rospy.sleep(2)
            else:
                twist.angular.z=0
                inte=0
                print('color')
                c+=1
        else:
            twist.angular.z=correction/1.5
            print('line')
            c=0

        """ to make it stop at K"""
        if localizer.state_prediction[k]>0.6:

        '''
        localizer.cmd_pub.publish(twist)
        rate.sleep()

    rospy.loginfo("finished!")
    rospy.loginfo(localizer.probability)
]\
```