Ham and Eggs bot

Individual Report

Robotics Group Project

Zihao You (K1923149)

April 2021

Introduction

Throughout this year, our team has managed to stay together for both terms as well as complete all the tasks given. However, because of living close enough to each other in London, our group has got advantages for meeting in person every time without the use of Microsoft Teams. This means that during the actual implementation stage, we haven't really divided and allocated different tasks to every team member. Instead, we are all sitting and working together. To be specific, me and Henry both came up with ideas, inspected the code and sometimes offered help when Michael was doing the programming. This is because our code contains a majority of Python and Michael has got the longest experience of that. Additionally, I have been actively connecting to all TAs by raising up questions that the other group members won't be able to solve. And in the next few sections I will walk through all of these in detail.

Contribution

Term 1

Before we started meeting in person, I ensured that I had successfully set up all the environment needed so that I was able to settle down quickly when it came to teamwork. For instance, installing Ubuntu operating system, setting up ROS and Gazebo, as well as integrating the light sensor in my machine. During the implementation, we worked together by creating GitLab platform and all the code for this term went to exactly one file. Although all Python code was made by Michael, it didn't mean that me and Henry were sitting there as audiences. Instead, I have tried to understand every line of code Michael has written and finally wrote the comments for the entire file (see appendix). Now let me quickly walk you through the code our group has implemented:

- First of all, instead of computing the distance between the desired light sensor value and the actual light sensor value. We checked if the sensor readings are below a threshold and from all those below the threshold, we checked which of those has the smallest value, then adjusted it in order to incrementally update the desired position of the light sensor. This refers to Method 2 of the Gazebo implementation of PID control, which pseudocode is given on the lecture slide and we had decided to pick that method but instead with 16 light sensors.

- The whole program consisted of four most important functions plus a class for defining PID control. The first one controls the robot's movement. The second one handles the light sensor scanning. Inside this function, we had defined a parameter "a" which corresponds to the stages during the obstacle avoidance process. To be specific, "a == 1" means moving backwards, if the robot detects an obstacle in front and reaches a minimum distance of 0.35. Next, the robot starts turning left in front of the obstacle at stage two (i.e. a == 2). Then at the following two stages, the robot is circumnavigating the obstacle. And the robot will move back onto the track at stage five. Additionally, the third and the fourth function has something to do with the robot's twisting, as well as the main function.

- For line following and obstacle avoidance, we used both PID control. However, the PID control used in obstacle avoidance is more like a staged approach. To be specific, the robot turns 90 degrees to it as long as it sees an object. But in real life, the robot has to also decide the shortest path (i.e. turning left or right) depending on the shape of the object. Also during the turning process, the robot always maintains a constant distance from the obstacle regardless of its shape.

- In order to compute the distance between the light sensor and the obstacle, we used a Lidar sensor, which looks like a cone at the front of the robot. Once that reaches it will hit a minimum distance of 0.35, but due to slight acceleration it will instead give a reading of roughly 0.3, which is the distance we are trying to maintain from the obstacle every time.

- And as we are using a staged approach here, the line is turned off in stage one and when it resumes, the light sensor will hit another reading and therefore the robot is going to be in the final stage.
- For obtaining the robot's angular and linear velocity, the methods are pretty much the same. Basically, we assumed that the robot is moving at a constant velocity throughout the entire motion, and this value will decrease if the turning angle becomes greater. After that is just a simple math calculation, by multiplying the turning angle and then subtracting this part from our velocity, which is the minimum one in this case and it should be proportional to the turning angle.

I had also created a Minutes document and recorded it for every task, there are two of them which belonged to term 1. This means that for every single task we had a separate deadline, in order for us to be able to meet the final project deadline by avoiding last-minute rush.

Furthermore, I had helped the team members figure out a task completion problem by messaging a TA on Microsoft Teams. To be specific, we were all initially unsure about whether we were supposed to record two laps without an obstacle in our video submission. After getting back from a TA, we were finally clear. It was enough to just include one lap for line following and another one for obstacle avoidance.

Term 2

Since we all got advantaged by living close together as well as enjoyed meeting in person, we had decided to keep the same group in the second semester. On the very first meeting of this term, we did the Open Loop Localisation Error Analysis that had been assigned in the lecture together in order for us to get more familiarised with the upcoming tasks for the major project. Again, all the implementation had been done by Michael, Henry gave assistance to him at the same time, and my job was to create an Excel spreadsheet, record all the readings from Gazebo, as well as perform relevant calculations (see (Figure 1) to (Figure 4) in appendix):

- In the first task, the robot moved forwards, and we took readings every 20mm. According to (Figure 1), we had set the time interval to 4s and the velocity to 0.3. The y-coordinate value remained almost constant, meaning that our robot won't wobble too much (i.e. the robot had followed almost a straight line). After repeating 10 times, we got an average Cartesian position error of roughly 1.209, which is pretty close to the expected error (i.e. v*t) of 1.2, since the percentage difference is less than 5%.
- In the second task (Figure 2), the robot moved backwards. Again, we kept the exact same distance and time interval as in Task 1, but instead set the velocity of the robot with equal magnitude but opposite sign. The y-coordinate value don't change much here as well. We also got pretty accurate readings in this task, as the percentage difference is again lower than 5%. To be specific, the average Cartesian position error is around -1.186, in comparison with the expected error of -1.2.
- However, when it came to the rectangular (Figure 3) and diagonal motions (Figure 4), the readings we got were not quite right, with an -83% in Task 3 and -17% in Task 4. They are both far greater than $\pm 5\%$ so I don't think we took the readings very carefully in these two tasks. Additionally, as the robot was moving in both x and y directions here, the method used for computing Cartesian position error would be slightly different. Instead of directly subtracting the starting position from the final position in the x-plane, I have used a mathematical formula that I have learned in high school for calculating the straight line distance between two coordinates, which is $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$.

After finishing Open Loop Localisation Error Analysis, we immediately moved onto the first task of the major group project, which is Bayesian Localisation. We had managed to get this done before the reading week and this task was again implemented within one single file and I wrote all the comments for the code as well, here is a quick overview of what we did for this task:

- There are 19 different functions for this program, which is more than the tasks in the first semester as the difficulty progresses. But we still managed to let all functions do different things, which follows the Responsibility-driven design technique in object-oriented programming.

- In our example, we assumed that there is 90% of chance that sensor will give the correct reading, while 10% for sensor provides incorrect readings. The robot's speed is set to be 0.05, and its turning speed to be pi/16. Also it's a 24-grid line here, so they are labelled from index 0 up to and including 23.
- The robot's moving left or right are monitored by a for-loop in python. To be specific, for the left motion it's looped from the beginning up to but not includes -1, whereas for moving right it's looped from 1(exclusive) up till the end. Basically it's a special case here, meaning that there is no chance of the robot moving onto the rightmost square if it tries to move left, and the situation gets reversed when it tries to move right.
- And for the rest part of this program, the probability of robot in a specific location is updated by the probability of the robot in the current location plus the probability of the robot in the next location (i.e. one square to the right) if it's trying to move left, and vice versa for moving right. We had used the exact same formula as in the lecture slide for the "Recursive Bayesian Filter in 1D" case.

In the meantime, our group had also conducted an Bayesian Localisation experiment by taking three readings, which are displayed as (Figure 5) to (Figure 9) in appendix:

- (Figure 5) shows what the first reading will look like after applying Bayesian Localisation, the robot's light sensor at this moment detects the black grid and it's heading left. The histogram on (Figure 6) shows the belief distribution between 0 and 23, which matches the index of the grid. The height of each bar indicates the probabilities of which colour light sensor can detect at each location. The last bar (i.e. the one at index 25) displays the sum of the probability, which must always be 1, by definition. In this reading, the beliefs are distributed quite uniformly.
- After a few moves, the robot is still going left, and (Figure 7) shows that some of the beliefs start becoming significant.
- Finally, the light sensor detects the white grid (Figure 8), and if you count the grid index carefully, you will notice that the location at which the light sensor detects is location 7, which in histogram (Figure 9) the belief peaks at the exact same location of roughly 0.9. I think this is what we're aiming for as the probability that the light sensor reads correctly is also 0.9.

During this term, I had interacted with TAs more often by regularly messaging them on Microsoft Teams when Henry and Michael had questions, especially during the A* planning implementation stage, since we all believed that it was the most challenging task in this term. For instance, I asked them if it is suitable to use Odom topic for error checking and our A* coordinate, are we allowed to adjust the actual cost (i.e. g(n)) after defining it by ourselves, and so on. And some questions regarding exam maps as well, by asking them what exactly the thin black bar placed between two cylinders is, as well as if it is allowed to modify the height of barriers manually for Lidar sensor scanning. The queries were all cleared after getting back from TAs.

As the tasks on Term 1 and the Bayesian Localisation task on Term 2 were my main focus, I also successfully delivered the presentation on these two parts on 30th March.

Last but not least, with strong attention to details, I did the proof-reading job for the Team Management report, by finding out more than 10 spelling and grammatical mistakes, both on Gantt chart and the main text, as well as giving suggestions of replacing some words for the sake of better description.

Conclusion & Improvement

Generally speaking, I really enjoyed this module, as well as worked with two intelligent teammates throughout this year. I have learned in every group meeting much more than during lectures. We had successfully managed to complete all the tasks in both terms. When we were actually doing the tasks, we firstly modelled and came up with different types of algorithms, then attached it to the robot in ROS if we were all happy with this particular method, plus the strong time management skills, I think this is pretty much our key to success.

However, due to the time constraint, we had still got some slight issues in our robot, such as being too strongly opposed from the obstacle, twisting randomly at sometime, the movements are not always smooth, like slow moving occasionally, and so on. From my perspective, all of these can be optimised accordingly if we had more time. And if we are lucky enough to undertake another project together in the future, I will urge every group member to finish all the implementations as early as possible and spend more time doing testing and debugging, from a single line statement all the way to the whole program. By preserving this, our code will potentially close to bug free.

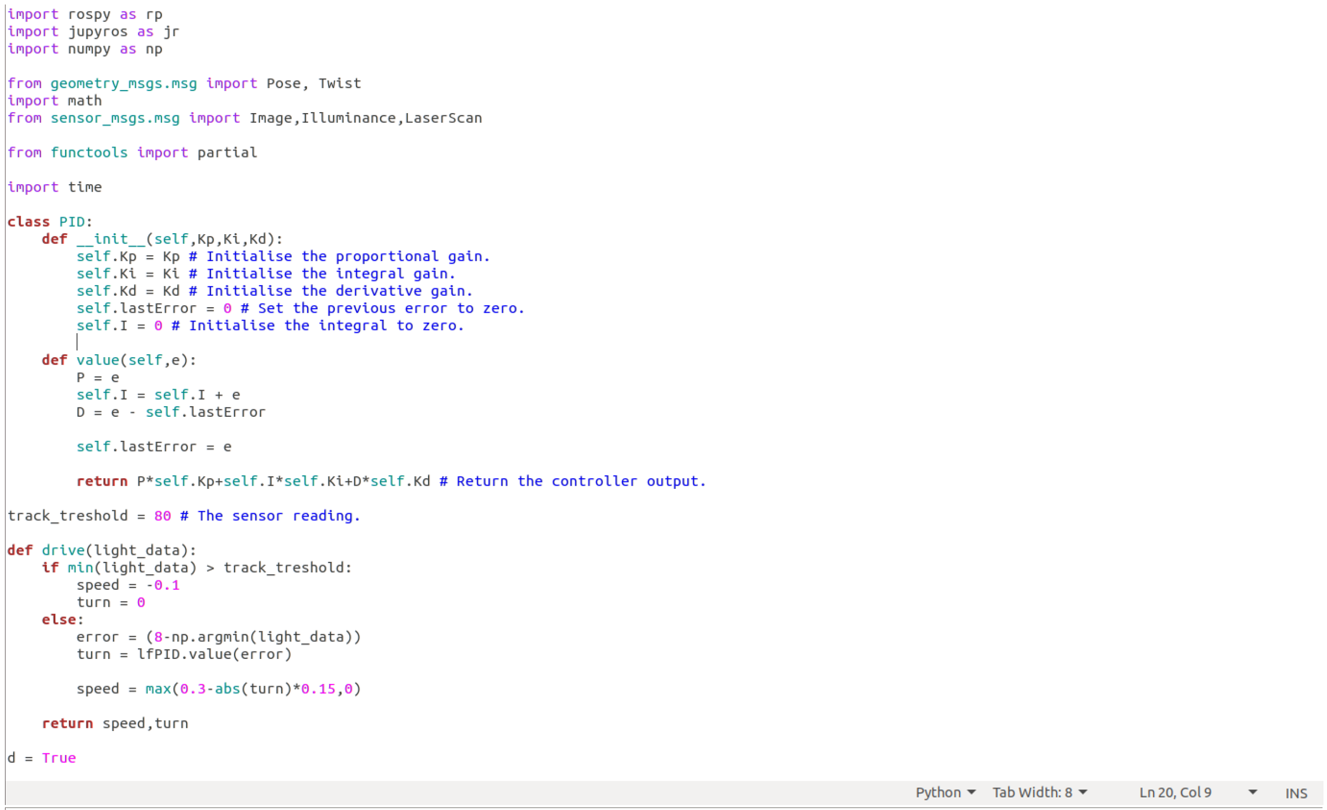
Reference

Some information in this report has been reused from our group presentation on 30th March:

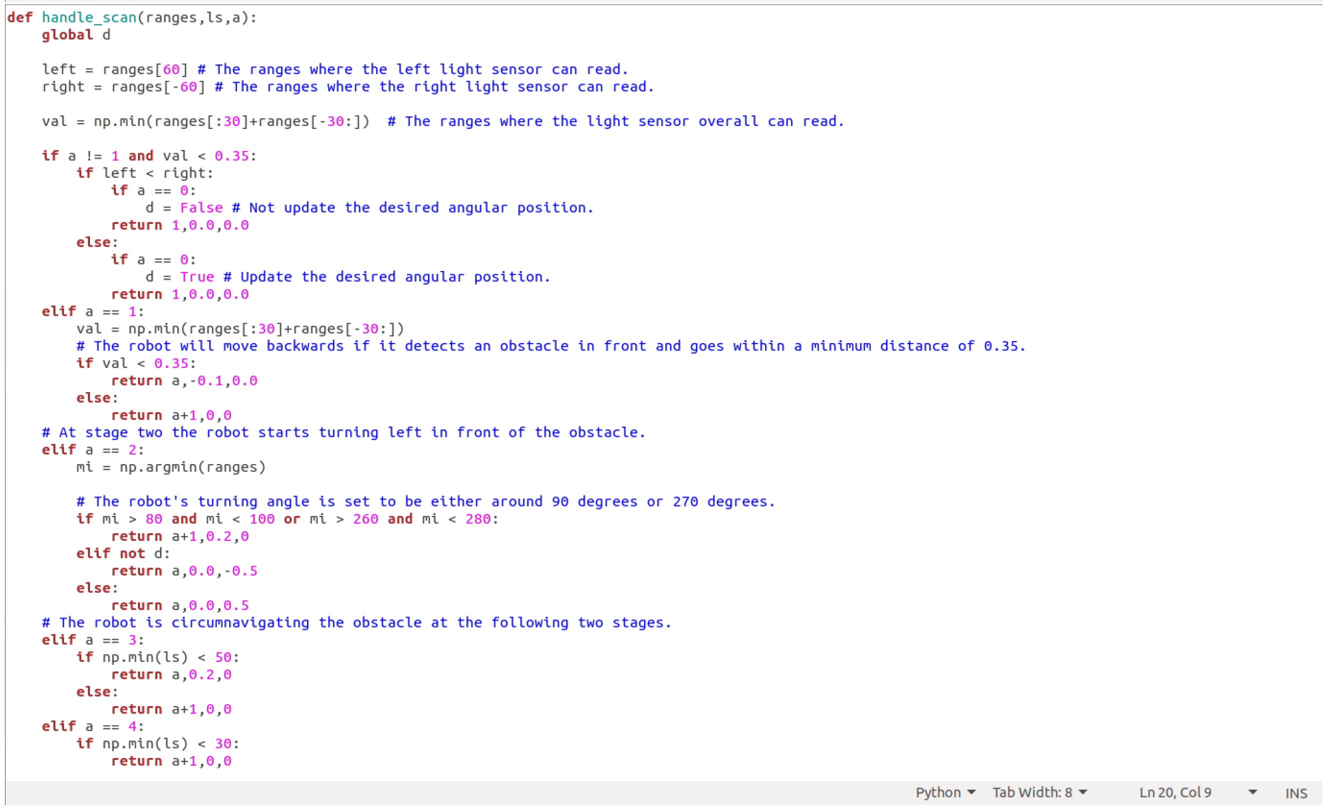
https://web.microsoftstream.com/video/daf695a1-71aa-45e9-a605-1e6c650a68e8

Appendix

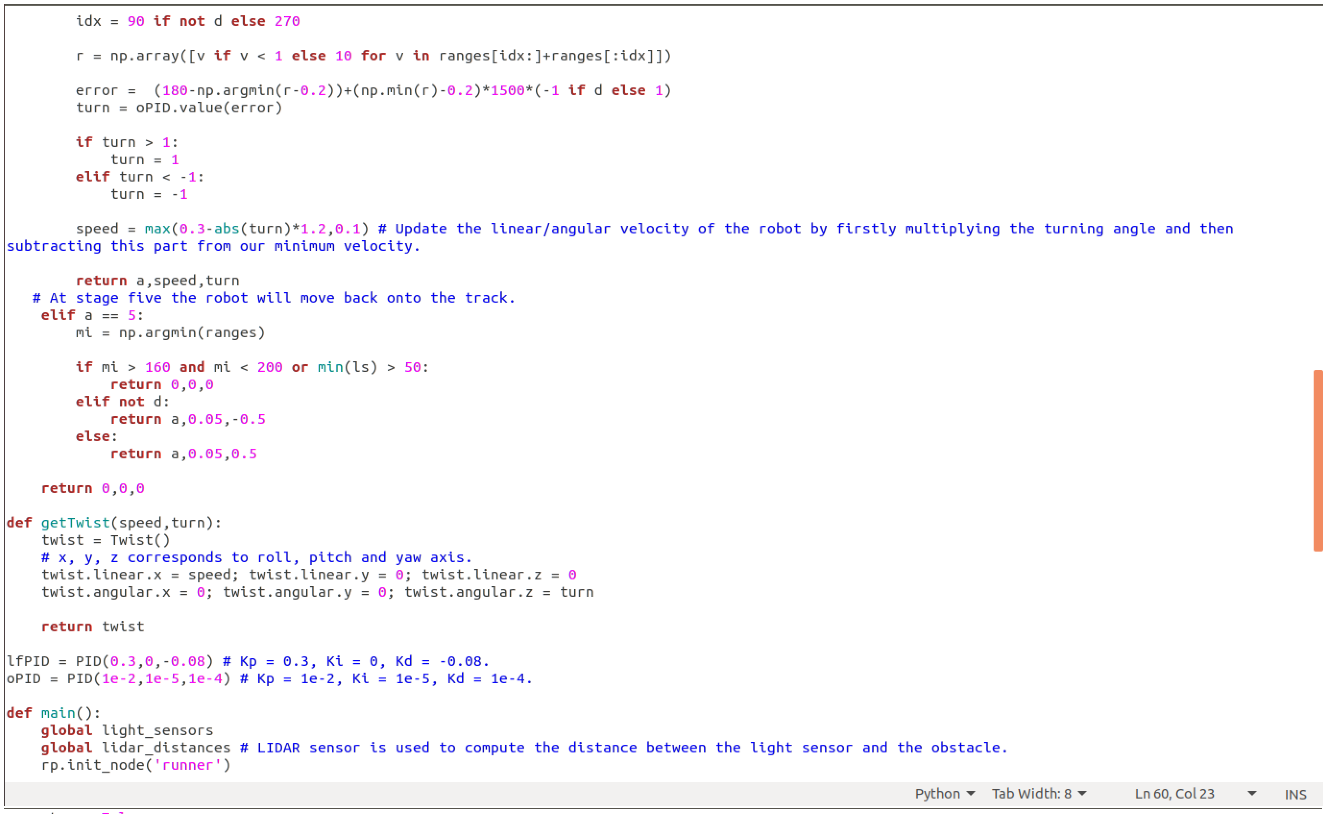
1. Term 1



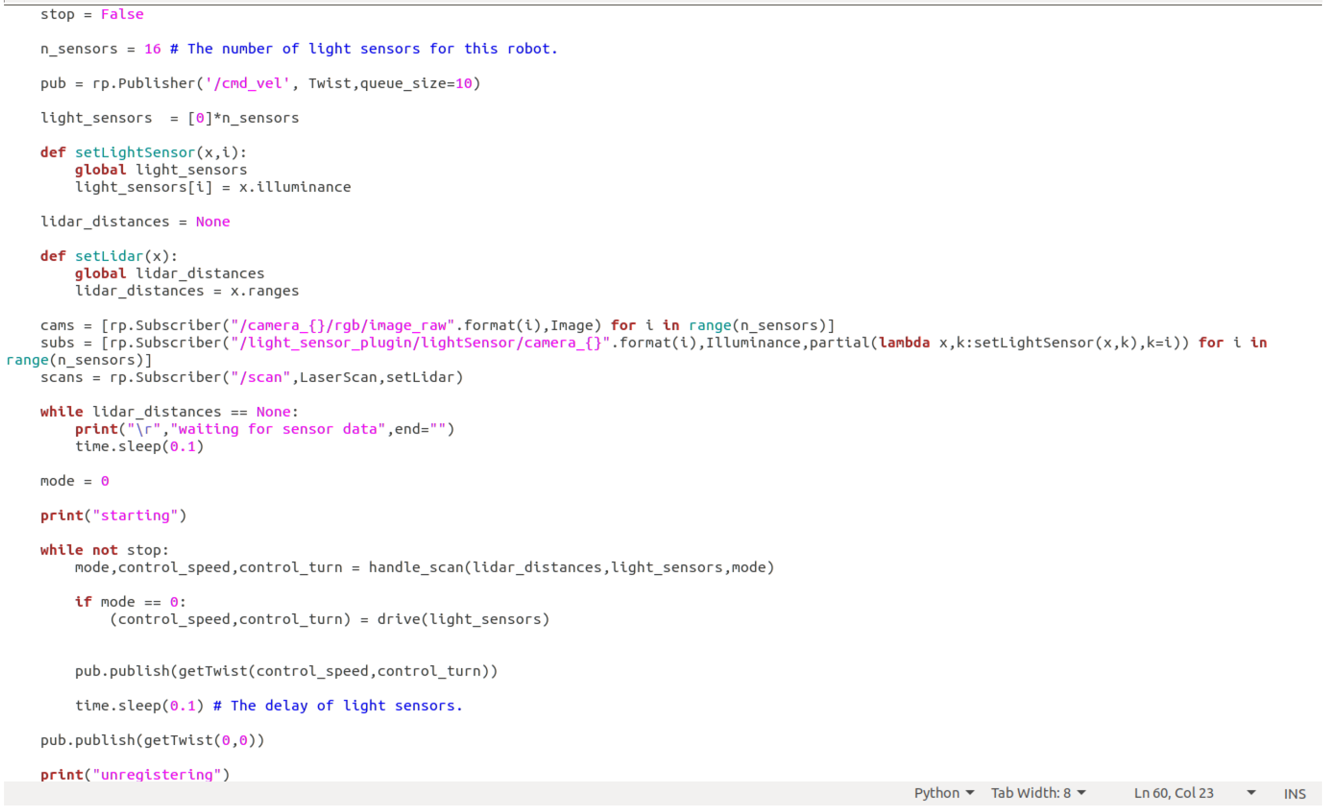
```python
import rospy as rp
import jupyros as jr
import numpy as np

from geometry_msgs.msg import Pose, Twist
import math
from sensor_msgs.msg import Image,Illuminance,LaserScan

from functools import partial

import time

class PID:
    def __init__(self,Kp,Ki,Kd):
        self.Kp = Kp # Initialise the proportional gain.
        self.Ki = Ki # Initialise the integral gain.
        self.Kd = Kd # Initialise the derivative gain.
        self.lastError = 0 # Set the previous error to zero.
        self.I = 0 # Initialise the integral to zero.
        
    def value(self,e):
        P = e
        self.I = self.I + e
        D = e - self.lastError

        self.lastError = e

        return P*self.Kp+self.I*self.Ki+D*self.Kd # Return the controller output.

track_treshold = 80 # The sensor reading.

def drive(light_data):
    if min(light_data) > track_treshold:
        speed = -0.1
        turn = 0
    else:
        error = (8-np.argmin(light_data))
        turn = lfPID.value(error)

        speed = max(0.3-abs(turn)*0.15,0)

    return speed,turn

d = True
```



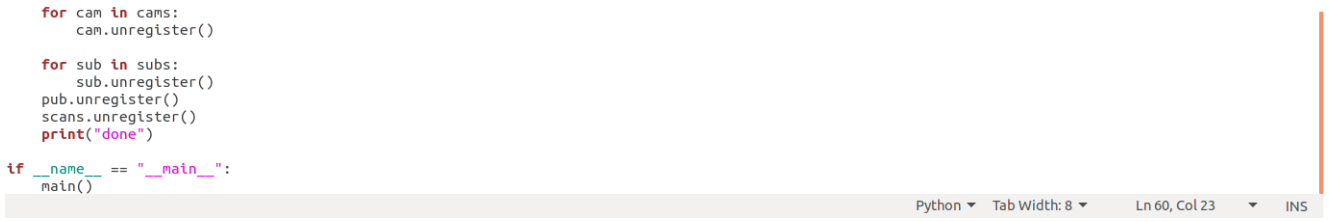
```python
def handle_scan(ranges,ls,a):
    global d

    left = ranges[60] # The ranges where the left light sensor can read.
    right = ranges[-60] # The ranges where the right light sensor can read.

    val = np.min(ranges[:30]+ranges[-30:])  # The ranges where the light sensor overall can read.

    if a != 1 and val < 0.35:
        if left < right:
            if a == 0:
                d = False # Not update the desired angular position.
            return 1,0.0,0.0
        else:
            if a == 0:
                d = True # Update the desired angular position.
            return 1,0.0,0.0
    elif a == 1:
        val = np.min(ranges[:30]+ranges[-30:])
        # The robot will move backwards if it detects an obstacle in front and goes within a minimum distance of 0.35.
        if val < 0.35:
            return a,-0.1,0.0
        else:
            return a+1,0,0
    # At stage two the robot starts turning left in front of the obstacle.
    elif a == 2:
        mi = np.argmin(ranges)

        # The robot's turning angle is set to be either around 90 degrees or 270 degrees.
        if mi > 80 and mi < 100 or mi > 260 and mi < 280:
            return a+1,0.2,0
        elif not d:
            return a,0.0,-0.5
        else:
            return a,0.0,0.5
    # The robot is circumnavigating the obstacle at the following two stages.
    elif a == 3:
        if np.min(ls) < 50:
            return a,0.2,0
        else:
            return a+1,0,0
    elif a == 4:
        if np.min(ls) < 30:
            return a+1,0,0
```



```
        idx = 90 if not d else 270

        r = np.array([v if v < 1 else 10 for v in ranges[idx:]+ranges[:idx]])

        error =  (180-np.argmin(r-0.2))+(np.min(r)-0.2)*1500*(-1 if d else 1)
        turn = oPID.value(error)

        if turn > 1:
            turn = 1
        elif turn < -1:
            turn = -1

        speed = max(0.3-abs(turn)*1.2,0.1) # Update the linear/angular velocity of the robot by firstly multiplying the turning angle and then
subtracting this part from our minimum velocity.

        return a,speed,turn
   # At stage five the robot will move back onto the track.
    elif a == 5:
        mi = np.argmin(ranges)

        if mi > 160 and mi < 200 or min(ls) > 50:
            return 0,0,0
        elif not d:
            return a,0.05,-0.5
        else:
            return a,0.05,0.5

    return 0,0,0

def getTwist(speed,turn):
    twist = Twist()
    # x, y, z corresponds to roll, pitch and yaw axis.
    twist.linear.x = speed; twist.linear.y = 0; twist.linear.z = 0
    twist.angular.x = 0; twist.angular.y = 0; twist.angular.z = turn

    return twist

lfPID = PID(0.3,0,-0.08) # Kp = 0.3, Ki = 0, Kd = -0.08.
oPID = PID(1e-2,1e-5,1e-4) # Kp = 1e-2, Ki = 1e-5, Kd = 1e-4.

def main():
    global light_sensors
    global lidar_distances # LIDAR sensor is used to compute the distance between the light sensor and the obstacle.
    rp.init_node('runner')
```



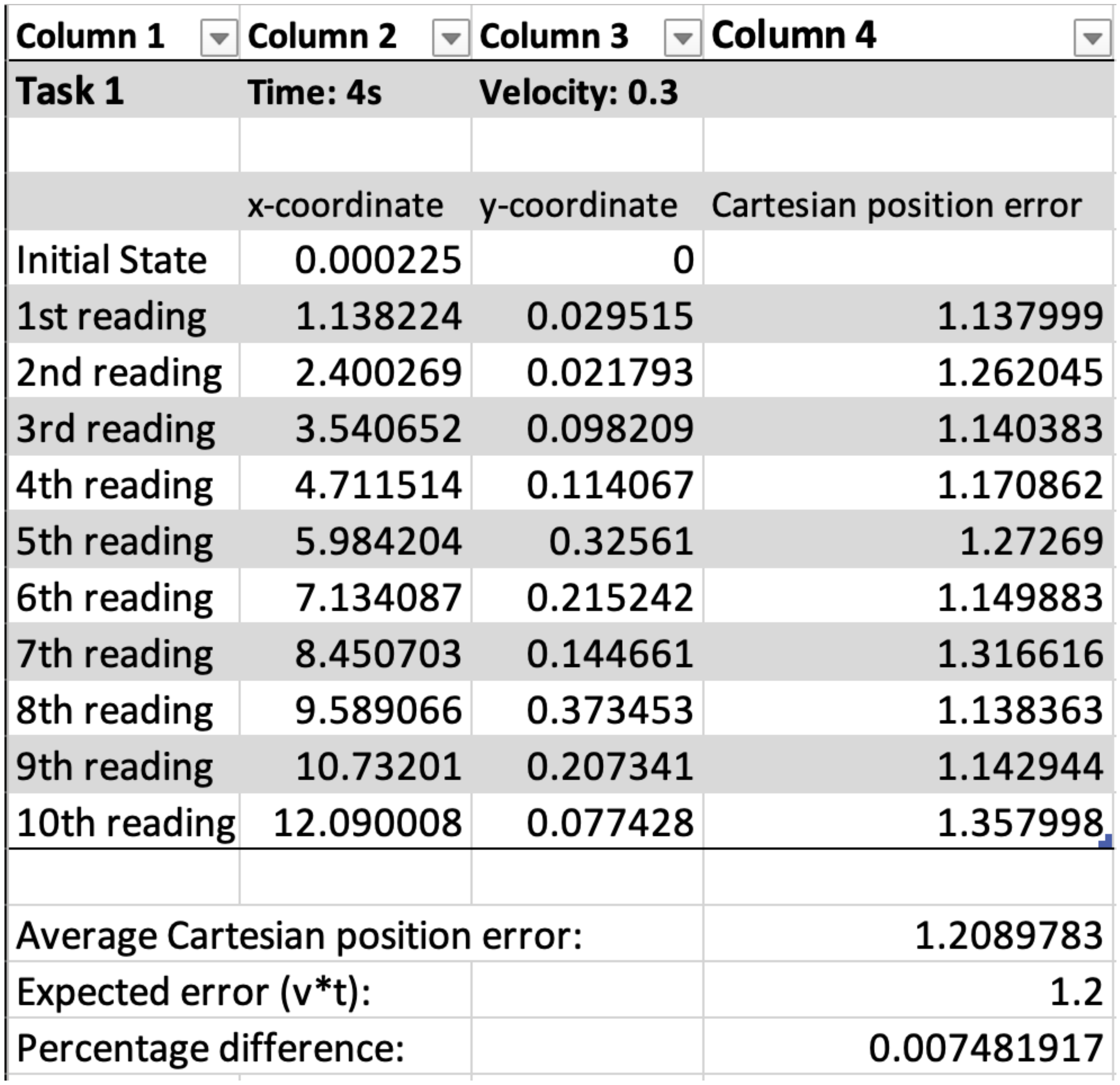
```
    stop = False

    n_sensors = 16 # The number of light sensors for this robot.

    pub = rp.Publisher('/cmd_vel', Twist,queue_size=10)

    light_sensors  = [0]*n_sensors

    def setLightSensor(x,i):
        global light_sensors
        light_sensors[i] = x.illuminance

    lidar_distances = None

    def setLidar(x):
        global lidar_distances
        lidar_distances = x.ranges

    cams = [rp.Subscriber("/camera_{}/rgb/image_raw".format(i),Image) for i in range(n_sensors)]
    subs = [rp.Subscriber("/light_sensor_plugin/lightSensor/camera_{}".format(i),Illuminance,partial(lambda x,k:setLightSensor(x,k),k=i)) for i in
range(n_sensors)]
    scans = rp.Subscriber("/scan",LaserScan,setLidar)

    while lidar_distances == None:
        print("\r","waiting for sensor data",end="")
        time.sleep(0.1)

    mode = 0

    print("starting")

    while not stop:
        mode,control_speed,control_turn = handle_scan(lidar_distances,light_sensors,mode)

        if mode == 0:
            (control_speed,control_turn) = drive(light_sensors)


        pub.publish(getTwist(control_speed,control_turn))

        time.sleep(0.1) # The delay of light sensors.

    pub.publish(getTwist(0,0))

    print("unregistering")
```



```
    for cam in cams:
        cam.unregister()

    for sub in subs:
        sub.unregister()
    pub.unregister()
    scans.unregister()
    print("done")

if __name__ == "__main__":
    main()
```

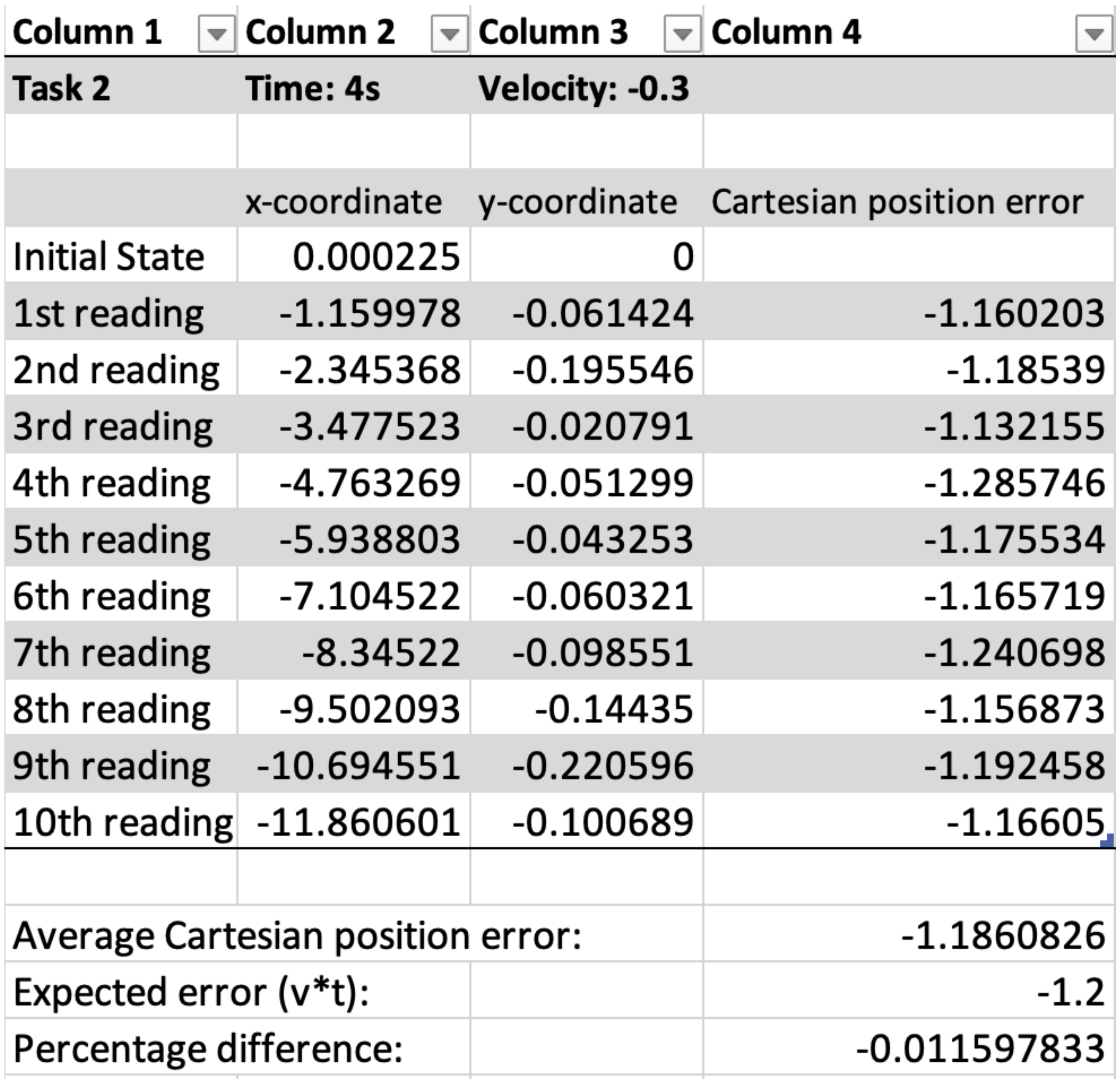
Python ▾ Tab Width: 8 ▾ Ln 60, Col 23 ▾ INS

2. Term 2 - Open Loop Localisation Error Analysis



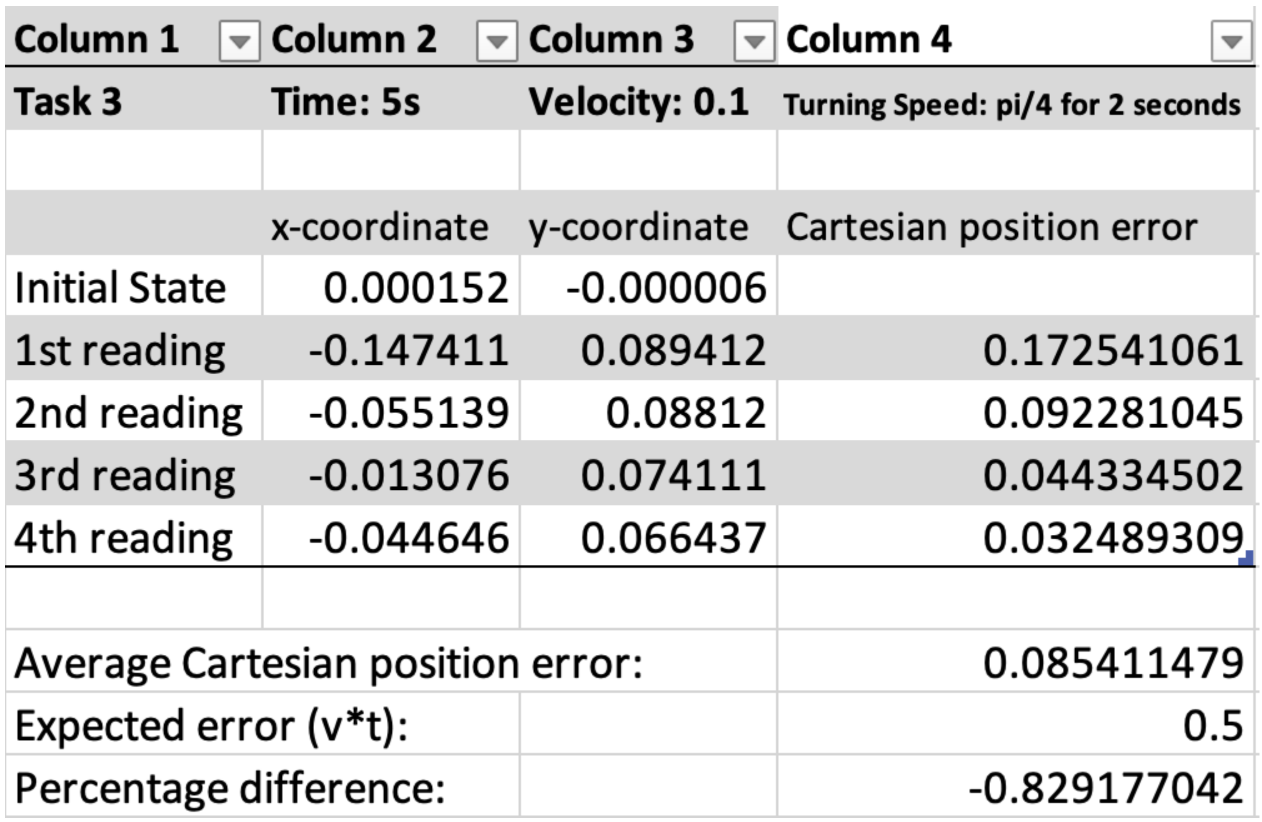
Column 1	Column 2	Column 3	Column 4
Task 1	**Time: 4s**	**Velocity: 0.3**	
	x-coordinate	y-coordinate	Cartesian position error
Initial State	0.000225	0	
1st reading	1.138224	0.029515	1.137999
2nd reading	2.400269	0.021793	1.262045
3rd reading	3.540652	0.098209	1.140383
4th reading	4.711514	0.114067	1.170862
5th reading	5.984204	0.32561	1.27269
6th reading	7.134087	0.215242	1.149883
7th reading	8.450703	0.144661	1.316616
8th reading	9.589066	0.373453	1.138363
9th reading	10.73201	0.207341	1.142944
10th reading	12.090008	0.077428	1.357998
Average Cartesian position error:			1.2089783
Expected error (v*t):			1.2
Percentage difference:			0.007481917

Figure 1:



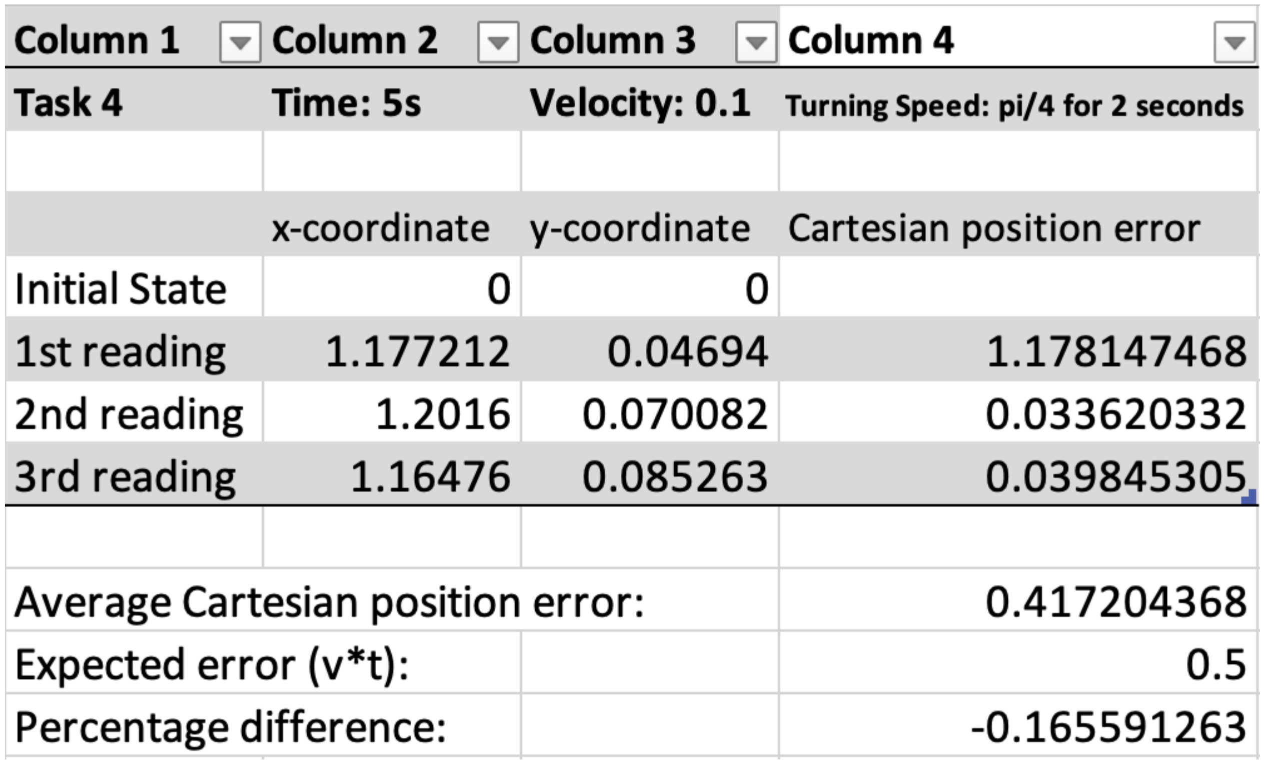
Column 1	Column 2	Column 3	Column 4
Task 2	**Time: 4s**	**Velocity: -0.3**	
	x-coordinate	y-coordinate	Cartesian position error
Initial State	0.000225	0	
1st reading	-1.159978	-0.061424	-1.160203
2nd reading	-2.345368	-0.195546	-1.18539
3rd reading	-3.477523	-0.020791	-1.132155
4th reading	-4.763269	-0.051299	-1.285746
5th reading	-5.938803	-0.043253	-1.175534
6th reading	-7.104522	-0.060321	-1.165719
7th reading	-8.34522	-0.098551	-1.240698
8th reading	-9.502093	-0.14435	-1.156873
9th reading	-10.694551	-0.220596	-1.192458
10th reading	-11.860601	-0.100689	-1.16605
Average Cartesian position error:			-1.1860826
Expected error (v*t):			-1.2
Percentage difference:			-0.011597833

Figure 2:



Column 1	Column 2	Column 3	Column 4
Task 3	**Time: 5s**	**Velocity: 0.1**	**Turning Speed: pi/4 for 2 seconds**
	x-coordinate	y-coordinate	Cartesian position error
Initial State	0.000152	-0.000006	
1st reading	-0.147411	0.089412	0.172541061
2nd reading	-0.055139	0.08812	0.092281045
3rd reading	-0.013076	0.074111	0.044334502
4th reading	-0.044646	0.066437	0.032489309
Average Cartesian position error:			0.085411479
Expected error (v*t):			0.5
Percentage difference:			-0.829177042

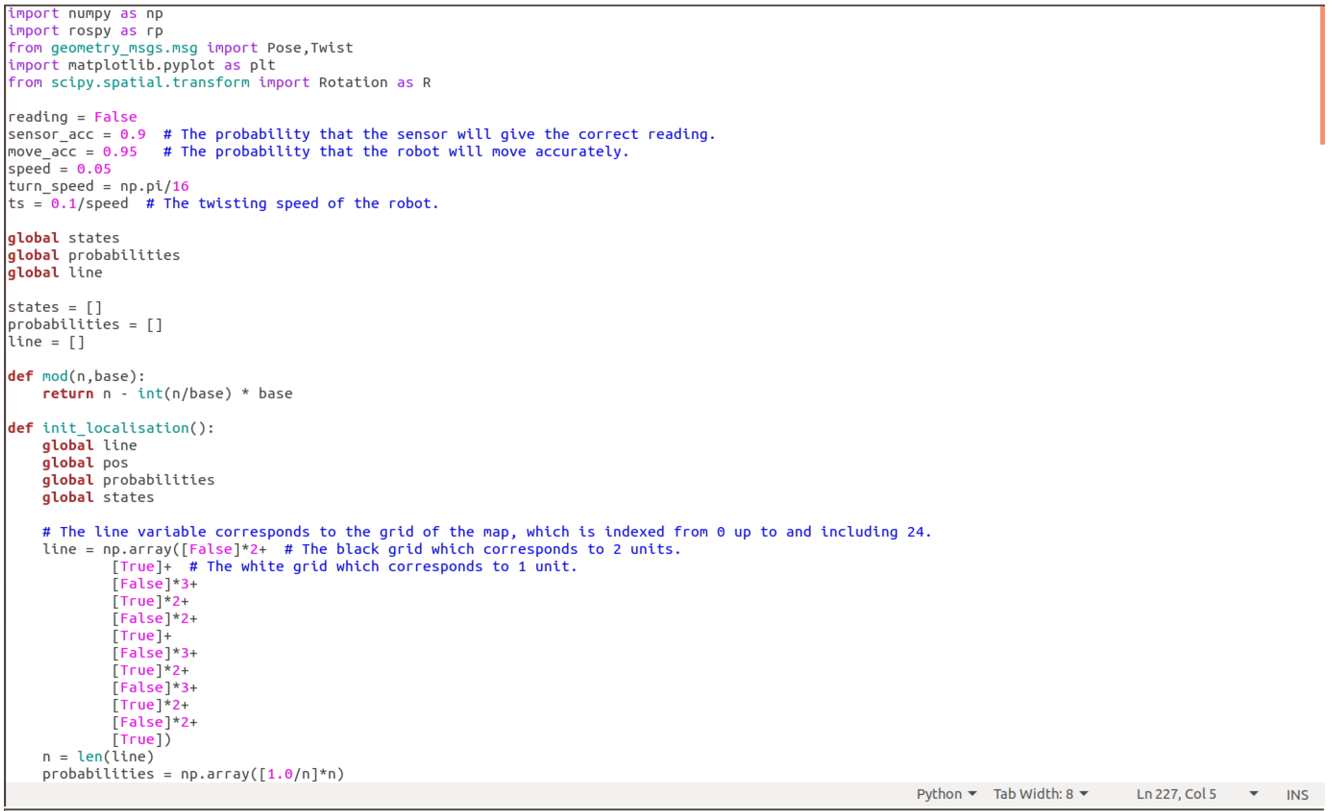
Figure 3:



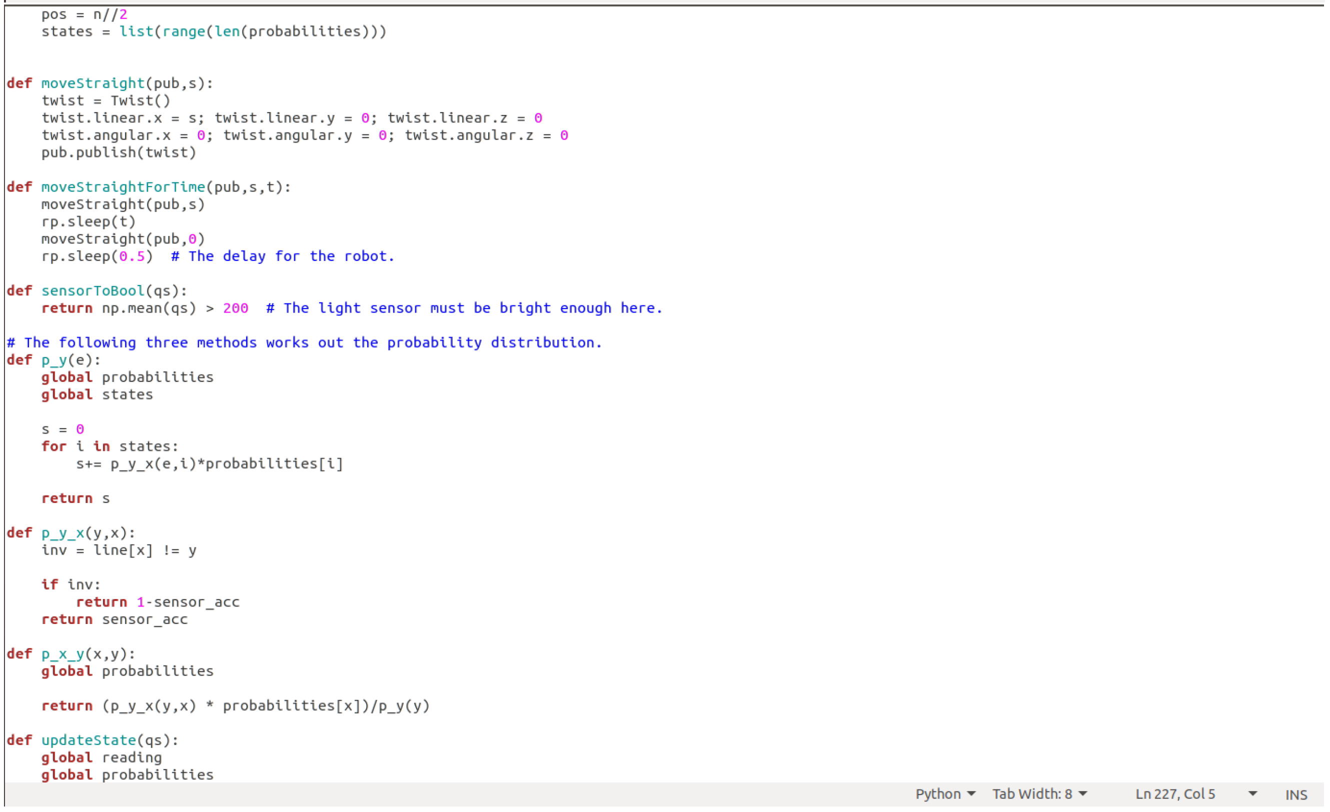
Column 1	Column 2	Column 3	Column 4
Task 4	**Time: 5s**	**Velocity: 0.1**	**Turning Speed: pi/4 for 2 seconds**
	x-coordinate	y-coordinate	Cartesian position error
Initial State	0	0	
1st reading	1.177212	0.04694	1.178147468
2nd reading	1.2016	0.070082	0.033620332
3rd reading	1.16476	0.085263	0.039845305
Average Cartesian position error:			0.417204368
Expected error (v*t):			0.5
Percentage difference:			-0.165591263

Figure 4:

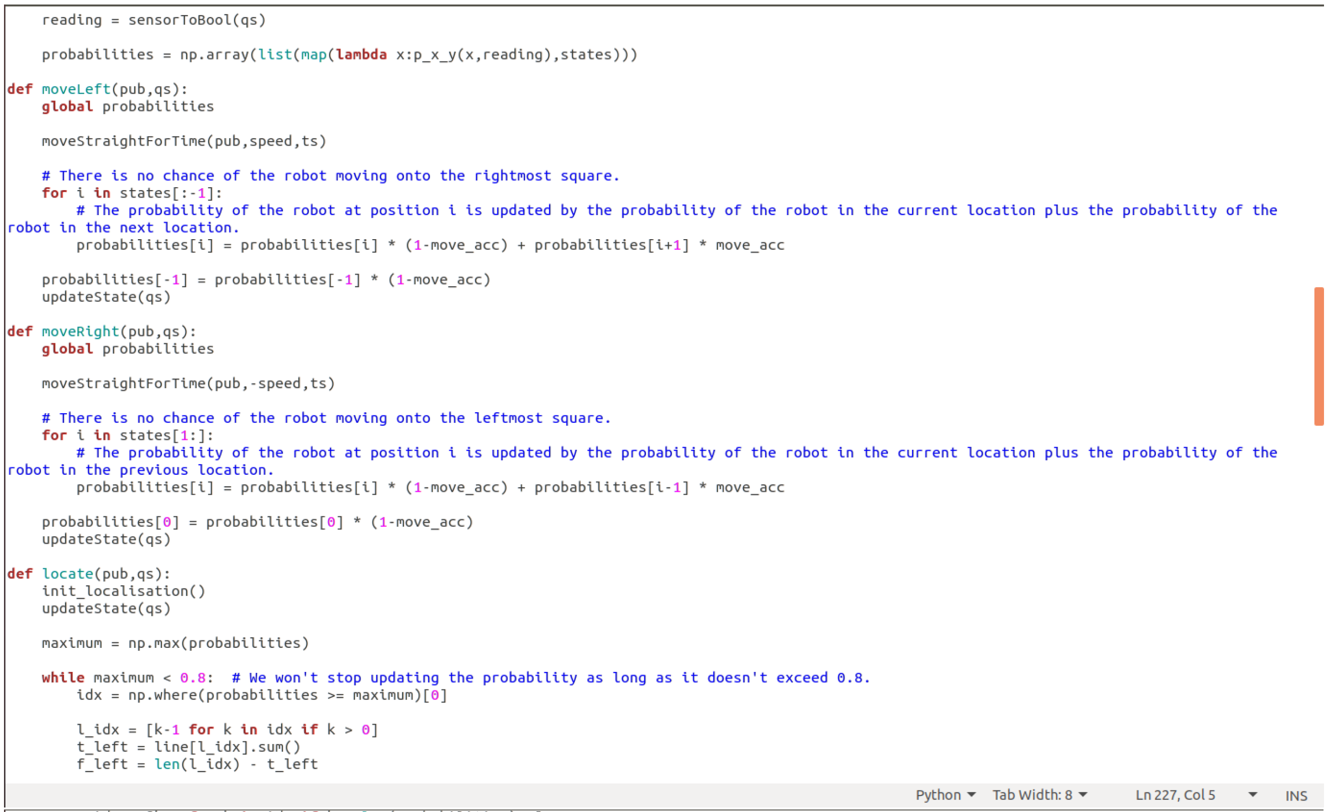
3. Term 2 - Bayesian Localisation: Implementation & Graphical Display



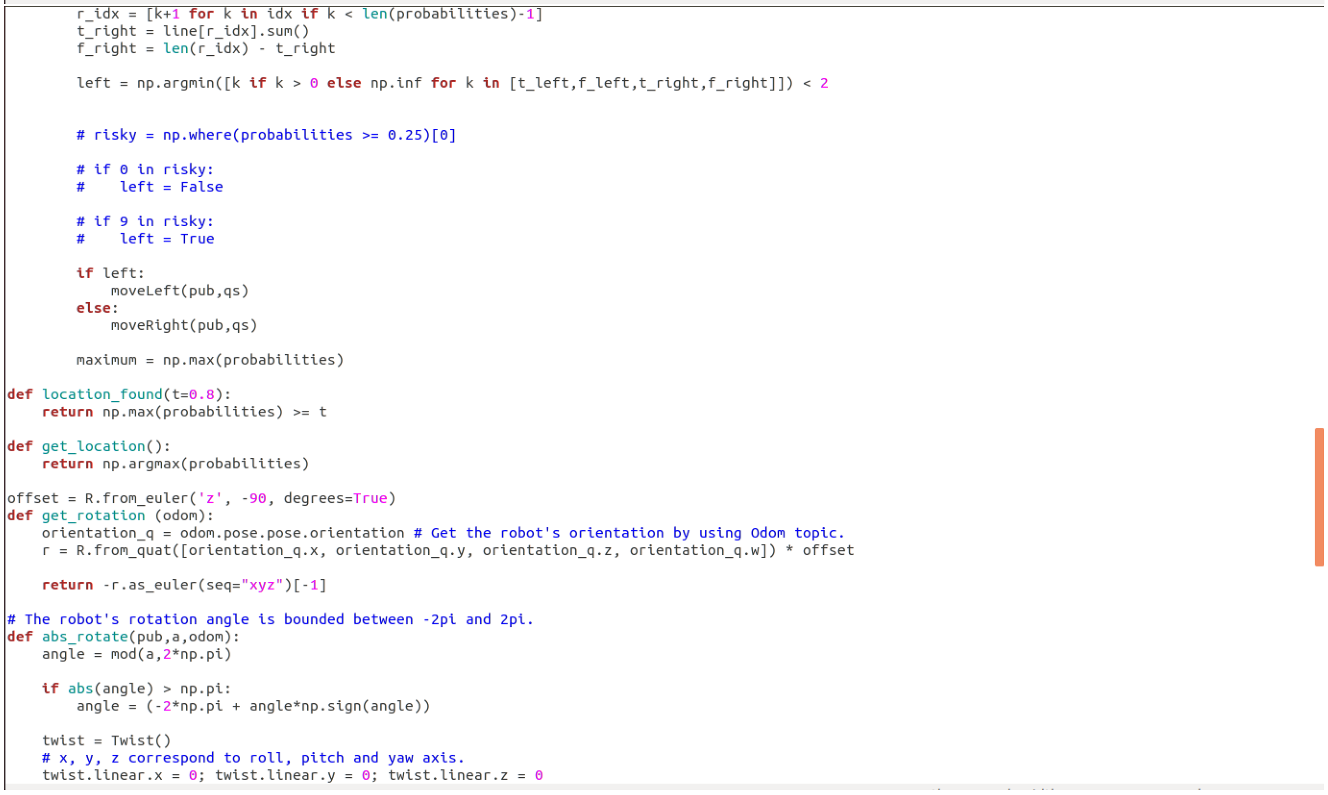
```
import numpy as np
import rospy as rp
from geometry_msgs.msg import Pose,Twist
import matplotlib.pyplot as plt
from scipy.spatial.transform import Rotation as R

reading = False
sensor_acc = 0.9  # The probability that the sensor will give the correct reading.
move_acc = 0.95   # The probability that the robot will move accurately.
speed = 0.05
turn_speed = np.pi/16
ts = 0.1/speed  # The twisting speed of the robot.

global states
global probabilities
global line

states = []
probabilities = []
line = []

def mod(n,base):
    return n - int(n/base) * base

def init_localisation():
    global line
    global pos
    global probabilities
    global states

    # The line variable corresponds to the grid of the map, which is indexed from 0 up to and including 24.
    line = np.array([False]*2+  # The black grid which corresponds to 2 units.
            [True]+  # The white grid which corresponds to 1 unit.
            [False]*3+
            [True]*2+
            [False]*2+
            [True]+
            [False]*3+
            [True]*2+
            [False]*3+
            [True]*2+
            [False]*2+
            [True])
    n = len(line)
    probabilities = np.array([1.0/n]*n)
```



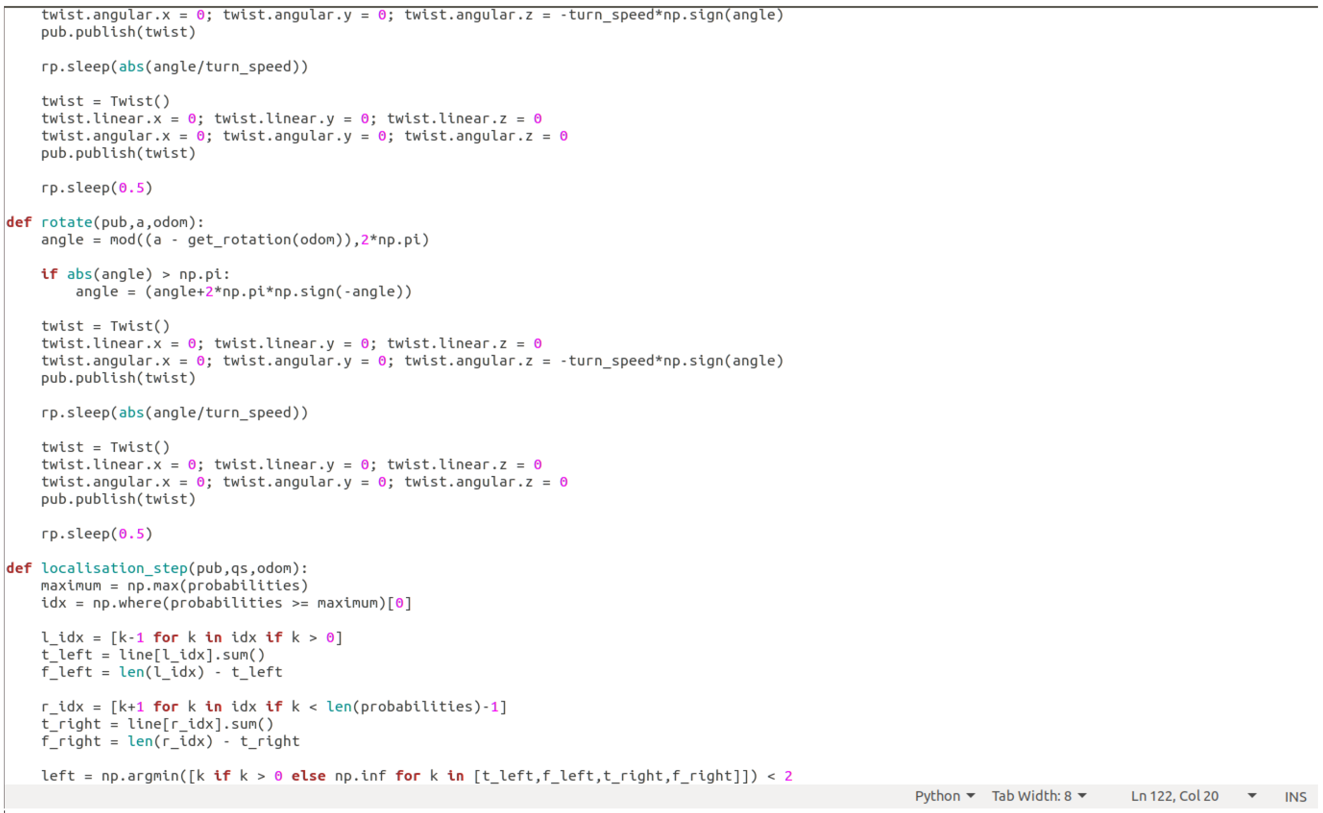
```
    pos = n//2
    states = list(range(len(probabilities)))


def moveStraight(pub,s):
    twist = Twist()
    twist.linear.x = s; twist.linear.y = 0; twist.linear.z = 0
    twist.angular.x = 0; twist.angular.y = 0; twist.angular.z = 0
    pub.publish(twist)

def moveStraightForTime(pub,s,t):
    moveStraight(pub,s)
    rp.sleep(t)
    moveStraight(pub,0)
    rp.sleep(0.5)  # The delay for the robot.

def sensorToBool(qs):
    return np.mean(qs) > 200  # The light sensor must be bright enough here.

# The following three methods works out the probability distribution.
def p_y(e):
    global probabilities
    global states

    s = 0
    for i in states:
        s+= p_y_x(e,i)*probabilities[i]

    return s

def p_y_x(y,x):
    inv = line[x] != y

    if inv:
        return 1-sensor_acc
    return sensor_acc

def p_x_y(x,y):
    global probabilities

    return (p_y_x(y,x) * probabilities[x])/p_y(y)

def updateState(qs):
    global reading
    global probabilities
```



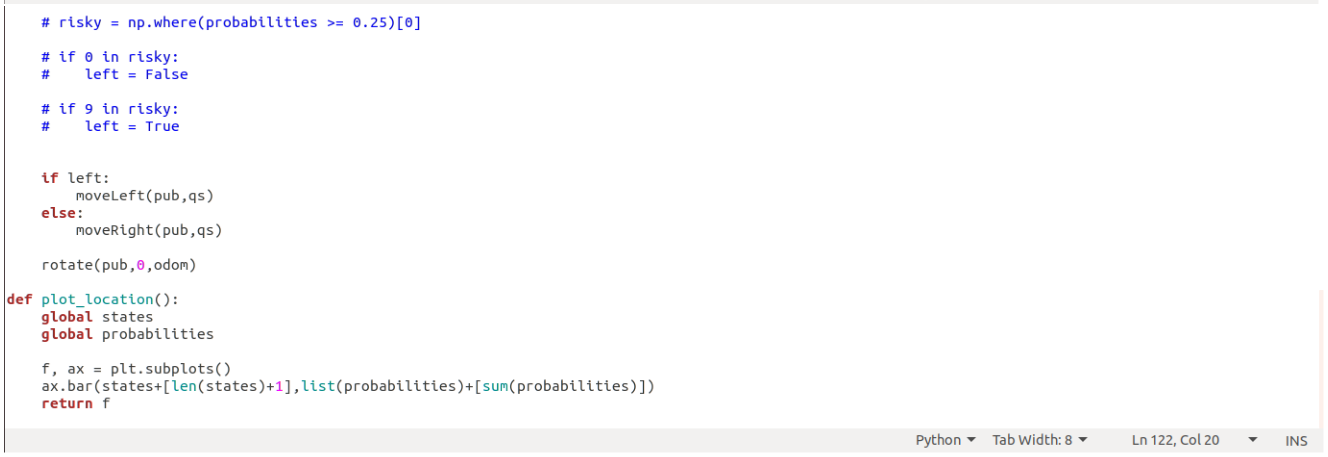
```
    reading = sensorToBool(qs)

    probabilities = np.array(list(map(lambda x:p_x_y(x,reading),states)))

def moveLeft(pub,qs):
    global probabilities

    moveStraightForTime(pub,speed,ts)

    # There is no chance of the robot moving onto the rightmost square.
    for i in states[:-1]:
        # The probability of the robot at position i is updated by the probability of the robot in the current location plus the probability of the
robot in the next location.
        probabilities[i] = probabilities[i] * (1-move_acc) + probabilities[i+1] * move_acc

    probabilities[-1] = probabilities[-1] * (1-move_acc)
    updateState(qs)

def moveRight(pub,qs):
    global probabilities

    moveStraightForTime(pub,-speed,ts)

    # There is no chance of the robot moving onto the leftmost square.
    for i in states[1:]:
        # The probability of the robot at position i is updated by the probability of the robot in the current location plus the probability of the
robot in the previous location.
        probabilities[i] = probabilities[i] * (1-move_acc) + probabilities[i-1] * move_acc

    probabilities[0] = probabilities[0] * (1-move_acc)
    updateState(qs)

def locate(pub,qs):
    init_localisation()
    updateState(qs)

    maximum = np.max(probabilities)

    while maximum < 0.8:  # We won't stop updating the probability as long as it doesn't exceed 0.8.
        idx = np.where(probabilities >= maximum)[0]

        l_idx = [k-1 for k in idx if k > 0]
        t_left = line[l_idx].sum()
        f_left = len(l_idx) - t_left
```



```
        r_idx = [k+1 for k in idx if k < len(probabilities)-1]
        t_right = line[r_idx].sum()
        f_right = len(r_idx) - t_right

        left = np.argmin([k if k > 0 else np.inf for k in [t_left,f_left,t_right,f_right]]) < 2


        # risky = np.where(probabilities >= 0.25)[0]

        # if 0 in risky:
        #    left = False

        # if 9 in risky:
        #    left = True

        if left:
            moveLeft(pub,qs)
        else:
            moveRight(pub,qs)

        maximum = np.max(probabilities)

def location_found(t=0.8):
    return np.max(probabilities) >= t

def get_location():
    return np.argmax(probabilities)

offset = R.from_euler('z', -90, degrees=True)
def get_rotation (odom):
    orientation_q = odom.pose.pose.orientation # Get the robot's orientation by using Odom topic.
    r = R.from_quat([orientation_q.x, orientation_q.y, orientation_q.z, orientation_q.w]) * offset

    return -r.as_euler(seq="xyz")[-1]

# The robot's rotation angle is bounded between -2pi and 2pi.
def abs_rotate(pub,a,odom):
    angle = mod(a,2*np.pi)

    if abs(angle) > np.pi:
        angle = (-2*np.pi + angle*np.sign(angle))

    twist = Twist()
    # x, y, z correspond to roll, pitch and yaw axis.
    twist.linear.x = 0; twist.linear.y = 0; twist.linear.z = 0
```



```python
    twist.angular.x = 0; twist.angular.y = 0; twist.angular.z = -turn_speed*np.sign(angle)
    pub.publish(twist)

    rp.sleep(abs(angle/turn_speed))

    twist = Twist()
    twist.linear.x = 0; twist.linear.y = 0; twist.linear.z = 0
    twist.angular.x = 0; twist.angular.y = 0; twist.angular.z = 0
    pub.publish(twist)

    rp.sleep(0.5)

def rotate(pub,a,odom):
    angle = mod((a - get_rotation(odom)),2*np.pi)

    if abs(angle) > np.pi:
        angle = (angle+2*np.pi*np.sign(-angle))

    twist = Twist()
    twist.linear.x = 0; twist.linear.y = 0; twist.linear.z = 0
    twist.angular.x = 0; twist.angular.y = 0; twist.angular.z = -turn_speed*np.sign(angle)
    pub.publish(twist)

    rp.sleep(abs(angle/turn_speed))

    twist = Twist()
    twist.linear.x = 0; twist.linear.y = 0; twist.linear.z = 0
    twist.angular.x = 0; twist.angular.y = 0; twist.angular.z = 0
    pub.publish(twist)

    rp.sleep(0.5)

def localisation_step(pub,qs,odom):
    maximum = np.max(probabilities)
    idx = np.where(probabilities >= maximum)[0]

    l_idx = [k-1 for k in idx if k > 0]
    t_left = line[l_idx].sum()
    f_left = len(l_idx) - t_left

    r_idx = [k+1 for k in idx if k < len(probabilities)-1]
    t_right = line[r_idx].sum()
    f_right = len(r_idx) - t_right

    left = np.argmin([k if k > 0 else np.inf for k in [t_left,f_left,t_right,f_right]]) < 2
```



```python
    # risky = np.where(probabilities >= 0.25)[0]

    # if 0 in risky:
    #     left = False

    # if 9 in risky:
    #     left = True


    if left:
        moveLeft(pub,qs)
    else:
        moveRight(pub,qs)

    rotate(pub,0,odom)

def plot_location():
    global states
    global probabilities

    f, ax = plt.subplots()
    ax.bar(states+[len(states)+1],list(probabilities)+[sum(probabilities)])
    return f
```

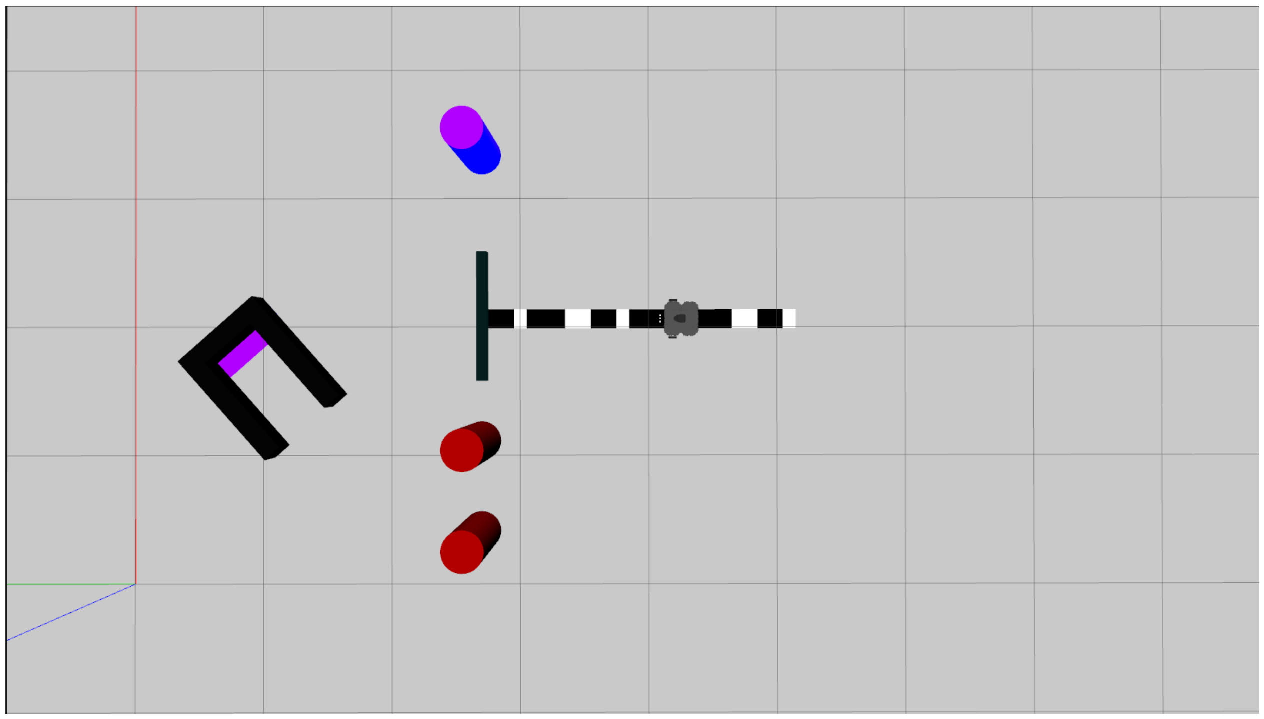


Figure 5:

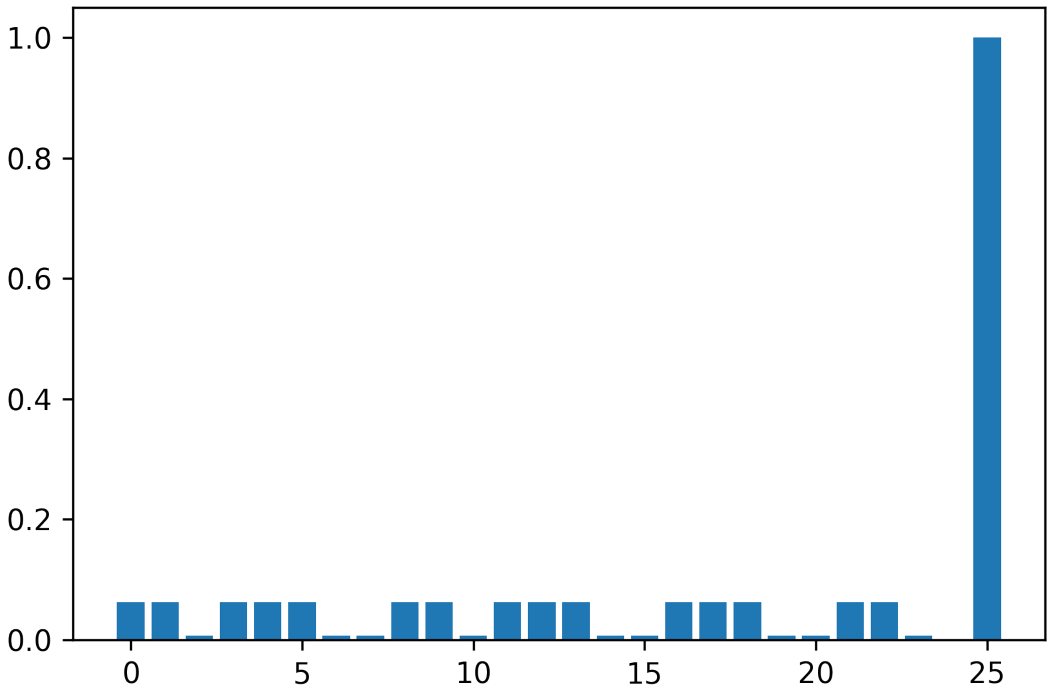


Figure 6:

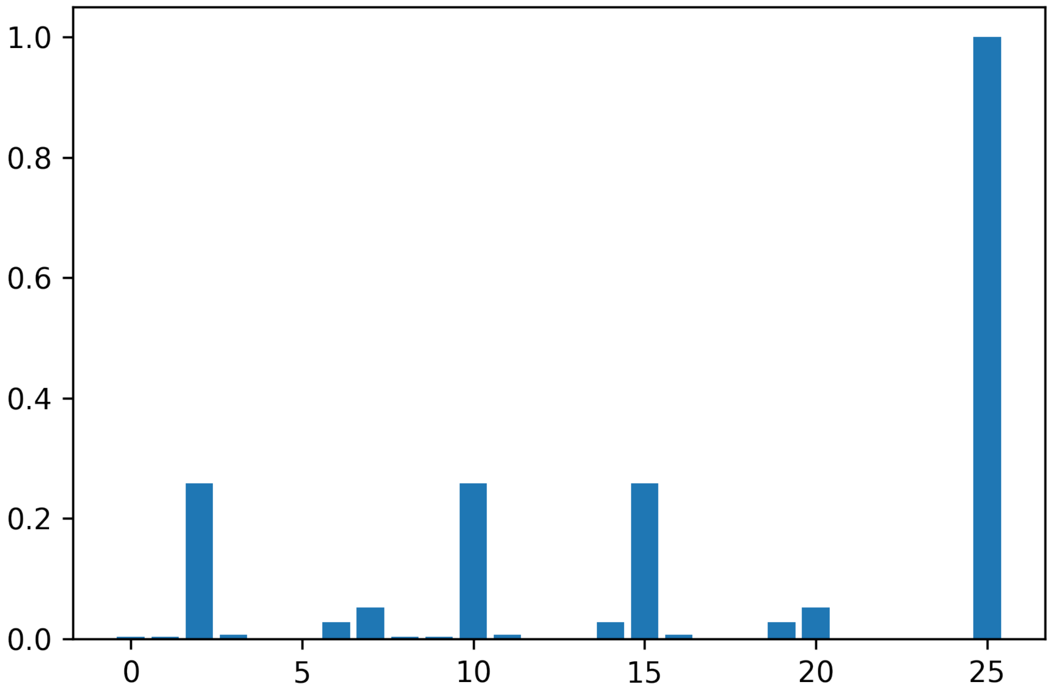


Figure 7:

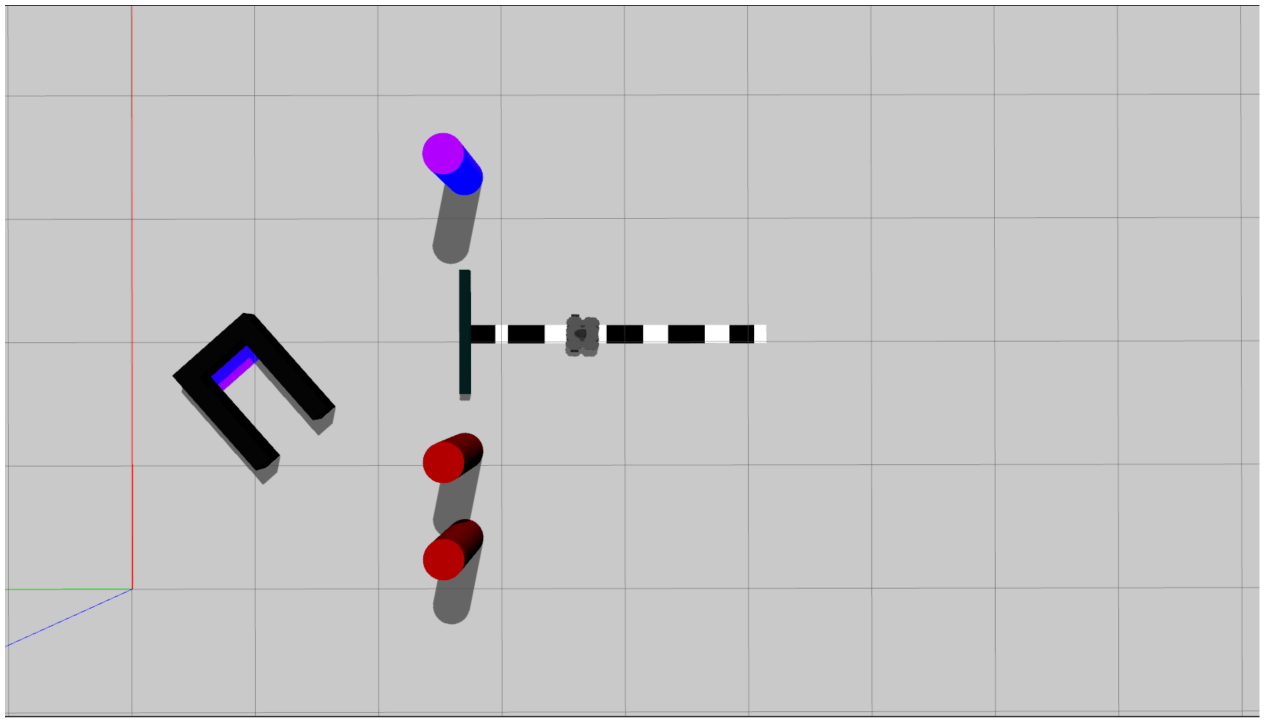


Figure 8:

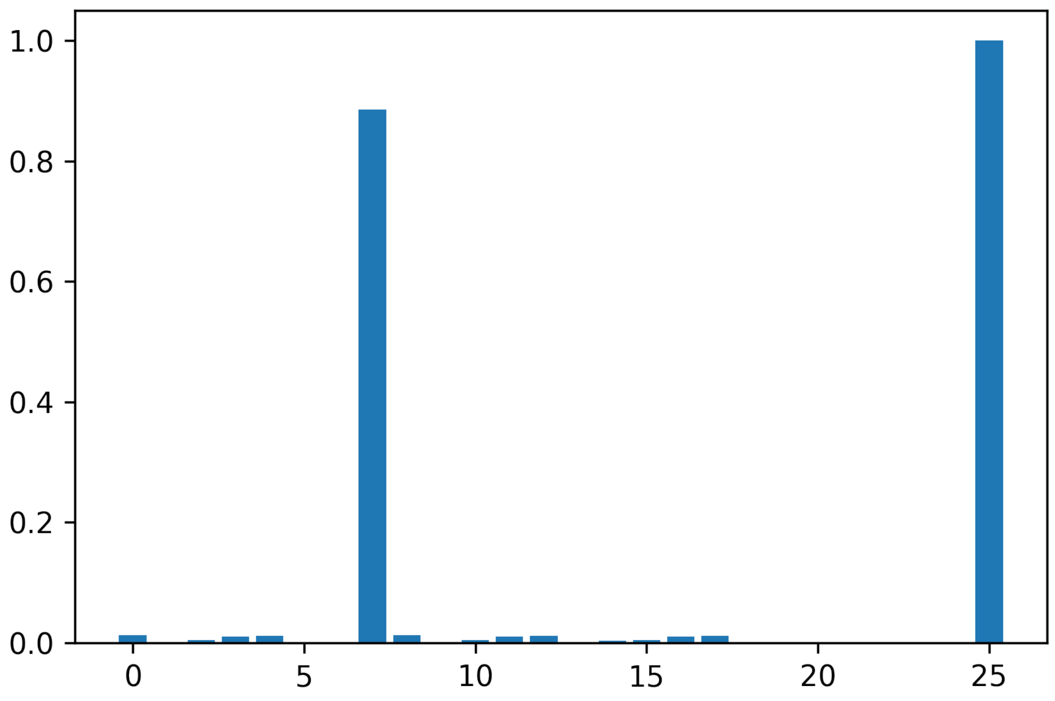


Figure 9: