# Lab 4: Map Localization by Joshua-Luke Lacsamana

## Task 1 Mathematics:

The main mathematical feature for task 1 was the inclusion of trilateration for estimating the robot's current post. Given that the robot can recognize three objects, the function

```
trilateration(self, red, green, yellow):
```

Takes in the radiuses of the three objects I chose to grab measurements of.

Based on the math from the slides in class, we can derive the x and y values of a robots pose by first breaking down three circle equations in the form of

$$C1: (x - x_1)^2 + (y - y_1)^2 = r_1^2$$
$$C2: (x - x_2)^2 + (y - y_2)^2 = r_2^2$$
$$C3: (x - x_3)^2 + (y - y_3)^2 = r_3^2$$

After expanding these circle equations, we can then find the line of intersection with circle groupings of C1/C2 and C2/C3, as follows:

$$C1\text{-}C2: (-2x_1 + 2x_2)\,x + (-2y_1 + 2y_2)\,y = r_1^2 - r_2^2 - x_1^2 + x_2^2 - y_1^2 + y_2^2$$
$$C2\text{-}C3: (-2x_2 + 2x_3)\,x + (-2y_2 + 2y_3)\,y = r_2^2 - r_3^2 - x_2^2 + x_3^2 - y_2^2 + y_3^2$$

Furthermore, we can then find the values of the intersection from the two intersection lines of the circle groupings, which will become our updated x and y pose values.

$$x = \frac{(CE - FB)}{(EA - BD)},\ y = \frac{(CD - AF)}{(BD - AE)}$$

## Conclusion for Task 1:

There were no heavy math concepts behind task 1 besides trilateration. The real challenges for task 1 are as follows:

- data manipulation needed to keep track of appropriate rows and columns, and to ensure that we were updating the visited cell array accordingly
- The previously exponential inaccuracies with traveling in a perfect line according to 90-degree angle increments
- Creating a viable algorithm that will make the robot head towards a cell not yet visited

For the first bullet-point, I came up with two arrays; the first one initializing a 4x4 box used to track visited cells, and the second containing the row and column placements for each grid. I then devised the following for loops:

```
# for loop grabs the row ranges, row will update to the index of the row
row_ranges = [(1, 2, 0), (0, 1, 1), (-1, 0, 2), (-2, -1, 3)]
for lower, upper, index in row_ranges:
if lower <= y <= upper:
row = index
break

# for loop grabs the column ranges, column will update to the index of the column
column_ranges = [(-2, -1, 0), (-1, 0, 1), (0, 1, 2), (1, 2, 3)]
for lower, upper, index in column_ranges:
if lower <= x <= upper:
column = index
break
```

Essentially, whenever the x and y values are updated, the for loop grabs the lower and upper bounds I assigned in the row/column_ranges array, and if they fit the criteria, it gives out the correct row and column that the robot is in. From there, another for loop enumerates through my array containing row and column placements, and if the row and column matches a placement, it breaks out and keeps the index, which corresponds to our grid location.

For the second bullet-point, I realized I can just fix the angle of the robot if its skewed slightly with the use of the range() method in python. Given that it is within a certain range, adjust it to get as close to a 90-degree incremented angle as possible. I dealt with a lot of trial and error with this function, but I was finally able to get range values that seem to fix the robots orientation every time.

For the third bullet-point, it took me a while to brainstorm this algorithm, but I figured out that if you compare the findings of the empty cell index to the current robots index, you can create movements accordingly. Here is the function:

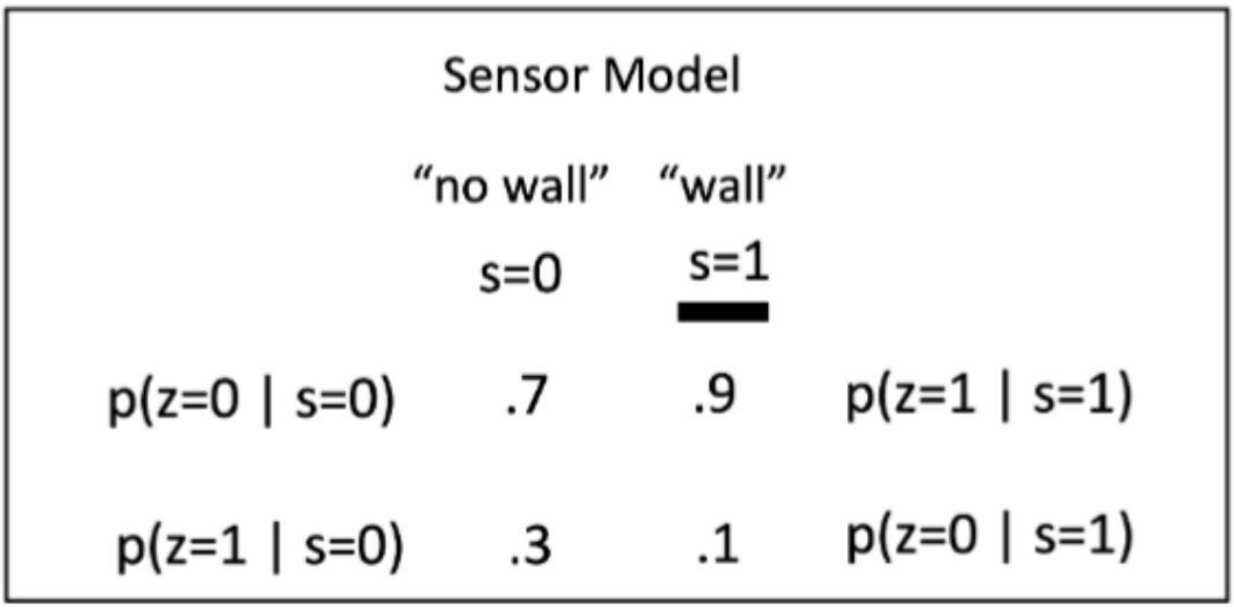
```
def find_empty_cell(self, current_index, track_visited_cells):
task_complete = False
# The breakdown of the algorithm is as follows:
# 1. Find the empty cell index
# 2. Check if the empty cell index is greater than the current index and the absolute
difference between the two is less than or equal to 3, if so, if the current position and the
current empty spot is on the same row, move left until you encounter it, if it is a special
case where the empty cell is below, move down
# 3. Check if the empty cell index is less than the current index and the absolute difference
between the two is less than or equal to 3, if so, if the current position and the current
empty spot is on the same row, move right until you encounter it, if it is a special case
where the empty cell is above, move up
# 4. If the empty cell index is greater than the current index and the absolute difference
between the two is greater than 3, move down
# 5. If the empty cell index is less than the current index and the absolute difference
between the two is greater than 3, move up
empty_cell_index = None

for index, cell in enumerate(track_visited_cells):
if cell == '.':
empty_cell_index = index
break
if empty_cell_index == None:
print("Webot has visited all cells!")
return
# print("Empty Cell Index:", empty_cell_index)
# print("Current Index:", current_index)
# print("absolute difference:", abs(empty_cell_index - current_index))
if empty_cell_index > current_index and abs(empty_cell_index - current_index) <= 3:
if current_index in range (0, 4) and empty_cell_index in range(0, 4):
self.rotate_to_this_degree(0, 4)
elif current_index in range(4, 8) and empty_cell_index in range(4, 8):
self.rotate_to_this_degree(0, 4)
elif current_index in range(8, 12) and empty_cell_index in range(8, 12):
self.rotate_to_this_degree(0, 4)
elif current_index in range(12, 16) and empty_cell_index in range(12, 16):
self.rotate_to_this_degree(0, 4)
else:
self.rotate_to_this_degree(270,4)
elif empty_cell_index < current_index and abs(empty_cell_index - current_index) <= 3:
# print("go left")
if current_index in range (0, 4) and empty_cell_index in range(0, 4):
# print("should work")
self.rotate_to_this_degree(180, 4)
elif current_index in range(4, 8) and empty_cell_index in range(4, 8):
self.rotate_to_this_degree(180, 4)
elif current_index in range(8, 12) and empty_cell_index in range(8, 12):
self.rotate_to_this_degree(180, 4)
```

```
elif current_index in range(12, 16) and empty_cell_index in range(12, 16):
self.rotate_to_this_degree(180, 4)
else:
self.rotate_to_this_degree(90, 4)
elif empty_cell_index > current_index and abs(empty_cell_index - current_index) > 3:
# print("go down")
self.rotate_to_this_degree(270,4)
else:
# print("go up")
self.rotate_to_this_degree(90, 4)
return
```

## Task 2 Mathematics:

Task 2 involved implementing a bayes filter probability function into my program. We are given the following sensor model to calculate our probabilities in each of our cells on the map.




Our Z values correspond to the absense or presence of obstacles in our enviornment, in our specific case, the walls around our grids. Our z values are collected using self.get_lidar_range_image()[left, right, front readings]. However, orientation is especially important in ensuring the correctness of our probabilities.

For example, if our robot was oriented facing north, then our front, left, and right distance sensors would match to the appropriate north, west, and east coordinates. If our robot was facing south, however, then our front, left, and right sensors would not match the

coordinates. This took me a while to realize, as my first probabilistic outcomes did not match where the robot was. Here is the solution i came up with to address this problem:

```
def adjust_sensor_readings(self, front_sensor, left_sensor, right_sensor, back_sensor):
# Define sensor mapping relative to orientation
orientation_mapping = {
"N": {"front": front_sensor, "left": left_sensor, "right": right_sensor},
"E": {"front": left_sensor, "left": back_sensor, "right": front_sensor},
"S": {"front": back_sensor, "left": right_sensor, "right": left_sensor},
"W": {"front": right_sensor, "left": front_sensor, "right": back_sensor}
}
orientation = self.get_compass_reading()
if orientation in range(80, 100):
robot_orientation = "N"
elif orientation in range(170, 190):
robot_orientation = "W"
elif orientation in range(260, 280):
robot_orientation = "S"
elif orientation in range(350, 361) or orientation in range(0, 11):
robot_orientation = "E"
# Get sensor readings based on robot's orientation
orientation_sensors = orientation_mapping[robot_orientation]
# Extract adjusted sensor readings
adjusted_front_sensor = orientation_sensors["front"]
adjusted_left_sensor = orientation_sensors["left"]
adjusted_right_sensor = orientation_sensors["right"]
return adjusted_front_sensor, adjusted_left_sensor, adjusted_right_sensor
```

This code defines a dictionary, which, when given left, front, right, and back distance sensor measurements, will place these measurements accordingly to the right returned variables given the current robots orientation.

After these adjustments have been made, grabbing our z values is as follows:

```
z_left = 1 if adjusted_left_sensor < 0.8 else 0
z_right = 1 if adjusted_right_sensor < 0.8 else 0
z_front = 1 if adjusted_front_sensor < 0.8 else 0
```

Z left, right, and front will be one if a wall is detected, otherwise 0

Because the robot has preexisting knowledge of the walls around all the grids cells, we can use this information to grab our states S with the following code:

```
for i, cell in enumerate(maze):
# Grab S values for each cell
        next_west = 1 if cell.west == "W" else 0
        next_north = 1 if cell.north == "W" else 0
```

```
next_east = 1 if cell.east == "W" else 0
cell_pb = self.probability_for_lab4(next_west, z_left) *
self.probability_for_lab4(next_north, z_front) * self.probability_for_lab4(next_east,
z_right)

probability_map[i] = cell_pb
```

The “maze” we are looping over is the following:

```
maze = [
Cell('W','W','O','W', False), Cell('O','W','O','W', False), Cell('O','W','O','O', False),
Cell('O','W','W','O', False),
Cell('W','W','O','O', False), Cell('O','W','O','O', False), Cell('O','O','W','O', False),
Cell('W','O','W','O', False),
Cell('W','O','W','O', False), Cell('W','O','O','W', False), Cell('O','O','W','W', False),
Cell('W','O','W','O', False),
Cell('W','O','O','W', False), Cell('O','W','O','W', False), Cell('O','W','O','W', False),
Cell('O','O','W','W', False)
]
```

This maze comes occupied with cell objects, which is a class provided to us in the course's canvas files. Essentially, based on a West-North-East-South orientation, if there is a “W”, it represents there's a wall, with “O” meaning no wall. Looping over this maze and accessing the cell’s variables, we can check if there is a wall, which will make our S = 1 according to our sensor model. Given our S and Z, we now need to grab our probability values which are accessed through self.probability_for_lab4()

```
def probability_for_lab4(self, S, Z):

prob = 0

if S == 0:
        if Z == 0:
                prob = 0.7
else:
prob = 0.3

else:
        if Z == 1:
                prob = 0.9
        else:
                prob = 0.1

return prob
```

This code is straightforward, as it copies the exact model as our sensor model provided.

After multiplying our left, right, and forward probabilities together into a cell probability, we place this resulting value into the following list:

```
probability_map =[0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0]
```

We repeat the process until all values are filled on the grid.

Because we want the normalized probability of all of our grids, we include the following steps:

```
norm_factor = 1/sum(probability_map)
```

To grab our normalization factor, we put a 1 over the summation of all the probability values in our array.

```
for i, prob in enumerate(probability_map):
probability_map[i] = norm_factor * prob
```

Finally, given this norm factor, we multiply it with each of our preexisting probabilities on our list.

After this step is complete, we now have all our normalized probabilities for each grid in our map. This probability helps us see what the likelihood of our robot's position is on our cell grids.

## Conclusion for Task 2:

This task was by far the most extensive task for me. This task involved a ton of moving parts, and a lot of what I assumed beforehand with the nature of encoders and measurements proved to be fatal for most of my time.

For task 2, the biggest hurdles came out to be the following:

- Failing to update my x and y pose appropriately
- The algorithm required to move in a maze one meter at a time without hitting the wall
- Understanding how to implement my distance sensor measurements based on the orientation of my robot

For the first problem, I went to Chance's TA office hours, where he was able to explain in depth a lot of the issues that students encounter with their x and y pose updates. I learned

a fundamental amount of information, such as the fact that calling my update pose function after a big distance movement leads to high levels of inaccuracy due to the inconsistencies with my theta. I learned that the update pose call must be called at every timestep while my vehicle is moving. The other important thing I learned was being able to have my x and y values be built into my robot class itself as a self-variable. This made my life so much easier with function calls. All my previous movement functions needed to be revised to accommodate these new updates to my x and y pose.

For the second problem, I originally tried a variation of random movements whenever my front distance sensor encountered a wall, but that would prove to be tedious and nearly impossible for my robot to traverse all the cells. Instead, I took a step back and figured out what the robot should do should it encounter certain walls with its distance sensor. Once it made correct rotation movements based around that, the rest was history.

For the third problem, I already went into some detail in the mathematics section of my lab report, but the way I figured out something was wrong was by noticing that the probabilities that were being printed were only correct when my robot was facing north. I could tell the rest of my math was working fine, just not correctly because of the orientation issue.

One thing that I would like to improve is my rotation adjustment function. There are many cases where the rotation will take the long way, and I know there is a way to have the robot rotate in the correct direction to fix it.