Introduction

This report goes over the occupancy grid algorithm which is used to map the robot's surroundings and generate the maze maps required for this lab. The robot uses its onboard sensors to scan its surroundings to detect free and occupied spaces and updates this on the map. The occupancy grid algorithm is explained further in the sections that follow:

Task 1A – Occupancy Grid

For this task, I implemented the occupancy grid algorithm by creating a matrix of matrices – a 4D array using NumPy. Each cell in the map contains sub cells which are initially marked with an "unknown" value and then are updated with either "empty" or "occupied" values as the robot navigates the maze. For this lab, I implemented a 3×3 sub cell matrix for each cell. *Figure 1* shows a visual representation of this occupancy grid.

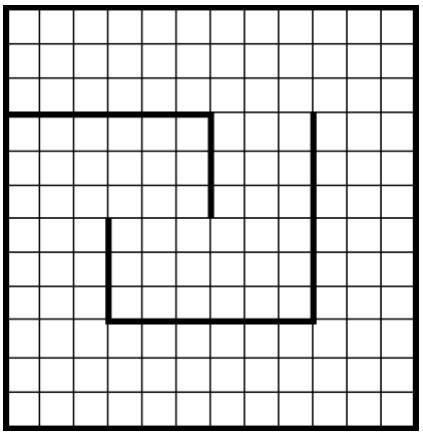

Figure 1: 12x12 occupancy grid (3x3 sub cells per cell). Darker lines are walls.

Each sub cell is initialized with the unknown value. If there exists no wall/obstacle, it is updated as "empty". If there exists a wall/obstacle, it is updated as "occupied". To update, LiDAR sensor readings from the robot are passed into a function that calculates log odds.

The inverse sensor model used in updating the occupancy grid is described below:

$$p(m_i|z_t, s_t) = \begin{cases} 0.3 & empty \\ 0.6 & occupied \\ 0.5 & unknown \end{cases}$$

The log odds inverse sensor model used to update the sub cells in the occupancy grid is described below:

$$l(m_i|z_t, s_t) = log \frac{p(m_i|z_t, s_t)}{1 - p(m_i|z_t, s_t)}$$

Using the inverse sensor model and log odds calculations above, I was able to update each sub cell in the occupancy grid accordingly. *Figure 2* shows a log odds update for the sub cells in front of the robot. If the robot revisits a cell, the log odds for the sub cells are updated by adding the previous log odds + current log odds – prior.

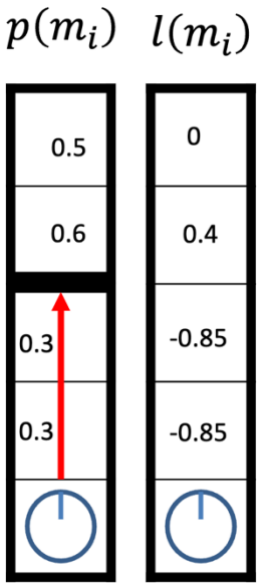

Figure 2: Inverse sensor model readings (left). Sub cells updated with log odds (right).

For my 3×3 sub cell configuration, the values for occupancy are 0.3 for all empty cells at a distance that is less than the measured wall, 0.6 for occupied cells at a distance equal to the measured wall, and 0.5 for unknown cells not calculated by the sensor. In *Figure 3*, you can see how the robot's surroundings matches with the occupancy grid for the cell it is currently in.

Figure 3: Robot position in map (bottom) and corresponding occupancy grid (top).

As seen by the occupancy grid matrix above for cell 4 in the 4D array, the log odds calculation of 0.41 (occupied sub cell) and -0.85 (empty sub cell) accurately represent what is seen in the Webots simulation. These values also match those shown in *Figure 2*. There are walls to the left and front of the robot but none behind or to the right of it. The value in the center of the matrix will always be empty because that is the sub cell right underneath the robot. Additionally, for my implementation, although the unknown sub cells are assigned 0.5, they are automatically updated with the log odds calculation (log odds of 0.5 is zero) for my implementation to work. *Figure 4* shows an excerpt of my code that updates the top middle matrix sub cell using log odds:

```
# update North sub cells
sub_cells[0, 1] = round(sub_cells[0, 1] - prior + log_inv_sensor_model(z_val[0], c), 2)
if z_val[0] == occupied_val:
    sub_cells[0, 0], sub_cells[0, 2] = sub_cells[0, 1], sub_cells[0, 1]
```

Figure 4: Excerpt code that updates top middle sub cell with log odds.

In the code excerpt above, *sub_cells* is a matrix part of the 4D array. The top middle sub cell is specified by *sub_cells[0, 1]* and the log odds with the *current sensor reading + previous – prior* are calculated. The *z_val* is the occupancy value, *c* is the block size. The IF-statement below

determines if the top middle sub cell was occupied. If true, it updates the sub cells to the left and right of it as occupied. In *Figure 3,* the occupancy matrix for the cell shows this in action. This is representative of the actual map and will come in handy for generating the map of the maze discussed in the next section.

Task 1B – Generating Map

Generating the map of the maze is quite simple now that we have an occupancy grid. In my program, the map of the maze is printed in real time into the console. The generated maze is essentially the occupancy grid but instead of a hundred float values that are hard to read immediately, it is represented with 'W' for wall, '→' for visited (and direction), and empty space for empty occupancy. Due to the difficulties of printing 4D arrays, I had to remap every value from the 4D occupancy grid to a 2D array. In my code, this is handled by the *wall_mapping* function.

To map '→' which shows the robot's direction and marks a cell as "visited", the *wall_mapping* function finds the exact sub cell the robot is on. It accomplishes this using 4 variables: *I, j, k, l*. The variables *i* and *j* are used to get the cell in the 4D array. The variables k and l are used to get the exact sub cell in that cell. By finding the index at *(I, j, 1, 1)*, we can mark the robot position. To map 'W' which represents a wall, the *wall_mapping* function finds each sub cell in the cell the robot that is currently in that is equal to the log odds calculation for an occupancy value of 0.6 (occupied). Since this can be any sub cell, we find and assign the indexes at *(I, j, k, l)* with 'W'. So far, this function maps every cell as visited and the walls – whatever is not mapped is left empty which defines it as the empty occupancy value.

In conclusion to generating the map, the occupancy grid plays a crucial component as its values are used to visually generate the map of the maze. A fully traversed maze results in a completed occupancy grid and therefore, a completely generated map. *Figure 5* below shows an example of the actual maze map and generated maze based on occupancy values.

Figure 5: Maze in Webots simulator (left). Generated Maze from occupancy values (right).

Conclusion

In summary, Lab 5 had its challenges which I've encountered while trying to understand and implement the occupancy grid algorithm. Upon understanding how the algorithm worked, I was unsure on what data structure I wanted to use to create the occupancy grid. Eventually, I found out that NumPy had 4D arrays which worked perfectly for creating an occupancy grid. Thanks to the previous labs, I was able to get the general motion and sensor readings working quickly and easily however, calculating log odds had its fair share of challenges where I encountered incorrect values. To fix this, I had to set the unknown values straight to log odds so when the function calculated it, the previous log odds would be zero. The biggest challenge was printing the maze. At first, I tried to print the entire occupancy grid and I quickly found out that printing a 4D array consisting of 400 sub cells looked like a mess. I decided to implement a 3×3 sub cell matrix instead for this reason however, realized after mapping the 4D array to 2D array, I could've done a 5×5 sub cell matrix. Fortunately, the 3×3 sub cells per cell worked completely fine and generated the maze accurately. Since I wasn't printing the occupancy grid all at once, I decided to print each occupancy matrix of a cell every time the robot enters a new cell. Although I could print these values with the 2D array, I already was generating the maze and printing the occupancy matrix cell by cell was easier to read and follow along. I could've used matplotlib for generating the maze but that's less fun than printing it straight into the console.