Lab 5 Report

[Task 1 – Localization with Colored Cylinders]

Lab 5 Task 1 is about 'Localization with Trilateration' which is essentially just finding out where on a grid the robot is at any given point and being able to find that location from anywhere by detecting the distances of three separate points on the grid. Trilateration makes a circle around each of the three known points by giving them a radius equal to the distance from point to robot. You form power lines and a radical axis through circle points and the coordinated where the overlapping circles all meet should pinpoint the x, y value for the robot.

My last lab felt sloppy, stressful, and cluttered so I decided to start Lab 5 on paper before programming which I have not done in a long time. C++ has been my favorite language ever since I studied object-oriented programming and the benefits of class structures so on paper, I designed task one with objects in mind and was happy to discover that in translation Python proved that OOP is one of its strengths. For this task I decided to build a class called Pillar where I could organize the different details I would need, including each pillar's color, each pillar's (x , y) coordinates, and the distance away from the robot measured by the front distance sensor. This object-oriented structure for both 'setting' and printing details did wonders on my mental health and code readability since I have the habit of over-engineering a function definition and getting lost in the weeds.

The first step I took was simply spinning the robot until I scanned the center of three different pillars. Once the pillars' center distance is scanned, my program updates its Pillar object by deciding which color 'ID' was scanned and fixing that pillars 'distance' stat to its corresponding object. The tricky part of this was that sometimes the angle of the pillar wont center exactly depending on where my start location is, and the robot will keep spinning until it does. Slowing the spin velocity on the robot during environment scanning mostly fixed that issue.

The next step was finding out how Trilateration works with my three distances. Given the formulas for Trilateration and a lot of experimenting I correctly implemented a Trilateration function with python code. Formulas and implementation are shown below in the Calculations section. After making sure all grid cells read correctly with my trilateration function the difficulty was moving the robot to every square and only printing visited cells once. I had issues at every step when trying to solve the full map path, so I went with a simpler solution of always scanning then driving left, then starting over at the right column every time I get close to the left wall. This solution may not work when starting at random locations.

[Task 2 – Localization with Internal Walls]

Task two is largely the same focus of being capable of calculating your robot location from any starting point but this time we are ditching the pillars and there are additional wall obstacles throughout the grid. We use the sensors to find out which part of each grid square is walled off, representing walls as compass directions, North, South, East, and West. The goal here is to compute localization based on exclusively the presence of walls in each square on the grid using our Webots toolkit; encoders, IMU, and sensor readings off walls. I never liked the wobbly traversal I used in the previous lab so instead I used the wall avoidance techniques we learned in week one and based my calculations on distance traveled. I use the IMU to check which direction the robot is facing, distance and position sensors to

track how far I have driven and if there are walls to the left front and right. I used the given WNES values and the direction the robot considers the cardinal direction it is facing during each grid cell calculation. Rather than finding the exact x y value of my robot I set the general x y values for the robot pose in range (-15,15) based on the grid cell that the robot is in. The issue with this task is that despite avoiding walls it does not correctly calculate the grid cell when starting at random unknown locations.

Calculations [YouTube : https://youtube.com/playlist?list=PLmQVFU1FBDddYV_4IRW1zfXH6CAKuZjIM]

Trilateration # Ax + By = C

```
A = (-2 * pillars[0].x) + (2 * pillars[1].x)

B = (-2 * pillars[0].y) + (2 * pillars[1].y)

C = (pillars[0].radius) - (pillars[1].radius) - (pillars[0].x **2)
        + (pillars[1].x **2) - (pillars[0].y **2) + (pillars[1].y **2)
```

Dx + Ey = F

```
D = (-2 * pillars[1].x) + (2 * pillars[2].x)

E = (-2 * pillars[1].y) + (2 * pillars[2].y)

F = pillars[1].radius - pillars[2].radius - (pillars[1].x **2)
        + (pillars[2].x **2) - (pillars[1].y **2) + (pillars[2].y **2)
```

Grid Cell Finder

```
Column 1 | -20 < X < -10

Column 2 |-10 < X < 0

Column 3 |   0 < X < 10

Column 4 | 10 < X < 20

Row 1 | 10 < Y < 20

Row 2 | 0 < Y < 10

Row 3 | -10 < Y < 0

Row 4 | -20 < Y < -10
```

```
if (E * A) != (B * D):
    X = (C*E - F*B)/(E*A - B*D)
    Y = (C*D - A*F)/(B*D - A*E)
  else:
    "ERROR – Divide by Zero: (E * A) = (B * D)')"
```