NEURAL NETWORK ALGORITHM FOR COMPLETE COVERAGE PATH PLANNING IN INDUSTRIAL ROBOTIC PLATFORMS: A SIMULATION-BASED STUDY

by

CHUKWUBUIKEM VICTOR EWELIKE

Submitted to the Graduate and Professional School of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Chair of Committee,	Chukwuzubelu Ufodike
Committee Members,	Nie Xiaofeng
	Darbha Swaroop
Head of Department,	Reza Langari

August 2023

Major Subject: Engineering Technology

ABSTRACT

This thesis discusses complete coverage path planning (CPP) algorithms used for robotic systems in dynamic and changing environments. The focus is on the Neural Network algorithm [9] and its adaptation for practical use on an industrial-ready robotic platform. Various approaches to CPP are described, including offline and online algorithms, and a structured approach using grid mapping-based methods. The thesis also mentions the physical implementation of the algorithm on a multi-robotic system and discusses the limitations of current methods for industrial applications. The objective of the research is to develop a system for complete coverage path planning with higher coverage completeness, lower path repetition rate, and less path execution time. The research is limited to real-world simulations using Gazebo World and Robot Operating System (ROS).

DEDICATION

This work is dedicated to the robotics research efforts of the Digital Manufacturing and Design (DMD) lab.

ACKNOWLEDGEMENTS

I would like to thank my committee chair, Dr. Chukwuzubelu Ufodike for his guidance and support throughout this research. I would like to thank my committee members, Dr. Nie Xiaofeng, and Dr. Darbha Swaroop for their support of this work.

Thanks also go to my friends and colleagues and the department faculty and staff for making my time at Texas A&M University a great experience. Thanks to the ETID department for sponsoring my academics as a graduate teaching assistant. Finally, thanks to my mother and father for their encouragement.

Colleagues:

- Steven Longa
- Daniyal Ansarid
- Brey Caraway
- Gaius Nzebuka Ph.D.
- Yifan Wu

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supervised by a thesis committee consisting of Professor Dr. Chukwuzubelu Ufodike [advisor], Dr. Nie Xiaofeng of the Department of Engineering Technology Industrial Distribution, and Dr. Darbha Swaroop of the Department of Electrical and Computer Engineering. The Research Goal and Objective were provided by Dr. Chukwuzubelu Ufodike. All other work conducted for the thesis was completed by the student independently.

Funding Sources

Graduate study was supported by the Engineering Technology and Industrial Distribution department at Texas A&M University.

This work was also made possible in part by the Digital Manufacturing and Distribution lab (DMD-Lab), and Professor Ufodike Research group (PURG).

NOMENCLATURE

NN	Neural Network
CCPP	Complete Coverage Path Planning
BINN	Bio-inspired Neural Network
CPP	Coverage Path Planning
COM	Center of Mass
ROS	Robot Operating System
DMD	Digital Manufacturing and Design

TABLE OF CONTENTS

Page

Page

TABLE OF FIGURES

Page

LIST OF TABLES

Page

1. INTRODUCTION

Coverage path planning (CPP) refers to the process of identifying a route that covers all parts of an area while avoiding obstacles [7]. This process entails discretizing the environment and planning paths with minimal overlap. CPP algorithms can be categorized as either offline or online. Offline CCP algorithms assume full knowledge of the environment, resulting in optimal coverage, but this is not always a practical scenario. Online CCP algorithms, on the other hand, use sensors to determine environmental conditions in real-time, making it a more realistic approach but may generate sub-optimal results. Various methods can achieve CPP. One approach is the random sweeping of the workspace used by Roomba [8]. However, this approach has the problem of multiple paths overlapping, especially when implemented. A more structured approach to CPP is breaking the environment-free space into cells that are easy for robots to cover with simple motions while avoiding obstacles. One such approach is using the neural network method proposed by Yang et al [9]. This is a grid mapping-based method that discretizes a 2D space into a network of neurons. CPP is achieved based on the neural activity of each cell, which works to attract or repel the robot. Ding and Luo et al propose a complete coverage neural network path planning for unmanned surface vehicles [10]. In [11], a bio-inspired neural network path planning for complete coverage in unknown environments is outlined. A biologically inspired neural network methodology with safety considerations for real-time collision-free navigation of an intelligent vehicle with safety considerations in a non-stationary environment is proposed [12]. In this work, an adaptation of the bio-inspired neural network (BINN) algorithm for discrete and centralized programming [13] is explored by Sun, Zhu, Tian, and Luo for underwater autonomous vehicles. This algorithm simplifies the shunting equation used to model the neural activity, thus ensuring a more computationally

efficient method. In [37], an implementation of the neural dynamic algorithm on a two-wheeled driven robot is used to achieve area coverage. Finally, [16] proposes a different path selection strategy using a heuristic approach based on the rolling window.

2. BACKGROUND

2.1. Neural Network Approach

Hodgkin and Huxley proposed the circuit model for the nerve cell membrane and a dynamic equation to describe it (the Hodgkin and Huxley model). Grossberg applied this model to biology, machine vision, sensing, motion control, and other fields by proposing the shunting equation.

The concept behind the Neural Network model is to treat the dynamic and changing environment as a neural network. The unclean neurons will attract the robot across the entire workspace, while obstacle neurons have a local effect to prevent collisions. The path of a mobile robot is dynamically planned in real-time based on the activity landscape.

2.1.1. Original Equation

A computational model for a path of membrane: dynamics of the voltage across the membrane V_m is described using a state equation:

A computational model for the path of a membrane involves describing the dynamics of the voltage across the membrane V_m using a state equation:

$$C_m \frac{dV_m}{dt} = -(E_p + V_m)\, g_p + (E_{Na} - V_m)\, g_{Na} - (E_K + V_m)\, g_K \quad \textbf{(1)}$$

C_m is the membrane capacitance and E_K, E_{Na}, and E_p are the Nernst potentials for potassium ions, sodium ions, and passive leak current in the membrane. Parameters g_K, g_{Na}, *and* g_p represent the conductance of potassium, sodium, and passive channel.

By setting the terms and variables in (1):

$$\begin{aligned} C_m &= 1 \\ x_i &= E_p + V_m \\ A &= g_p \\ B &= E_{Na} + E_p \\ D &= E_k - E_p \\ S_i^e &= g_{Na} \\ S_i^i &= g_K \end{aligned} \tag{2}$$

The shunting equation is obtained by substituting the variables in (2):

$$\frac{dx_i}{dt} = -Ax_i + (B - x_i)\, S_i^e(t) - (D + x_i)\, S_i^i(t) \tag{3}$$

where x_i is the neural activity of the *i*th neuron. Parameters *A, B,* and *D* are non-negative constants representing the passive decay rate (A), and the upper (B) and lower bounds (D) of the neural activity. S^e and S^i are excitatory and inhibitory inputs to the neuron.

2.1.2. Model Algorithm

The computational model can be further refined by appropriately defining the external inputs from the changing environment and the internal neural connections. In this way, unclean areas can be modeled as peaks and obstacles as valleys, ensuring the optimal path for the membrane is found. **Figure 1** shows the representation of the grid environment.

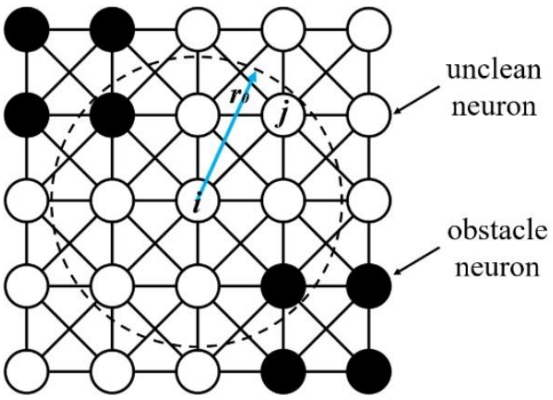


Figure 1. Neural Grid Representation Reprinted from [10]

Based on the original equation (3), the dynamic of the *i*th neuron in the neural network can be characterized by a shunting equation as

$$\frac{dx_i}{dt} = -Ax_i + (B - x_i)\left([I_i]^+ + \sum_{j=1}^{k} w_{ij}\,[x_j]^+\right) - (D + x_i)\,[I_i]^- \tag{4}$$

k is the number of neural connections of the *i*th neuron to its neighboring neurons. The external input I_i to the *i*th neuron is defined as

$$I_i = \begin{cases} E, & \text{if it is an unclean area} \\ -E, & \text{if it is an obstacle area} \\ 0, & \text{otherwise} \end{cases} \tag{5}$$

E ≫ B is a very large positive constant:

$$[a]^+ = \max\{a, 0\}$$

$$[a]^- = \max\{-a, 0\} \tag{6}$$

The connection weight $w_{i,j}$ between the *i*th and *j*th neurons can be defined as

$$w_{i,j} = f(|q_i - q_j|) \tag{7}$$

q_i is the location of each neuron, and *f(a)* can be any monotonically decreasing function.

- The unclean areas globally attract the robot by neural activity propagation.
- In this computational model, positive neural activity can propagate throughout the entire state space, while negative activity is restricted to a local region.
- Obstacles have only local effects to avoid collisions.
- The locations of the unclean areas cleaned areas, and obstacles may vary with time (dynamic).
- The environmental dynamics are represented in **Figure 2**.

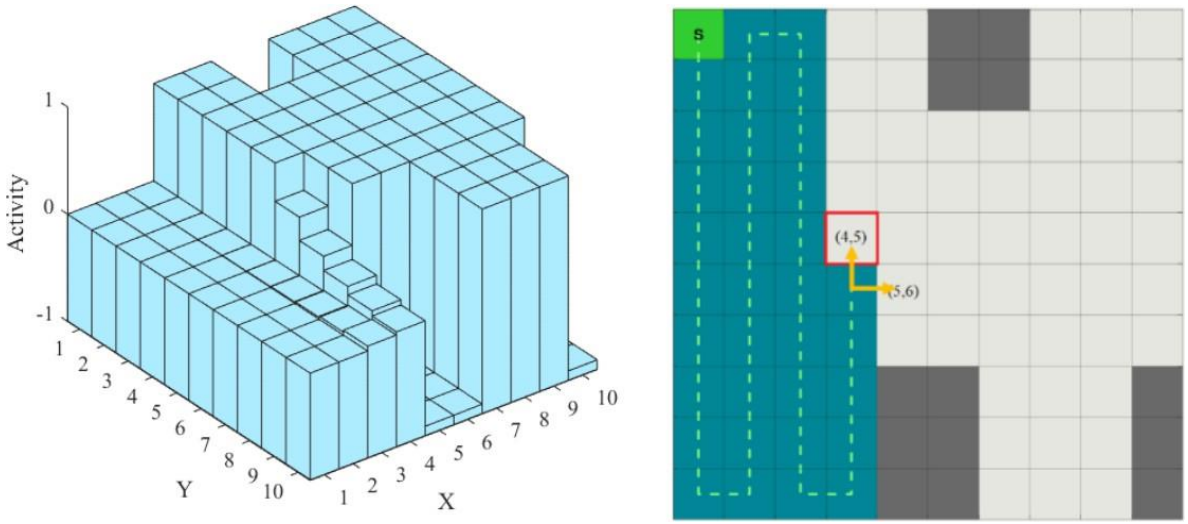


Figure 2. Neural Landscape Dynamics Reprinted from [10]

2.1.3. Next position decision formula

The robot path is generated from both the dynamic activity landscape and the previous robot location to achieve fewer changes in navigation direction:

$$p_n \Leftarrow x_{p_n} = \max\{x_j + cy_j, j = 1, 2, \cdots, k\} \quad \textbf{(8)}$$

As is observed in **Figure 3**, p_c is the current robot location, and p_n is the next robot location. k is the number of neighboring neurons of p_c. X_j is the neural activity of the jth neuron, and y_j is a monotonically decreasing function of the difference between the current and the next robot's moving directions.

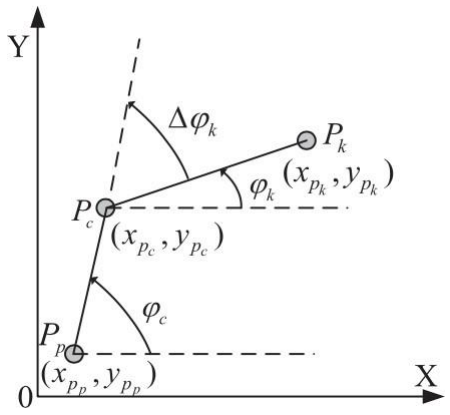


Figure 3. Path Selection Angles Reprinted from [13]

$$y_j = 1 - \frac{\Delta\psi_j}{\pi} \quad \textbf{(9)}$$

Where $\Delta\psi \in [0, \pi]$ is the absolute angle change between the current and the next moving direction. when the robot goes straight $\Delta\psi = 0$, which means the y_j is get maximum. This increases the probability of being selected for the next position.

Whenever the neural activity at the current robot location is smaller than the largest neural activity of its neighboring locations, the robot starts to move to its next location.

2.1.4. Alternative path selection strategy

- Proposed in reference [16]
- Utilizes a planning window as shown in **Figure 5**.

- Selects a list of positions from this window a shown in **Figure 4**, in a hierarchical order based on the status of each surrounding neuron.
    - Left > bottom > top > right > right-bottom > right-top.

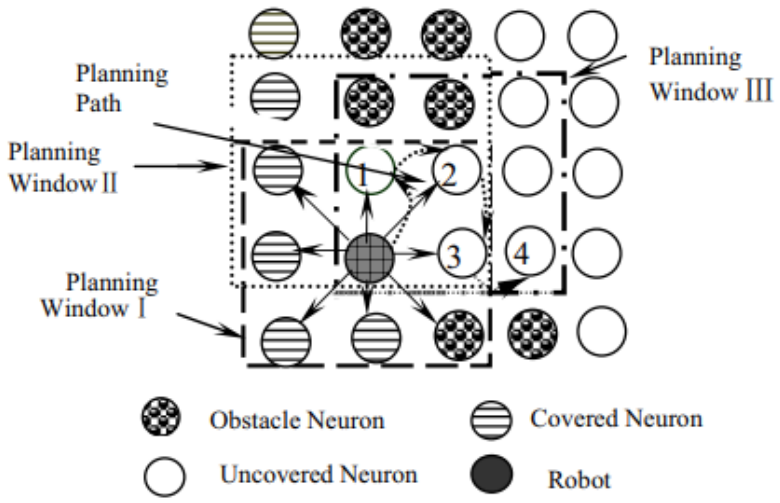


Figure 4. Grid Planning Windows Reprinted from [16]

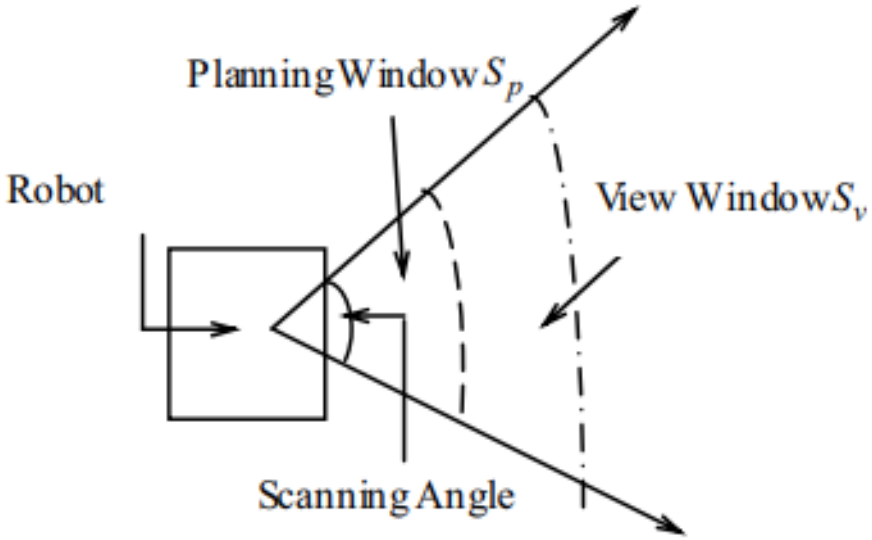


Figure 5. Planning & Robot View Window Reprinted from [16]

2.1.5. Implementation

The modern algorithm consists of three phases:

- The initialization phase: The initialization algorithm aims to initialize the starting position of the robot and set all the neural activities as zeros.
- The map-building phase: This phase primarily relies on the onboard sensors of the robot, which have a limited reading distance. In an unknown environment, a map of the surrounding area can be dynamically constructed using the available sensor data.
- Complete coverage navigation: The unclean areas globally attract the robot in the whole state space through neural activity propagation, while the obstacles have only local effects in a small region to avoid collisions.

2.1.6. Multiple robots (Discrete Programming)

Two assumptions:

- All the robots share environment information.
- Each robot considers the other robots as moving obstacles.

Algorithm Flow

1. Set up the grid map to represent the working environment.
2. Construct the neural networks of the grid map, each robot corresponds to a neural network.
3. The other robots and the obstacles can be seen as the inhibitory input to avoid the collision between the robots and the obstacles.

Centralized Programming

Assumptions:

- Each robot is only responsible for its own assigned area.
- The workspace needs to be divided by the decomposition Method into several nonoverlapping sub-regions.

3. RESEARCH OBJECTIVES

Most of the related work conducted using the proposed algorithm has been simulation-based. However, the physical implementation of the Neural Network Complete Coverage Path Planning algorithm as shown in **Figure 6** was demonstrated using a multi-robotic system [37]. The system consisted of three iRobot Create platforms, which are two-wheeled differential drives equipped with wheel encoders. The robots were fitted with three Devantech SRF05 ultrasonic sensors and a Microsoft Kinect sensor. A mounting platform was installed on the robot, housing an HP PC for computational processing. Odometry data was sent from the PC to the motor wheels for robot control, and data transmission occurred via USB-supporting serial port communication. The system was tested using offline CCPP. Performance metrics such as the time taken to complete the coverage task, power consumption by each robot, individual distances traveled, and number of turns for each robot were measured. The data collected for single and multi-robot testing is shown in **Figure 7** & **Figure 8**.

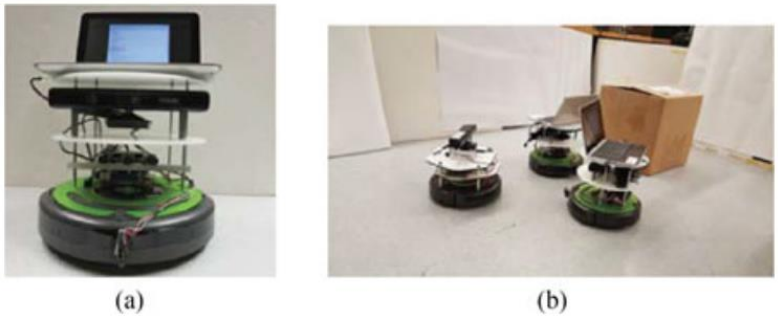


Fig. 7. Test scenario in cleaning workspace. (a) Built robot; (b) robots located in positions E and F, with Robot 3 located in Potion G.

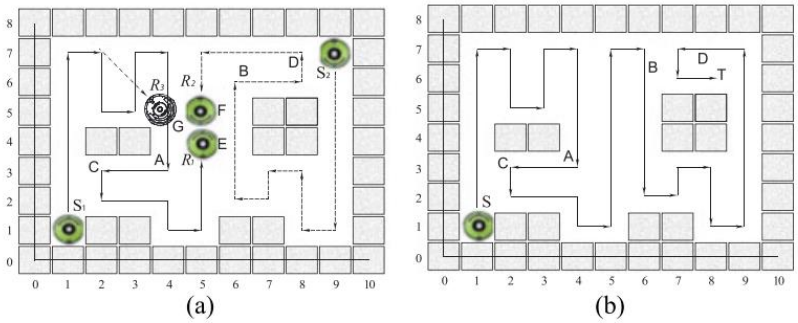


Fig. 8. Test scenario in a CAC navigation workspace. (a) Multirobot system; (b) single robot.

Figure 6. Physical Implementation Reprinted from [37]

Model	Overlap	Turns	Length
Oh et al. 's model	5	26	78.02
Proposed model	0	30	70.83

Figure 7. Single Robot Data Reprinted from [37]

Although the current method used to test the physical system is serviceable, it may not be suitable for industrial applications. The implemented setup is not ideal, as a commercial robot built for complete area coverage will not typically include a mobile PC attached to it that

executes the NN CCPP algorithm and controls the motion of the robot. A more feasible approach for industrial applications would be to program a single-board computer (SBC) to execute the NN CCPP algorithm. These SBCs are purpose-built for robotic applications and have lower computational power than mobile PCs, but are cost-effective and consume less power for efficient performance. Examples of such SBCs include Raspberry Pi, BeagleBone, and TI TDA4VM.

Measurement	Multirobot System		Two Robots	Single Robot
	Robot 1	Robot 2		
Time (min)	N/A	N/A	2.1	4.3
Power (W)	2.97	2.75	5.72	5.72
Distance	27	25	52	52
Turns	12	10	22	23

Figure 8. Multi-robot Data Reprinted from [37]

The objective of this research is to describe a practical attempt at complete coverage path planning on an industrial-ready robotic platform using the proposed Neural Network algorithm [9] in combination with the heuristic path selection strategy [16]. The Neural Network algorithm will be adapted to develop a system that allows a single robot to achieve complete coverage path planning with higher coverage completeness, lower path repetition rate, and less path execution time. This functionality is also considered for multi-robot collaboration. Although attempts were made to implement the navigation on a physical single and multi-robotic system, the expected output for this research was limited to real-world simulations using *Gazebo World* and *Robot Operating System (ROS)*.

4. METHODOLOGY

4.1. MATLAB Simulation work

4.1.1. Obtain occupancy grid map from MATLAB navigation toolbox.

In this research thesis, the initial step towards implementing the path planning algorithm in the simulation was to acquire a simulated environment acting as the navigation workspace. To this end, the MATLAB *occupancyMap [1]* function was utilized to create a 2-D occupancy grid map object. The *occupancyMap* creates a 2-D occupancy grid map object. Each cell in the occupancy grid has a value representing the probability of the occupancy of that cell. Values close to 1 represent a high probability that the cell contains an obstacle. Values close to 0 represent a high probability that the cell is not occupied and obstacle free. It should be noted that the *occupancyMap* function is distinct from *binaryOccupancyMap*, [14] which represents cells using only 0 and 1 values denoting free and obstacle spaces, respectively. For this MATLAB simulation environment, only the *occupancyMap* was used. This function offers several features for interacting with the represented map, including the ability to manipulate the occupancy map and create a customized occupancy map. The primary functions of the *occupancyMap* used in this study are listed in **Error! Reference source not found.** below, while additional object functions are outlined in reference [2].

Table 1. Occupancy Map Functions

checkOccupancy	Check if locations are free or occupied
getOccupancy	Get occupancy probability of locations
grid2world	Convert grid indices to world coordinates
inflate	Inflate each occupied location
occupancyMatrix	Convert occupancy map to matrix
setOccupancy	Set occupancy probability of locations
show	Display 2-D occupancy map
updateOccupancy	Update occupancy probability at locations
world2grid	Convert world coordinates to grid indices

4.1.2. Define workspace boundaries of the occupancy grid map.

The code section below shows the creation and population of an occupancy grid map with defined boundaries. It should be emphasized that the occupancy map comprises two primary frames of reference: XY coordinates and row-column (ij) grid indices. It is essential to identify the reference frame being used when working with the *occupancyMap* object since the behavior differs when attempting to manipulate or interact with the occupancy map. For instance, point (1,1) in the XY coordinate frame is located at the bottom left corner of the occupancy map, whereas in the ij grid index frame, it is situated at the top left corner. By default, the dimensions of the *occupancyMap* objects are created using the XY coordinate frame. However, to specify dimensions using row-column indices, the "grid" keyword is utilized in the object function. Additional information concerning the creation of an occupancyMap object can be found in reference [3]. For this study, the row-column grid indices served as the reference frame since a grid mapping approach was being implemented.

```
clear
clc

l = 10; %set length (# of rows) of grid map
w = 10; %set width (# of columns) of grid map
%res = 2;  %define the number of cells per meter for grid map ( sets resolution)

map = occupancyMap(l,w,2,"grid"); %generate grid map dimensions
n = zeros(map.GridSize(1), map.GridSize(2)); %generate array to store neural activity values

%set map boundaries
for x = 1:1:map.GridSize(1)
    setOccupancy(map,[1,x],1,"grid")
    setOccupancy(map,[x,1],1,"grid")
    setOccupancy(map,[map.GridSize(1),x],1,"grid")
    setOccupancy(map,[x,map.GridSize(2)],1,"grid")

    pause(0.005)
    show(map)
end
```

Initially, an *occupancyMap* is generated with specified row and column values. Optionally, the resolution of the occupancy map can also be specified. Defining a resolution allows for the grid/cell size of the occupancy map to be determined. The default resolution is set to 1 cell per meter. For this particular code section, a 10-row by 10-column occupancy map with a default resolution of 1 is used, resulting in a 10m x 10m occupancy map. To establish defined boundaries, the *setOccupancy* function is employed within a for-loop to assign all values in rows 1 & 10, and columns 1 & 10, to occupancy values of 1. Subsequently, an array variable of zeros, which is the same size as the occupancy grid map created, is generated. This array variable functions as the neural activity grid that represents the overall status of the map. The changes in values of this array will ultimately represent the dynamic landscape of the workspace as the robot navigates through the map. An additional array variable *mat* is created, which contains the actual values of each grid in the occupancy map. The code section's anticipated output is a visualization of the generated boundaries, as shown in **Figure *9*** below.

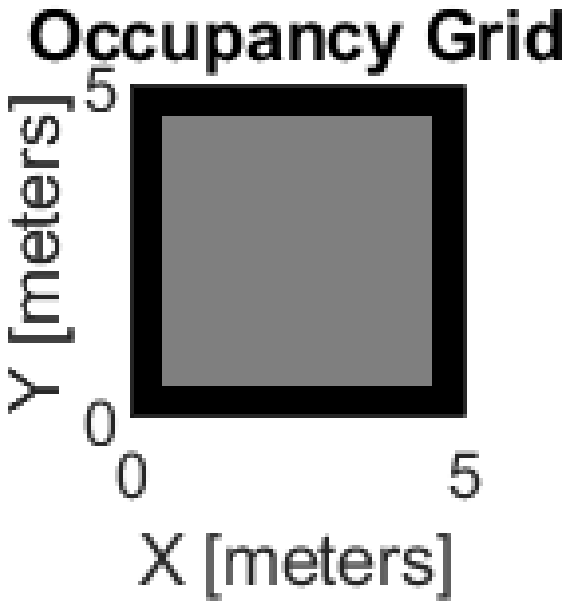


Figure 9. Occupancy Grid Workspace

4.1.3. Initiate a single robotic agent at the start coordinate.

The created occupancy map is shown on a figure which can be plotted on. To simulate the visualization of the robot in the occupancy map, a triangular marker is plotted at a specified grid location. This grid location is used as the robot start location. For this work, the grid index start location of (2,2) was used. This initializes the robot start location at the top left corner of the occupancy grid inside the boundary. The coordinate of the start location was chosen arbitrarily and can be varied for testing purposes. As can be observed in the code snippet below, multiple string variables for triangular markers that point in different directions were created for better visualization of the robot heading in the occupancy map. The coordinate of the start location is also assigned to a variable (pc) for use later in the coverage navigation section of this program.

```
  mat = occupancyMatrix(map); %an array containing the actual values of each grid in the
occupancy map
  %initialize starting location
  up_dir = '^r'; %indicator for robot heading direction on map
  down_dir = 'vr'; %indicator for robot heading direction on map
  left_dir = '<r';
```

```
right_dir = '>r';
hold on
startloc = [2,2]; %starting grid location
xy = grid2world(map,[startloc(1),startloc(2)]); %convert starting grid location to xy coordinates
plot(xy(1), xy(2), down_dir) %plot starting location xy coordinates on map
```

Once the map and its boundaries have been generated and the robot start location has been initialized, the navigation in the simulation can begin. This specific simulation code waits for a user prompt before it can begin running its coverage path planning. In this case, it displays a figure and waits for the user to press the spacebar key before the coverage path planning navigation can begin.

```
pc = startloc; %initialize starting grid location as current position
go = waitforbuttonpress;
```

4.1.4. Generate obstacle-free coverage path using Neural Network algorithm.

The subsequent code segment is responsible for the coverage path planning of the robot in simulation. The fundamental outline of this segment involves calculating the neural activity of each grid cell in the map and then utilizing these values to select the subsequent location to move to. Following each movement of the robot, the neural activity level of each grid cell in the map is then recalculated.

The following section of code pertains to the coverage path planning of a robot in a simulated environment. The fundamental procedure is comprised of calculating the neural activity of each grid cell within the map, and subsequently selecting the next location for movement based on these values. Following the robot's movement, the neural activity level of each grid cell is recalculated.

To derive the neural activity level of each grid cell, the setOccupancy function is utilized to assess the status of the corresponding grid cell on the occupancy map. This function returns one of three values: -1, 0, or 1, signifying an uncovered grid cell, a covered grid cell, and an obstacle grid cell, respectively. The resulting value of the setOccupancy function informs the updating of I values, which assume a value of 100, 0, and -100 to indicate an uncovered, covered, and obstacle grid cell, respectively. These I values are then subjected to the ReLu function, whereby they are evaluated by comparing them to zero and choosing the maximum value between the two. The I values undergo this evaluation twice, once for $[I]^+$ and once for $[I]^-$. The ReLu function used for $[I]^+$ finds the maximum value of I versus zero, while the one used for $[I]^-$ finds the maximum value of negative I versus zero.

```
while go == 1
%for go = 1:500
%Building neural network of map
for j = 1:1:map.GridSize(2) %{Reference; j is cols (goes left to right)}
    for i = 1:1:map.GridSize(1)  %{Reference; i is rows (goes up and down)}
        ab = grid2world(map, [i,j]); %current grid position for neural activity calculation
        status = checkOccupancy(map, [ab(1),ab(2)]); %check status of current grid position
        %convert grid values to 'I' values
        if status == -1
            I = 100;
        elseif status == 0
            I = 0;
        else
            I = -100;
        end
        I_plus = max([I 0]);
        I_neg = max([-1*I 0]);
```

In addition to the neural activity level of the current grid cell, the neural activity level of neighboring neurons must also be evaluated, as specified by the neural network algorithm in the accompanying journal article [9]. This evaluation is conducted using the ReLu function, with the neural activity value of the current grid cell serving as the threshold. While eight neighbors are

expected in an ideal instance, a try-catch block is employed for each direction due to potential index out-of-bounds errors, resulting from some grid cells lacking all eight neighbors. To circumvent this, a value of zero is assigned to the variable intended to store the returned value from the ReLu function evaluation when such errors occur. Additionally, distances from the current grid cell to its nearest neighbors are calculated by employing the custom MATLAB function *norm* [15], with grid index coordinates first converted to XY coordinates using the *grid2world* function.

```
        %check neural activity level of neighboring neurons, evaluate using ReLu and calculate the
euclidean distance
        try
            North = max([n(i-1, j) 0]); %North = max([n(i, j-1) 0]);
            dNorth = 0.7/((norm(ab - grid2world(map, [i-1, j])))); %dNorth = norm(ab -
grid2world(map, [i, j-1]));
        catch
            North = 0;
            dNorth = 0;
        end

        try
            South = max([n(i+1, j) 0]); %South = max([n(i, j+1) 0]);
            dSouth = 0.7/(norm(ab - grid2world(map, [i+1, j]))); %dSouth = norm(ab -
grid2world(map, [i, j+1]));
        catch
            South = 0;
            dSouth = 0;
        end

        try
            West = max([n(i, j-1) 0]); %West = max([n(i-1, j) 0]);
            dWest = 0.7/(norm(ab - grid2world(map, [i, j-1]))); %dWest = norm(ab -
grid2world(map, [i-1, j]));
        catch
            West = 0;
            dWest = 0;
        end

        try
            East = max([n(i, j+1) 0]); %East = max([n(i+1, j) 0]);
            dEast = 0.7/(norm(ab - grid2world(map, [i, j+1]))); %dEast = norm(ab - grid2world(map,
[i+1, j]));
```

```
        catch
            East = 0;
            dEast = 0;
        end

        try
            NW = max([n(i-1, j-1) 0]); %NW = max([n(i-1, j-1) 0]);
            dNW = 0.7/(norm(ab - grid2world(map, [i-1, j-1]))); %dNW = norm(ab - grid2world(map, [i-1, j-1]));
        catch
            NW = 0;
            dNW = 0;
        end

        try
            NE = max([n(i-1, j+1) 0]); %NE = max([n(i+1, j-1) 0]);
            dNE = 0.7/(norm(ab - grid2world(map, [i-1, j+1]))); %dNE = norm(ab - grid2world(map, [i+1, j-1]));
        catch
            NE = 0;
            dNE = 0;
        end

        try
            SW = max([n(i+1, j-1) 0]); %SW = max([n(i-1, j+1) 0]);
            dSW = 0.7/(norm(ab - grid2world(map, [i+1, j-1]))); %dSW = norm(ab - grid2world(map, [i-1, j+1]));
        catch
            SW = 0;
            dSW = 0;
        end

        try
            SE = max([n(i+1, j+1) 0]); %SE = max([n(i+1, j+1) 0]);
            dSE = 0.7/(norm(ab - grid2world(map, [i+1, j+1]))); %dSE = norm(ab - grid2world(map, [i+1, j+1]));
        catch
            SE = 0;
            dSE = 0;
        end
```

The neural activity values for all neighboring neurons are stored in an array *xplus*, as are the Euclidean distances, which are stored in an array *WIJ*. These values are then summed together using MATLAB matrix operations, with the resulting summation stored in a variable *weight*, which is subsequently employed in the neural network equation. To calculate the change in neural

activity values, the *A, B, and D* variables specified by the journal article are set, following which the MATLAB ODE solver, *ode45,* is employed. To solve the ordinary differential equation, an initial value is specified and passed into the MATLAB ODE solver. In this instance, the current neural activity value in the relevant grid cell is utilized. The output of the ode45 function is an array of values, with the final value in the array representing the new neural activity value for that grid cell. This value is then stored in the corresponding grid index value of the array variable *n*. This calculation process is repeated in a for loop for all grid cells in the occupancy map, with the expected neural activity values after each calculation being 0.5, 0, and 0.5, corresponding to an obstacle grid cell, a covered grid cell, and an uncovered grid cell, respectively. The values in the *n* array variable are utilized in subsequent evaluations for the path selection algorithm.

```
        %store evaluated neural activity values for neighboring neurons (8x1 matrix/array)
        xplus = [North, South, West, East, NW, NE, SW, SE];

        %store euclidean distances (1x8 matrix/array)
        wij = [dNorth; dSouth; dWest; dEast; dNW; dNE; dSW; dSE];

        %calculate the weight
        weight = xplus * wij; % Matrix operation calculating for the weight (order of operation
 should not be changed)

        %set variables
        A = 80;
        B = 1;
        D = 1;
        %define & evaluate equations using ODE solver
        eqn = @(t,Xi) ((-A*Xi) + (B-Xi)*(I_plus + weight) - (D+Xi)*I_neg);
        [t,Xi] = ode45(eqn, [0:1], n(i,j));
        n(i,j) = Xi(end);

  %         if Xi(end) < 0
  %             n(i,j) = -0.6;%min(Xi);
  %         elseif Xi(end) > 0.5
  %             n(i,j) = 0.6;%max(Xi);
  %         elseif (Xi(end) > 0) && (Xi(end) < 0.5)
  %             n(i,j) = 0;%min(Xi);
  %         end
     end
  end
```

Before delving into the path selection algorithm of the program, it is imperative to validate whether there exist any neural activity values that are greater than or equal to 0.5. Such values serve as evidence that certain areas on the neural activity grid utilized to represent the map remain uncovered. This check functions as a stopping criterion for the program, with program execution halting once there are no longer any uncovered spaces.

```
if sum(n>=0.5, 'all') == 0
    go = 0;
    break
end

%update the status of the current grid if the map/environment is not completely covered
setOccupancy(map,pc,0,'grid')
pause(0.005)
show(map)
```

The path selection algorithm utilizes a heuristic approach as detailed in the journal article cited in reference [16]. Initially, the neural activity values for all feasible subsequent locations that the robot can traverse are identified and recorded in an array. It is crucial to note that as a heuristic approach, the order of values stored in the *pn* array should remain unchanged. The values contained within this array are assessed to determine the status of all feasible next possible locations.

```
%--------------------PATH SELECTION ALGORITHM-------------------------------%
%Heuristic sequence
%left -> bottom -> top -> right -> right bottom -> right top


%Find all possible next positions
left = n(pc(1), pc(2)-1); %left as the next position
bottom = n(pc(1)+1, pc(2)); %bottom as the next position
top = n(pc(1)-1, pc(2)); %top as the next position
right = n(pc(1), pc(2)+1); %right as the next position
right_bottom = n(pc(1)+1, pc(2)+1); %right bottom as the next position
right_top = n(pc(1)-1, pc(2)+1); %right top as the next position
```

```
pn =[left, bottom, top, right, right_bottom, right_top]; %don't change order of this array

%check the status of all possible next locations
for value = 1:1:length(pn)
    if pn(value) < 0
        pn(value) = -1; %indicate cell is obstacle
    elseif pn(value) > 0.5
        pn(value) = 1; %indicate cell is uncovered
    elseif (pn(value) > 0) && (pn(value) < 0.5)
        pn(value) = 0; %indicate cell is covered
    end
end
```

After evaluating the status of all possible next locations, the code checks for a potential deadlock event, which occurs when all uncovered spaces in the array of possible next locations have been explored. In the event of a deadlock, the program assesses the status of all neighboring grid cells and selects the neighboring grid cell with the highest value. If multiple neighboring grid cells have the same maximum value, the program selects the grid cell that appears first in the array list stored in the variable *search* as the next location. The program hierarchy of paths is demonstrated in the switch case located in the code section below.

```
if sum(pn==1) == 0 %check to for a deadlock event
    %search for the uncovered location in closest neighbors using neural propagation
    kN = n(pc(1)-1, pc(2)) ;
    kS = n(pc(1)+1, pc(2)) ;
    kW = n(pc(1), pc(2)-1) ;
    kE = n(pc(1), pc(2)+1) ;
    kNW = n(pc(1)-1, pc(2)-1) ;
    kNE = n(pc(1)-1, pc(2)+1) ;
    kSW = n(pc(1)+1, pc(2)-1) ;
    kSE = n(pc(1)+1, pc(2)+1) ;
    search = [kN,kNE,kE,kSE,kS,kSW,kW,kNW];
    switch max(search)
        case search(1)
            pc = [pc(1)-1, pc(2)];
            gpc = grid2world(map, pc);
            %disp('north is next')
            plot(gpc(1), gpc(2),up_dir)
        case search(2)
            pc = [pc(1)-1, pc(2)+1];
```

```
            gpc = grid2world(map, pc);
            %disp('ne is next')
            plot(gpc(1), gpc(2),up_dir)
        case search(3)
            pc = [pc(1), pc(2)+1];
            gpc = grid2world(map, pc);
            %disp('east is next')
            plot(gpc(1), gpc(2),right_dir)
        case search(4)
            pc = [pc(1)+1, pc(2)+1];
            gpc = grid2world(map, pc);
            %disp('se is next')
            plot(gpc(1), gpc(2),down_dir)
        case search(5)
            pc = [pc(1)+1, pc(2)];
            gpc = grid2world(map, pc);
            %disp('south is next')
            plot(gpc(1), gpc(2),down_dir)
        case search(6)
            pc = [pc(1)+1, pc(2)-1];
            gpc = grid2world(map, pc);
            %disp('sw is next')
            plot(gpc(1), gpc(2),down_dir)
        case search(7)
            pc = [pc(1), pc(2)-1];
            gpc = grid2world(map, pc);
            %disp('west is next')
            plot(gpc(1), gpc(2),left_dir)
        case search(8)
            pc = [pc(1)-1, pc(2)-1];
            gpc = grid2world(map, pc);
            %disp('nw is next')
            plot(gpc(1), gpc(2),up_dir)
    end
```

If a deadlock event is not detected, the program reverts to its typical heuristic path selection method, where the next location is chosen based on the hierarchy of paths using a switch statement, as demonstrated in the section below.

```
     %once the uncovered location is found revert to the heuristic path selection
 else %continue with heuristic path selection
    switch max(pn)
        case pn(1)
            pc = [pc(1), pc(2)-1];
            gpc = grid2world(map, pc);
            %disp('left is next')
            plot(gpc(1), gpc(2),left_dir)
```

```
        case pn(2)
            pc = [pc(1)+1, pc(2)];
            gpc = grid2world(map, pc);
            %disp('bottom is next')
            plot(gpc(1), gpc(2),down_dir)
        case pn(3)
            pc = [pc(1)-1, pc(2)];
            gpc = grid2world(map, pc);
            %disp('top is next')
            plot(gpc(1), gpc(2),up_dir)
        case pn(4)
            pc = [pc(1), pc(2)+1];
            gpc = grid2world(map, pc);
            %disp('right is next')
            plot(gpc(1), gpc(2),right_dir)
        case pn(5)
            pc = [pc(1)+1, pc(2)+1];
            gpc = grid2world(map, pc);
            %disp('right bottom is next')
            plot(gpc(1), gpc(2),down_dir)
        case pn(6)
            pc = [pc(1)-1, pc(2)+1];
            gpc = grid2world(map, pc);
            %disp('right top is next')
            plot(gpc(1), gpc(2),up_dir)
    end
end
```

Finally, the program updates the array variable *mat* containing the actual values of each grid in the occupancy map.

```
mat = occupancyMatrix(map); %update grid values of the map
%mesh(n)
% Stop condition
end
```

The aforementioned procedure delineates the fundamental steps for simulating robot navigation through an occupancy map for complete coverage path planning. The said process is iterated within a while loop until all cells in the neural activity array variable *n* are covered.

4.1.5. Investigate coverage path planning with static obstacles.

To evaluate the performance of the path planning algorithm in various environments with varying static obstacles, different variations of the code were created for simulation work. These variations are referenced in Appendix A. For instance, in appendix A-1, instead of manually generating the occupancy map during program execution, a pre-made map is read from an Excel file and used as the navigation workspace for coverage. These pre-made maps were used to test the algorithm's performance in different environments with varying static obstacles. Images of these different maps are presented in Appendix B.

In appendix A-2, a modified calculation process is presented to simulate an online path-planning approach. In this approach, the robot is assumed to have an unknown representation of the navigation environment and only calculates the neural activity values for its closest neighbors to choose the next path. This allows for a more realistic simulation of a robot navigating in an unknown environment, where it can only perceive its immediate surroundings.

In appendix A-3, a modified version of the program is presented that incorporates the next position decision formula as outlined in the *Model Algorithm* section (2.1.2).

4.2. ROS simulation

The initial evaluation of the software developed for the physical robot involved conducting experiments in ROS simulation. The rationale for utilizing ROS simulation was to closely emulate the expected performance of the actual system. The successful navigation outcomes attained during ROS simulation are likely indicative of the corresponding performance of the physical system.

The simulated robot was initially designed using Autodesk Fusion 360, CAD software. The resulting CAD model was subsequently exported in a unified robot description format (URDF) utilizing the *syuntoku14/fusion2urdf* plugin. Further details regarding this plugin can be obtained from references [5][6].

In the present study, the CAD model was designed on a Windows machine. However, to facilitate ROS simulations with the CAD model, the URDF files are required to transfer to a Linux machine. The FileZilla application was utilized to accomplish this task. A comprehensive guide to using FileZilla for transferring files and folders from a Windows machine to a Linux machine is available in reference [4].

After transferring the URDF files to the Linux machine, a catkin workspace was established for ROS development purposes. The catkin workspace functions as a directory in which various ROS packages are stored. Within the catkin workspace, a new folder was created, and the URDF files were subsequently transferred into it.

Following the establishment of the catkin workspace and the transfer of URDF files into the workspace, a thorough ROS development was carried out, utilizing the navigation stack to configure the robot for navigation in simulation. Essential information on the fundamentals of ROS navigation configuration can be accessed through references [17][18].

The directory containing all the packages employed to establish the ROS navigation environment in simulation for this study can be accessed through the GitHub repository identified in reference [19].

The following is a summary of the steps undertaken to establish navigation in the ROS simulation environment:

4.2.1. Construct a generic environment in the Gazebo world.

A self-contained area was built to conform to the requirement of the navigation program. Non-compliance with this requirement could lead to suboptimal performance, such as the navigation never terminating. Instructions on how to accomplish this were sourced from the references [20][21]

4.2.2. Launch and control simulated robot in Gazebo world environment.

This robot was controlled in one of two ways. The first involved publishing a goal pose (in quaternion coordinates) to a topic ('/movebase_simple_goal') subscribed to by the ROS navigation package move_base, which attempts to move the robot to the goal. This navigation package also performs live obstacle avoidance by utilizing real-time laser scan data from the onboard range sensor (lidar) that is configured on the robot and published to a topic ('/scan' topic). The robot can also be controlled by manually publishing driving commands in the form of linear and angular velocities to the topic ('/cmd_vel'), to which the Gazebo world simulated robot is subscribed. The robot publishes its odometry data for localization to the '/odom' topic, which is generated from the world coordinates in its simulated environment for the simulated robot. However, this odometry data is highly prone to error due to drift.

4.2.3. Improving localization.

To enhance localization, the *amcl* [22]localization package is used. This package attempts to match the laser scans of the environment to a published map and utilizes this measurement information along with the odometry data to provide more accurate localization.

- It is important to note that for ROS navigation, localization, and target goal pose are expressed in meters and the quaternion coordinates frame. Therefore, it may be necessary to perform necessary conversions from this unit and/or coordinate frame.

4.2.4. Generate a map using the gmapping package.

To generate a map of the Gazebo test environment, the *gmapping* [23] package is utilized. The map size needs to be limited to the dimensions of the environment to manage the size of the map data. Failure to do so will result in a map much larger than the mapped area, leading to heavy computational burdens that affect the performance of the navigation program.

4.2.5. Save the map.

The map generated is saved using the *map_saver* feature from the *map_server* [24] package to the *maps* folder in the catkin workspace.

4.2.6. Publish map to ROS topic.

This saved map can be published to a ROS topic using the *map_server* package (publish lidar map to ROS '/map' topic).

It is essential to comprehend all the topics in the ROS network for both ROS simulation and real robot navigation. Terminal commands like *rostopic list*, *rostopic info*, and *rostopic echo* assist in gathering information on these topics. More information about these commands can be found in the ROS wiki [25].

The data published on these topics can be visualized using the RViz [26] tool. RViz is a 3D visualization tool for ROS (Robot Operating System) that allows users to visualize and interact with various types of data in a 3D environment. It is designed to provide a graphical user interface

(GUI) for displaying sensor data, robot models, and other types of visualizations that are commonly used in robotics and autonomous systems.

RViz provides a variety of features, such as displaying 3D sensor data (e.g., point clouds), robot models, and various visualizations (e.g., arrows, spheres, and lines) that can be used to represent different types of data. It also allows users to interact with the visualizations and modify the way data is displayed. RViz can be used for debugging, testing, and developing algorithms for robots, and is commonly used in robotics research and development.

Foxglove Studio [27] is an alternative 3D visualization tool to RViz that can be used for ROS development.

Once the navigation environment in ROS simulation was established, MATLAB code was generated to read published maps from the ROS topic and convert the data to an occupancy grid object. The following code snippet achieves this function:

```
clear
prompt = "ipaddress: ";
ipaddress = input(prompt, "s");
rosinit(ipaddress, 11311)


sub = rossubscriber("/map",DataFormat="struct");
msg = receive(sub);
map = rosReadOccupancyGrid(msg);
occupancyMapObj = rosReadOccupancyGrid(msg);
show(occupancyMapObj)
```

After reading the published map into a MATLAB occupancy grid, navigation in the ROS simulation environment can begin. The navigation code used is similar to the program developed and detailed in the MATLAB simulation section. However, modifications were made to adapt the MATLAB simulation program to the ROS simulation environment due to the resolution of the

map published to ROS. Unlike in the MATLAB simulation where the navigation algorithm assumes that the robot diameter is the size of an individual grid cell and moves the robot one grid cell at a time, this would not be the case for the ROS simulated and real robot when navigating using the published LIDAR map. The published LIDAR map has a resolution of 50, which equates to a grid cell size of 2cm x 2cm. The ROS simulated and real robot have a maximum diameter of 50cm, and navigating one grid cell at a time would cause micro-movements in robot navigation.

To address this issue, the robot's movement on the occupancy map was scaled up to the size of the robot. It was reasoned that the robot would cover approximately 25 cells on the occupancy map. However, a reference point in this region of grid cells was needed to identify the robot's current position. In this case, the reference point was chosen to be the approximate center of mass (COM) of the scuttle robot. The robot's movement was defined to be driving from the COM to a distance at least that of the robot's diameter in all possible next locations/directions. The movement of this reference point was used to perform the necessary calculations that define the neural activity propagation and creates the dynamic landscape.

The scaling concept is illustrated in **Figure 10** & **Figure 11**. The blue grids circled in red represents the reference point of the robot's current location. The black borders inside the red outlines indicate the size of the robot. The light blue grid cell denotes the top left corner of the robot. The top left corner is of particular importance, as we will explain in the subsequent sections. The red grid cells represent the reference points (COM) of all the possible next locations.

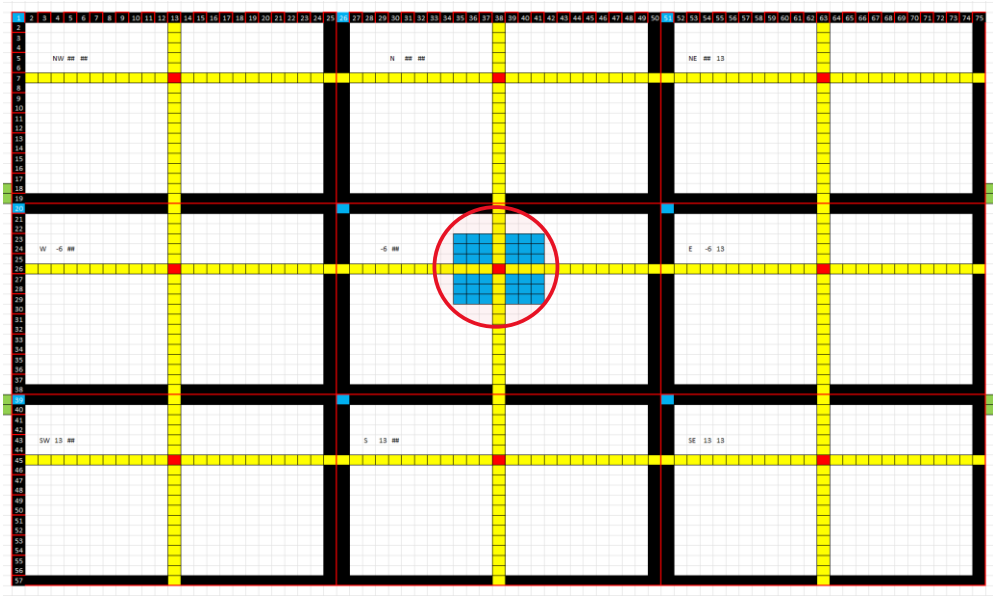


Figure 10. Robot Scaling

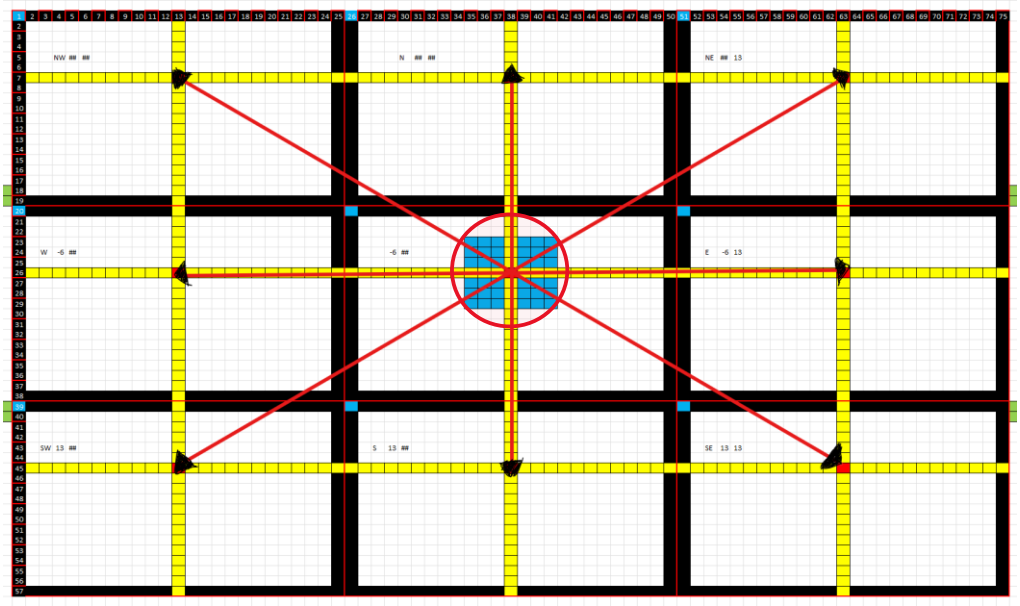


Figure 11. Robot Scaling Movement Directions

A MATLAB simulation program for ROS navigation was developed based on this framework, using the published map as the foundation for the real robot's movement through the occupancy

grid. The first step was to convert the published map into an occupancy matrix, which was then saved to an Excel file for storage and repeated use in situations where the ROS network is not configured and the map is not published. The stored map matrix can be read from the Excel file and converted into an occupancy grid. An array variable of zeros, the size of the occupancy grid map is initialized. The starting location is also initialized and plotted on the occupancy grid figure.

```
clear
clc

map = occupancyMap(readmatrix("map builder.xlsx", "Sheet","100 park"));
show(map)

n = zeros(map.GridSize(1), map.GridSize(2)); %generate array to store neural activity values

%initialize starting location
up_dir = '^r'; %indicator for robot heading direction on map
down_dir = 'vr'; %indicator for robot heading direction on map
left_dir = '<r';
right_dir = '>r';
hold on
pc = [15,18]; %starting grid location
xy = grid2world(map,pc); %convert starting grid location to xy coordinates
plot(xy(1), xy(2), down_dir) %plot starting location xy coordinates on map
```

To ensure the safe navigation of the robot, it is necessary to assess the status of each grid cell within the 25-cell area that the robot is assumed to cover. To accomplish this, the grid cell corresponding to the top-left corner of the robot's designated area is identified as the starting point for a for loop that parses through all cells representing the robot's field of view. If any obstacle spaces are identified within the region, the entire area is designated as an obstacle region. This approach is essential to preclude the possibility of the navigation algorithm directing the robot toward a static obstacle represented on the occupancy map. This check is conducted for all neighboring regions that comprise the next feasible locations.

```
go = waitforbuttonpress; %CHANGEE!!!
tic
while go == 1


%Building neural network of map


%find top left corner points in all directions
TL = [pc(1)-6, pc(2)-12];
NW_TL = [pc(1)-25, pc(2)-37];
N_TL = [pc(1)-25, pc(2)-12];
NE_TL = [pc(1)-25, pc(2)+13];
E_TL = [pc(1)-6, pc(2)+13];
SE_TL = [pc(1)+13, pc(2)+13];
S_TL = [pc(1)+13, pc(2)-12];
SW_TL = [pc(1)+13, pc(2)-37];
W_TL = [pc(1)-6, pc(2)-37];




%update status of current grid if map/environment is not completely covered
for tl_j = TL(2):1:TL(2)+24
   for tl_i = TL(1):1:TL(1)+18
      setOccupancy(map,[tl_i, tl_j],0.5,'grid')
   end
end
pause(0.005)
show(map)


%Find neural activity level and of all possible next positions
[kW,n] = directionStatus(W_TL,map,n);
[kN,n] = directionStatus(N_TL,map,n);
[Kp,n] = directionStatus(TL,map,n);
[kS,n] = directionStatus(S_TL,map,n);
[kNE,n] = directionStatus(NE_TL,map,n);
[kE,n] = directionStatus(E_TL,map,n);
[kSE,n] = directionStatus(SE_TL,map,n);
```

Following the acquisition of the status of all neighboring regions, the standard neural activity calculations are executed, with the reference point serving as the target location. The robot scaling approach employs the rolling window method, which is implemented through the code shown below. This code incorporates functions that enable the examination of the robot area within the various regions on the map, as well as the necessary calculations.

```
%--------------------PATH SELECTION ALGORITHM-------------------------------%
%Heuristic sequence
%left -> bottom -> top -> right -> right bottom -> right top




%check the status of all possible next locations
pn =[kW, kS, kN, kE, kSE, kNE]; %don't change order of this array


%check the status of all possible next locations
for value = 1:1:length(pn)
   if pn(value) < 0
      pn(value) = -1; %indicate cell is obstacle
   elseif pn(value) > 0.5
      pn(value) = 1; %indicate cell is uncovered
   elseif (pn(value) > 0) && (pn(value) < 0.5)
      pn(value) = 0; %indicate cell is covered
   end
end




%modify deadlock event
if sum(pn==1) == 0 %check to for deadlock event
   [kNW,n] = directionStatus(NW_TL,map,n);
   [kSW,n] = directionStatus(SW_TL,map,n);
   search = [kN,kNE,kE,kSE,kS,kSW,kW,kNW]; %search for uncovered location in closest neighbors using neural propagation
   switch max(search)
%        case search(1)
%           pc = moveNextDirection(TL,up_dir,map);
%           %disp('north is next')
       case search(1)
          pc = moveNextDirection(N_TL,up_dir,map);
          %disp('north is next')
       case search(2)
          pc = moveNextDirection(NE_TL,up_dir,map);
          %disp('ne is next')
       case search(3)
          pc = moveNextDirection(E_TL,right_dir,map);
          %disp('east is next')
       case search(4)
          pc = moveNextDirection(SE_TL,down_dir,map);
          %disp('se is next')
       case search(5)
          pc = moveNextDirection(S_TL,down_dir,map);
          %disp('south is next')
       case search(6)
          pc = moveNextDirection(SW_TL,down_dir,map);
          %disp('sw is next')
       case search(7)
```

```
            pc = moveNextDirection(W_TL,left_dir,map);
            %disp('west is next')
        case search(8)
            pc = moveNextDirection(NW_TL,up_dir,map);
            %disp('nw is next')
    end


     %once uncovered location is found revert back to heuristic path selection
else %continue on with heuristic path selection
    switch max(pn)
        case pn(1)
            pc = moveNextDirection(W_TL,left_dir,map);
            %disp('west is next')
        case pn(2)
            pc = moveNextDirection(S_TL,down_dir,map);
            %disp('south is next')
        case pn(3)
            pc = moveNextDirection(N_TL,up_dir,map);
            %disp('north is next')
        case pn(4)
            pc = moveNextDirection(E_TL,right_dir,map);
            %disp('east is next')
        case pn(5)
            pc = moveNextDirection(SE_TL,down_dir,map);
            %disp('se is next')
        case pn(6)
            pc = moveNextDirection(NE_TL,up_dir,map);
            %disp('ne is next')
    end
end


%mat = occupancyMatrix(map); %update grid values of the map


% Stop condition
%check if map/environment is completely covered and end operation
if sum(n>=0.5, 'all') == 0
    go = 0;
    break
end


end
toc
```

```
function [y,n] = directionStatus(x,map,n_mat)
%Use a try block to iterate through and calculate n-values for all cells equating to the size of the bot in a specified direction
    try
        n = n_mat;
```

```
    for j = x(2):1:x(2)+24 %{Reference; j is cols (goes left to right)}
        for i = x(1)-1:1:x(1)+18  %{Reference; i is rows (goes up and down)}
            status = checkOccupancy(map, [i,j], "grid"); %check status of current grid position
            %convert grid values to 'I' values
            if status == 1
                i = x(1) + 6;
                j = x(2) + 12;
                I = -100;
                calculateNeuralActivity
                return
            end
        end
    end


    %if no cell in the group is an obstacle, check the status of the cell where COG is located and use it's n-value for the specified direction
    i = x(1)+6;
    j = x(2)+12;
    %ab = grid2world(map, [i,j]);
    status = checkOccupancy(map, [i,j], "grid");
    switch status
        case -1 %uncovered
            I = 0;
            calculateNeuralActivity
            return
        case 0 %covered
            I = 100;
            calculateNeuralActivity
            return
    end

  catch %Catch "index out of bounds" errors and assign n-value of specified direction as an obstacle (-0.6)
    y = -0.6;
    return
  end %for the try/catch


  function calculateNeuralActivity


    I_plus = max([I 0]);
    I_neg = max([-1*I 0]);

    %check neural activity level of neighboring neurons, evaluate using ReLu and calculate the euclidean distance
    %input the correct displacement values for the COG of the neighboring direction
    [North, dNorth] = ReLu(-19,0); %n(i, j-1)
    [South, dSouth] = ReLu(19,0); %n(i+1, j)
    [West, dWest] = ReLu(0,-25); %n(i, j-1)
    [East, dEast] = ReLu(0,25); %n(i, j+1)
    [NW, dNW] = ReLu(-19,-25); %n(i-1, j-1)
    [NE, dNE] = ReLu(-19,25); %n(i-1, j+1)
```

```
        [SW, dSW] = ReLu(19,-25); %n(i+1, j-1)
        [SE, dSE] = ReLu(19,25); %n(i+1, j+1)


        %store evaluated neural activity values for neighboring neurons (8x1 matrix/array)
        xplus = [North, South, West, East, NW, NE, SW, SE];

        %store euclidean distances (1x8 matrix/array)
        wij = [dNorth; dSouth; dWest; dEast; dNW; dNE; dSW; dSE];

        %calculate the weight
        weight = xplus * wij; % Matrix operation calculating for the weight (order of operation should
not be changed)

        %set variables
        A = 80;
        B = 1;
        D = 1;
        %define & evaluate equations using ODE solver
        eqn = @(t,Xi) ((-A*Xi) + (B-Xi)*(I_plus + weight) - (D+Xi)*I_neg);
        [t,Xi] = ode45(eqn, [0:1], n(i,j));
        n(i,j) = Xi(end);
        y = n(i,j);
%         for m = x(2):1:x(2)+24 %{Reference; j is cols (goes left to right)}
%             for p = x(1)-1:1:x(1)+18  %{Reference; i is rows (goes up and down)}
%                 n(p,m) = y;
%             end
%         end



    end


    % function to check the neural activity level of neighboring neurons, evaluate using ReLu, and
calculate the euclidean distance
    function [dir, dir_dist] = ReLu(RowDisp,ColDisp)
        %input the correct displacement values for the COG of the neighboring direction
        ab = grid2world(map, [i,j]); %current grid position for neural activity calculation
        try
            dir = max([n(i + RowDisp, i + ColDisp) 0]);
            dir_dist = 0.7/((norm(ab - grid2world(map, [i + RowDisp, j + ColDisp]))));
%             disp(dir_dist)
        catch
            dir = 0;
            dir_dist = 0;
        end
    end

end %for the function
```

```
function pc = moveNextDirection(pTL,heading,map)
    pc = [pTL(1)+6, pTL(2)+12];
    gpc = grid2world(map, pc);
    plot(gpc(1), gpc(2),heading)

end
```

Upon configuring the robot for operation in ROS simulation and preparing the MATLAB program for ROS navigation, the next phase entailed the development of a MATLAB program that integrates both subsystems. To ensure optimal navigation in both simulated and physical environments, various features were integrated into the program. These features included an autonomous robot controller [28], which receives the current robot pose and next target location and iteratively computes the linear and angular velocity values which are published to the *cmd_vel* topic on the ROS network. This drives the robot toward the next position determined by the navigation algorithm. To simulate a more realistic navigation scenario, live obstacle avoidance was incorporated into the program, utilizing published lidar data on the *scan* topic to detect obstructions that impede or approach too closely to the robot during navigation. Recovery behaviors were added in situations where an obstacle was detected, or if the robot became physically stuck in its environment due to unforeseen circumstances. Additionally, a deadlock escape feature was implemented to enable the program to detect instances in which the robot's navigation becomes trapped. This feature detects when the robot is roaming for an extended period in a particular area that has already been covered while attempting to search for a path to an uncovered location on the map environment. A coverage visualization feature was also integrated into the program, allowing for live updates of a coverage map that is published to a ROS topic, and displayed through RViz to show the robot's coverage progress during navigation. Lastly, data collection was included for testing purposes. The program is shown in appendix A-4.

Several adaptations of the program in Appendix A-4 were developed for various purposes. One version of the program was designed to work in conjunction with map decomposition and task allocation software. This version of the program includes a feature that navigates the robot to a specified starting location before beginning the navigation process. The navigation is set to run continuously until all map sections have been covered, and coverage data is recorded for each section.

In addition, two other versions of the program were created for multi-robot navigation within the same map environment. However, the development of the multi-robot system is beyond the scope of this thesis and will not be detailed in this report. The codes for these programs can be found in the GitHub repository [38] but may require further development for reliable functionality and scalability.

4.3. Physical System Development

The proposed implementation for this study involves setting up a ROS network for at least two robots. The peer-to-peer communication capabilities of ROS enable seamless communication between all systems involved. The systems involved include Linux machines with ROS and a PC with MATLAB. The MATLAB software supports ROS and includes a ROS toolbox that allows for connectivity to ROS networks. By utilizing the ROS communication framework, driving commands can be published from MATLAB to specific topics that the robots are subscribed to. The topics that the robots will be listening to involve driving commands and these commands will be published through the ROS communication framework. By connecting to the ROS network, the MATLAB software can seamlessly communicate with the robots, allowing for the implementation of the proposed navigation system. The software utilized for the physical system is identical to that

of the ROS simulation. ROS serves as the conduit that enables the seamless transition from simulated work to a physical system.

The first step in developing the physical system involves acquiring the necessary equipment, starting with a differential drive robot. In this work, the two-wheeled differential drive robot Scuttle was utilized, which is commonly used in the Multidisciplinary Engineering Technology (MXET) course labs. Additional information about the Scuttle robot can be found on the scuttle.org website or by reviewing the MXET 300 lab course curriculum for the spring of 2023.

For the proposed setup, the Scuttle robot was equipped with a Raspberry Pi, a 12-volt battery pack, a 360-degree 2D RP LIDAR, two DC motors with attached wheels, and an L298N motor driver. It is worth noting that certain brackets and components of the robot were 3D printed using the 3D printers in the DMD lab. While the instructions on how to build the Scuttle differential drive robot will not be detailed in this paper, the necessary components and equipment utilized in the physical system setup have been outlined in Appendix C.

All hardware components were acquired through sourcing and purchase, and the robot was built from scratch to match the models utilized in the MXET 300 lab. A bill of materials detailing the necessary components will be provided in appendix C. The approximate cost for a single robot build is estimated to be around $500. To build the robots, access to a lab workshop with standard tools for hardware and electronics prototyping is necessary.

The next equipment utilized in this study was a Linux machine running Ubuntu version 20.04.5 LTS with ROS Noetic installed. To facilitate the setup of the ROS network, a router was acquired for convenience. The router was used to establish a local area network, allowing all

devices to communicate with each other effortlessly over ROS. All devices were connected to the Wi-Fi network broadcasted by the router. It is worth noting that a dedicated router is not required to connect all devices to a ROS network. Instead, all devices simply need to be within the same local area network (LAN) for a ROS network to be established. Further details about the router can be found in the bill of materials. In addition, an Xbox gamepad remote controller was acquired to facilitate manual control of the robot. This proved useful for tasks such as driving the robot to map a room or driving the robot to its starting location.

To ensure optimal performance, it is recommended to have access to a multimeter to check the battery voltage after prolonged operation. The nominal operating battery voltage range for the robot is between 10.5 volts to 12 volts, as the driving performance of the robot is significantly impacted below this range. A 12-volt adapter is utilized to charge the robot batteries. This adapter features a barrel connector that attaches to a custom-built cable.

It is also highly recommended to have an intermediate knowledge of the ROS navigation stack, computer-aided design (CAD), and Linux.

5. TESTING & DATA COLLECTION

This study involved conducting three types of tests to evaluate the performance of the coverage algorithm. The first test involved assessing coverage performance through MATLAB simulations under ideal conditions. A perfectly set up environment with an n-by-n symmetric map, where the robot could perfectly fit and cover each grid cell. The occupancy maps created for this test had well-defined borders with simple, generic obstacles.

The second type of test involved MATLAB simulations using real-world parameters. The occupancy maps were created using lidar scans of real environments, accounting for the robot's size and scaling its coverage radius to match the environment during navigation. The robot could not be assumed to fit perfectly into each grid cell, as this was not true. This test also accounted for the imperfections of the lidar scanned maps, such as non-well-defined borders, irregular obstacle shapes, and areas too small for the robot to fit. Proper spacing between the robot and the edges of borders/obstacles was implemented to simulate collision avoidance.

Lastly, ROS simulations were conducted to mimic real-world performance as closely as possible. An enclosed virtual environment was built in Gazebo World, spawning a simulated robot equipped with lidar to create the occupancy grid map of the environment. The neural network coverage algorithm generated waypoints for the robot to follow, and the controller used the given waypoints and the robot's current pose to generate linear and angular velocity to drive the robot in the virtual environment.

Throughout testing and data collection, the focus was on evaluating the system's performance based on research objectives. Performance specifications evaluated included coverage efficiency, turn count, path length, and path overlap, as well as the measurement of simulation runtimes. While the main concern was how the system would perform in a simulated

real-world setting, a hypothesis was tested that the start locations of the robot would affect the measured data. To test the hypothesis, an experiment was conducted analyzing the effects of two different start locations (center location and top left corner) on coverage efficiency, duration, and path overlap. It was theorized that starting at corner locations would yield better performance than center locations. The results are shown below. It is worth noting that these results reflect the best algorithm model parameters found during testing.

5.1. MATLAB Simulation Results

List of Model parameters: ***A*** *= 30,* ***B*** *= 1,* ***D*** *= 1,* ***U*** *= 0.7,* ***I*** *= -100, 0, 100*

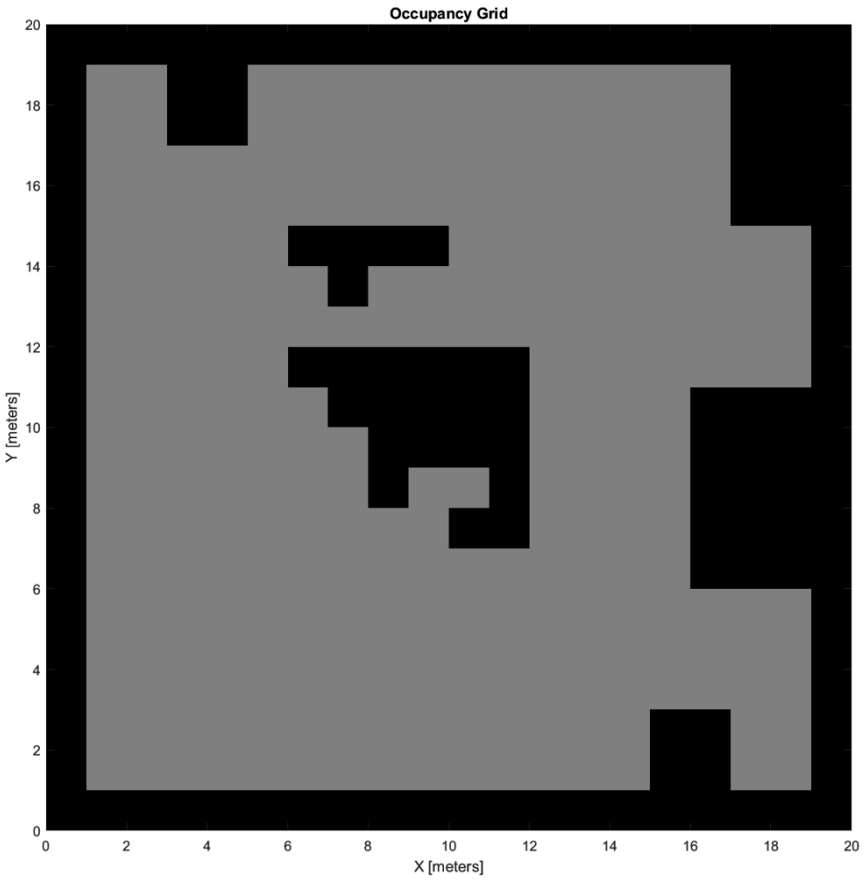


Figure 12. Irregular-Shaped Obstacle Test Map

Table 2. Irregular Shaped Obstacle Test Map Data, Corner Location Start

Corner location	
Starting Coordinate *(row-column grid index)*	[2, 2]
Turn Count	159
Path Length (m)	434.55
Overlap	133
Sim Runtime (s)	12.02

Table 3. Irregular Shaped Obstacle Test Map Data, Center Location Start

Center location	
Starting Coordinate *(row-column grid index)*	[8, 11]
Turn Count	140
Path Length (m)	356.78
Overlap	66
Sim Runtime (s)	16.61

Figure 12 illustrates the results of the first test, which was conducted on a map containing both irregularly shaped obstacles and borders. For this test, coverage efficiency was not measured, as the setup ensured that all uncovered locations would be covered. The starting corner location used was the row-column grid index [2, 2], with the center locations set at [8, 11]. It should be noted that the test results were consistent across multiple iterations, with the only variation occurring when start locations were changed.

The results show that the center location offered better coverage performance, with a reduction in turns of approximately 12%, a shorter path of about 18%, and less overlap by roughly

51% in navigation. When considering the average of these percentages, the overall performance difference was 27%, favoring the center location. Of particular interest was the significant decrease in the overlap. The simulation runtimes were not analyzed, but rather utilized to provide a general estimate of program speed.

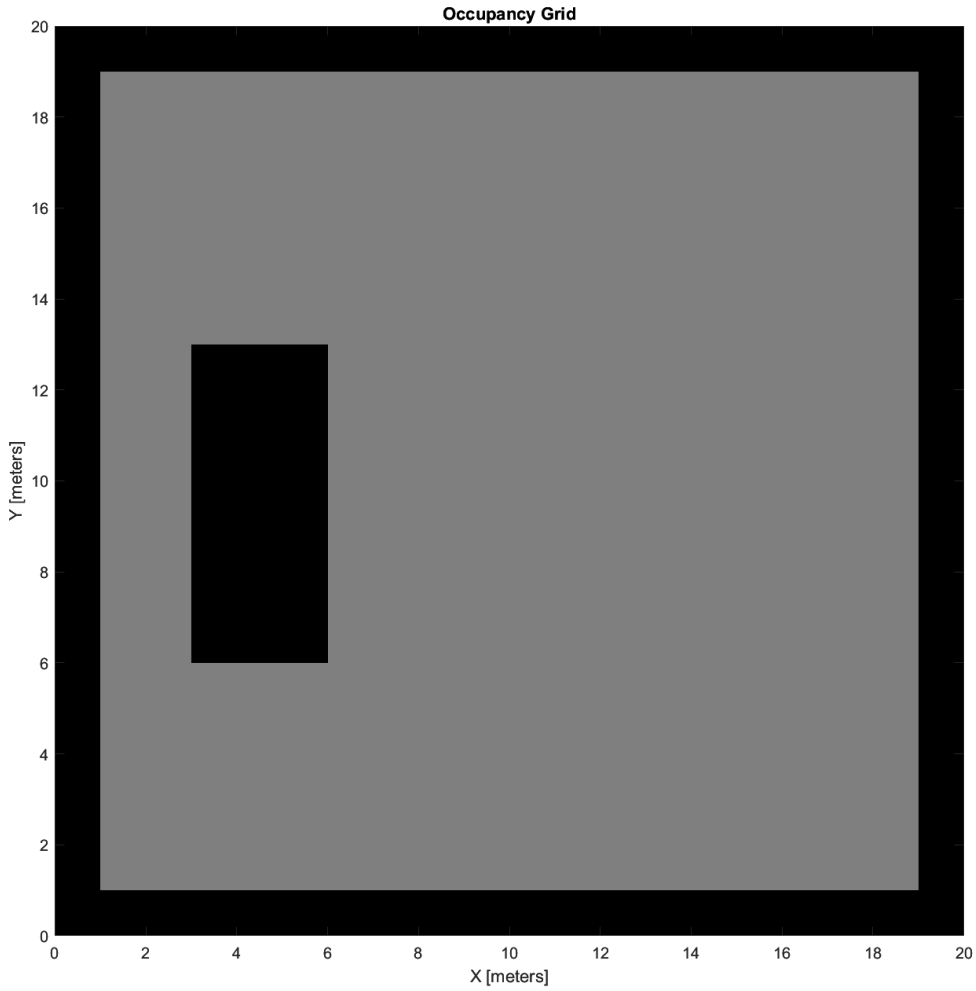


Figure 13. Rectangular-Shaped Obstacle Test Map

Table 4. Rectangular Shaped Obstacle Test Map Data, Corner Location Start

Corner location	
Starting Coordinate *(row-column grid index)*	[2, 2]
Turn Count	86
Path Length (m)	310.49
Overlap	6
Sim Runtime (s)	14.26

Table 5. Rectangular Shaped Obstacle Test Map Data, Center Location Start

Center location	
Starting Coordinate *(row-column grid index)*	[10, 10]
Turn Count	122
Path Length (m)	313.89
Overlap	9
Sim Runtime (s)	8.17

The subsequent test was conducted using a map that included an obstacle in a rectangular shape, as seen in **Figure 13**. The starting corner location was identified by the row-column grid index [2,2], and the center locations were set at [10,10]. The outcomes showed that beginning from the corner location resulted in superior coverage performance. The navigation had fewer turns by approximately 30%, the path was approximately 2% shorter, and there was approximately 33% less overlap. When the average of these percentages was taken, the overall performance was found to be 22% better when starting from the corner location.

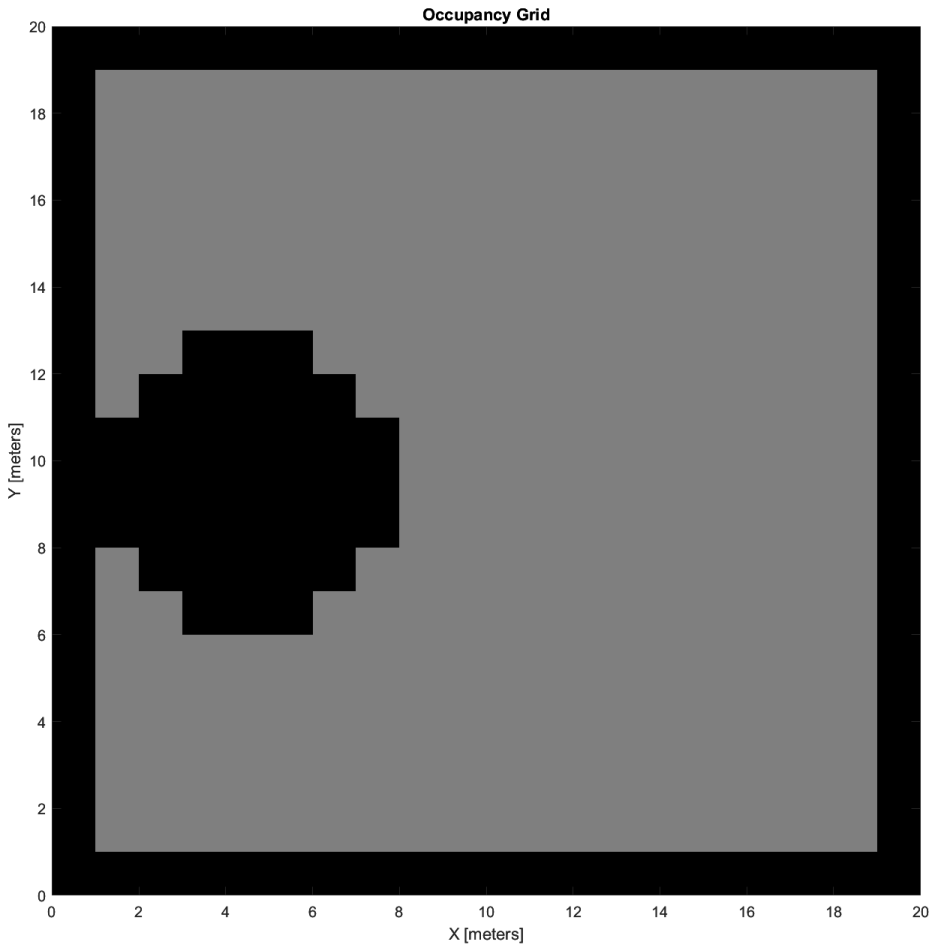


Figure 14. Circular Shaped Obstacle Test Map

Table 6. Circular Shaped Obstacle Test Map Data, Corner Location Start

Corner location	
Starting Coordinate *(row-column grid index)*	[2, 2]
Turn Count	123
Path Length (m)	321.35
Overlap	25
Sim Runtime (s)	7.88

Table 7. Circular Shaped Obstacle Test Map Data, Center Location Start

Center location	
Starting Coordinate *(row-column grid index)*	[10, 10]
Turn Count	165
Path Length (m)	356.81
Overlap	53
Sim Runtime (s)	7.88

The final test of this series involved a map featuring a circular obstacle, as depicted in **Figure 14**. Once again, the starting corner location was set to the row-column grid index [2,2], while the center location was set to [10,10]. The results revealed that starting at the corner location yielded better coverage performance. Specifically, there were 25% fewer turns, a 10% shorter path, and 24% less overlap in the navigation. On average, the overall difference in performance was 20% in favor of the corner location.

5.2. MATLAB Simulations with Real-world Parameters Results

List of Model parameters: ***A*** *= 70,* ***B*** *= 1,* ***D*** *= 1,* ***U*** *= 0.7,* ***I*** *= -100, 0, 100*

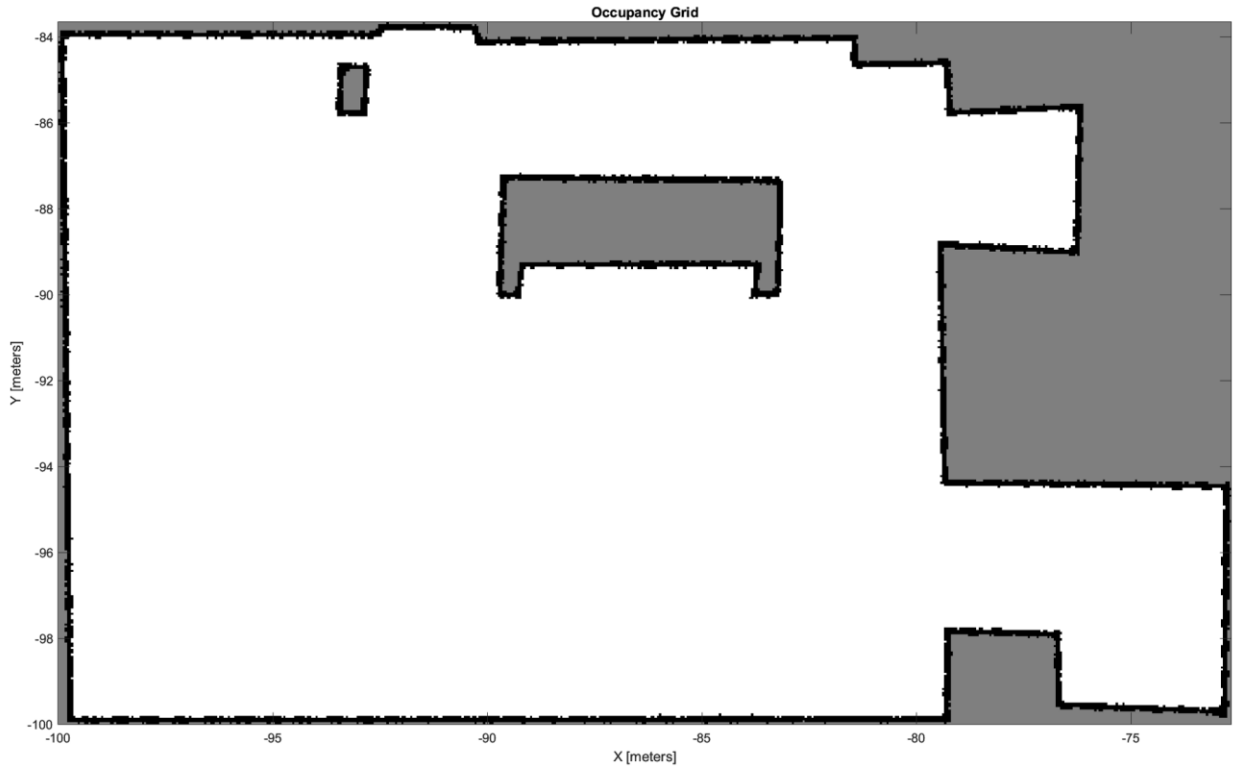


Figure 15. MATLAB Simulation Map Scaled for Robot Size, Living Room

Table 8. Corner Location Start Data, Living Room

Corner location	
Starting Coordinate *(row-column grid index)*	[15, 18]
Coverage efficiency (%)	79.14
Turn Count	117
Path Length (m)	340.23
Overlap	87
Sim Runtime (mins)	1.69

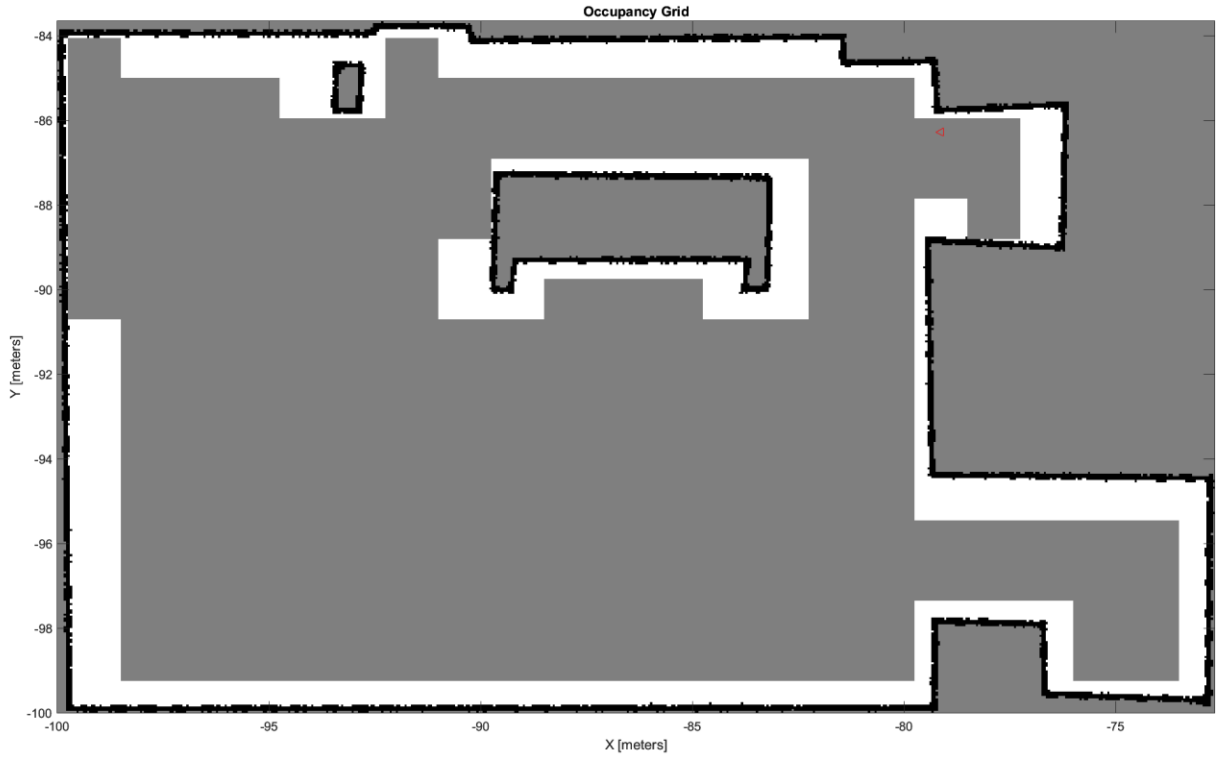


Figure 16. Corner Start Coverage for Living Room Map

Table 9. Center Location Start Data, Living Room

Center location	
Starting Coordinate *(row-column grid index)*	[150, 250]
Coverage efficiency (%)	81.92
Turn Count	95
Path Length (m)	273.92
Overlap	28
Sim Runtime (mins)	1.44

Figure 17. Center Start Coverage for Living Room Map

The map shown in **Figure 15** is a lidar-scanned map of an apartment living room. The purpose of these tests was to measure coverage efficiency, as not all locations can realistically be covered. The row-column grid index [15,18] was used as the starting corner location, and [150, 250] was used for the center locations. Like the tests for the initial MATLAB simulations, the results for each start location were the same for multiple tests using the same start locations, with the only difference occurring after the start locations were changed. It is worth noting that the map used for this test is an edited and cleaner version of the original lidar-scanned map.

The results show that better coverage performance was achieved when starting at the center location. The navigation involved fewer turns by about 18%, a shorter path by about 19%, and less overlap by about 68%. Although the coverage efficiency for both starting locations was relatively

similar, the overall difference in performance was 35% better when starting at the center location. **Figure 16** and **Figure 17** provide images of the covered area after navigation for both starting locations.

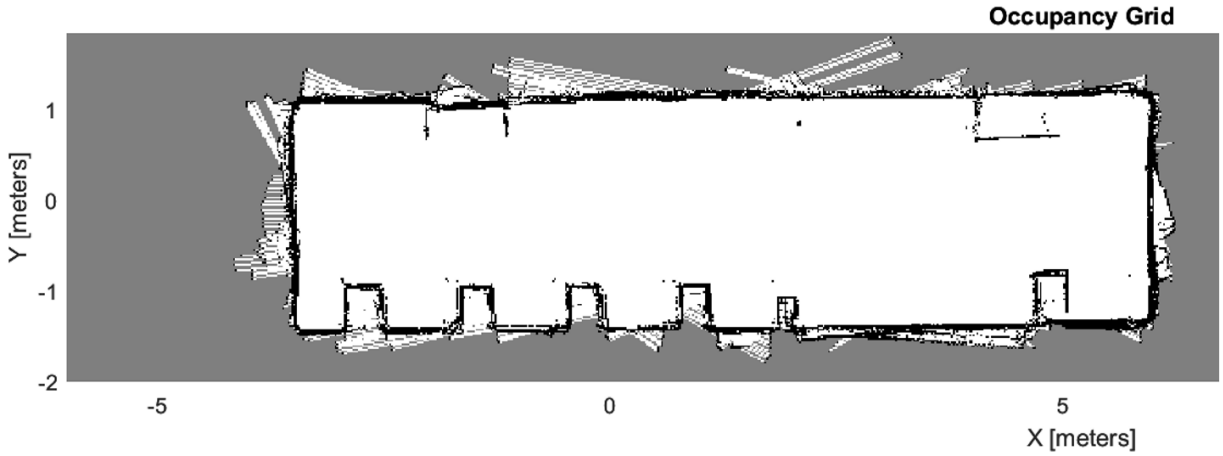


Figure 18. MATLAB Simulation Scaled Map, Lab Space

Table 10. Corner Location Start Data, Lab Space

Corner location	
Starting Coordinate *(row-column grid index)*	[68, 135]
Coverage efficiency (%)	58.15
Turn Count	49
Path Length (m)	34
Overlap	3
Sim Runtime (mins)	0.80

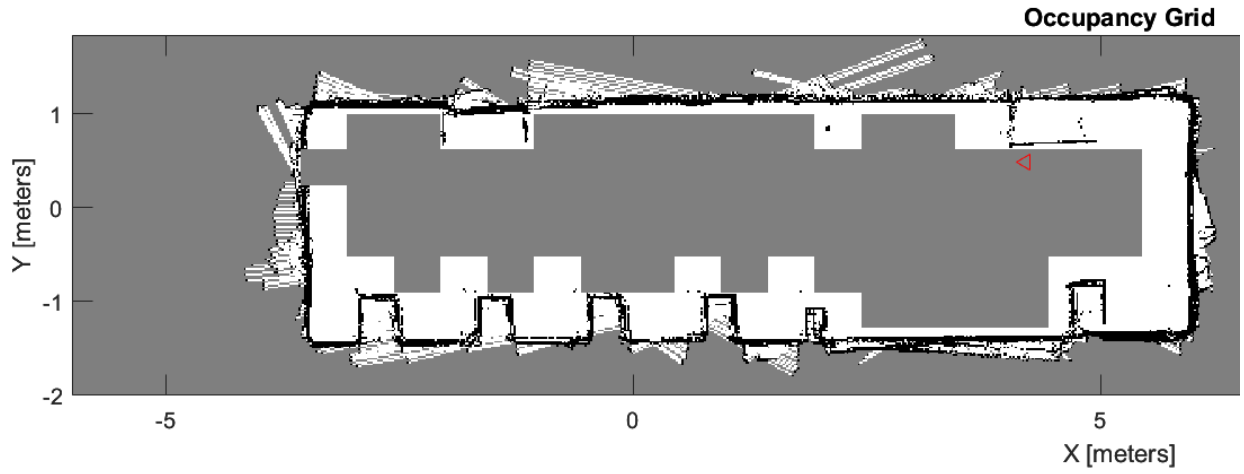


Figure 19. Center Start Coverage for Lap Space Map

Table 11. Center Location Start Data, Lab Space

Center location	
Starting Coordinate *(row-column grid index)*	[93, 350]
Coverage efficiency (%)	55.96
Turn Count	43
Path Length (m)	40
Overlap	12
Sim Runtime (mins)	0.42

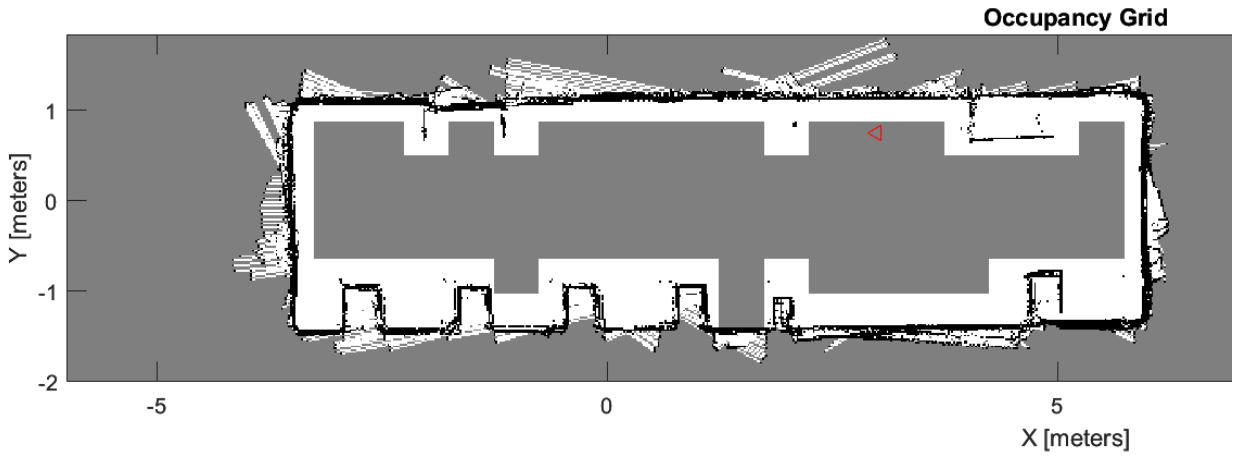


Figure 20. Corner Start Coverage for Lap Space Map

The map depicted in **Figure 18** is a lidar-scanned map of a common area in a lab space. The starting corner location was set to the row-column grid index [68, 135], while the center location was set to [93, 350]. As observed from the results, better coverage performance was achieved when starting at the corner location. The navigation involved fewer turns by about 12% when starting from the center. However, there was a 15% shorter path, and 75% less overlap when starting from the corner. Although coverage efficiency was similar for both starting locations, the overall difference in performance was about 34% in favor of the corner. location. **Figure 19** and **Figure 20** display the covered area after navigation for both starting locations.

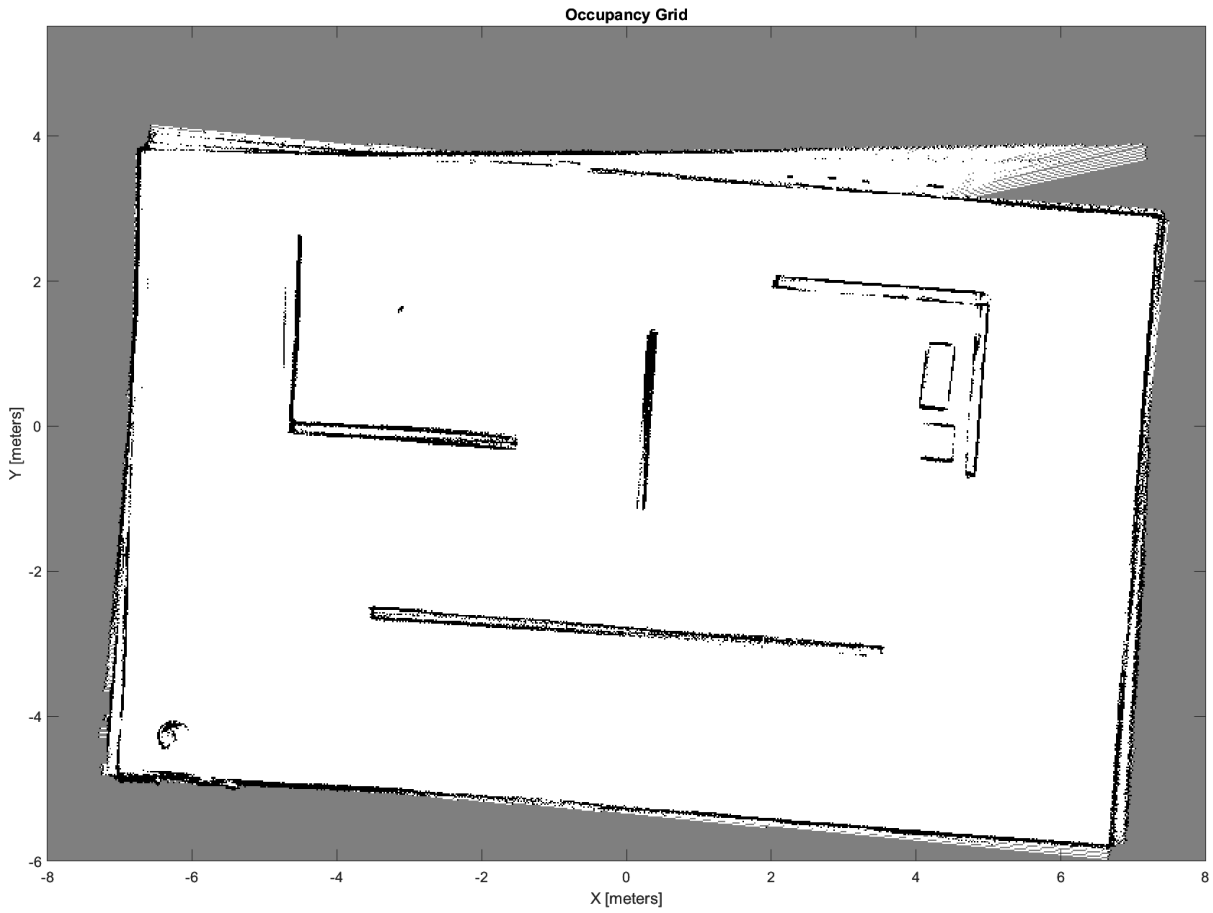


Figure 21. MATLAB Simulation Scaled Map, Gazebo Environment

Table 12. Center Location Start Data, Gazebo Environment

Corner location	
Starting Coordinate *(row-column grid index)*	[127, 100]
Coverage efficiency (%)	75.64
Turn Count	272
Path Length (m)	273
Overlap	135
Sim Runtime (mins)	3.55

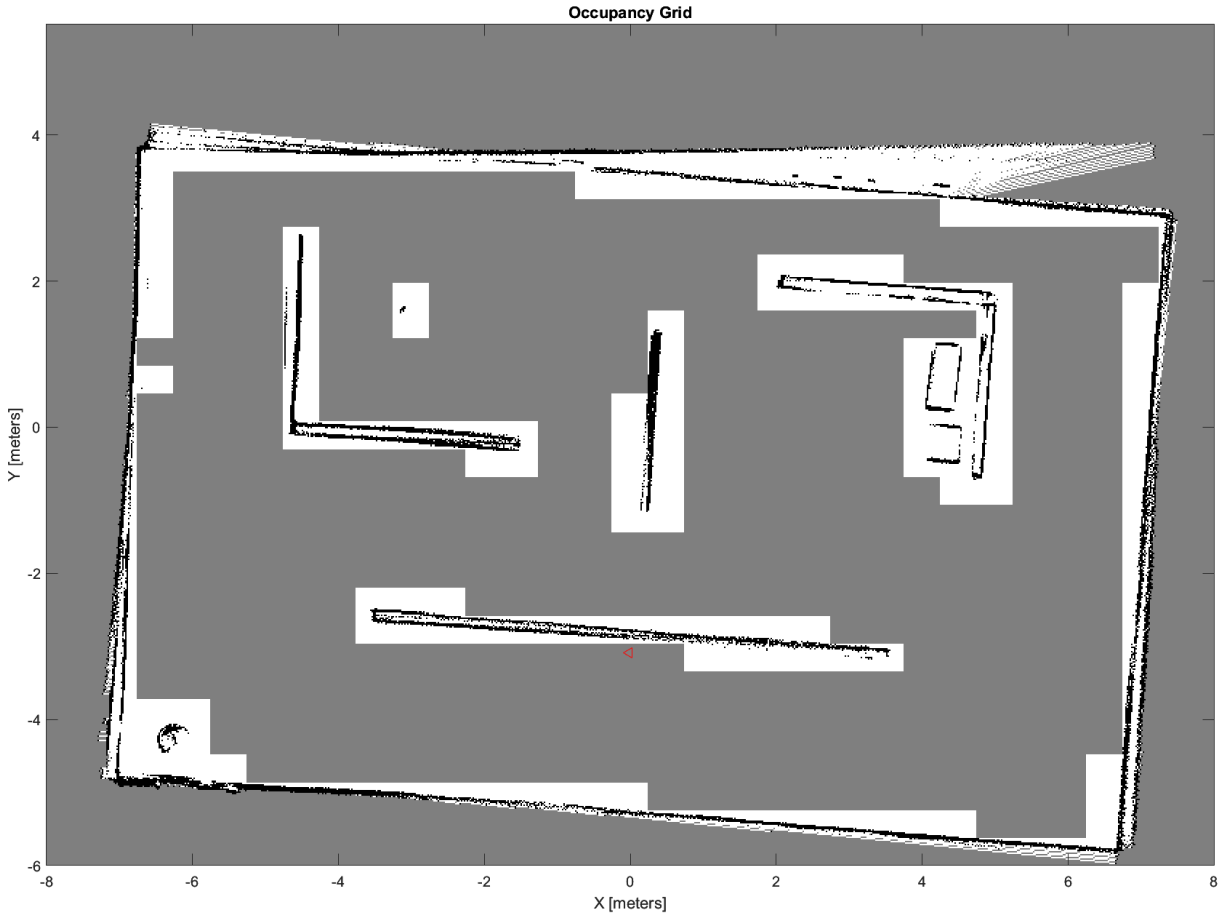


Figure 22. Corner Start Coverage for Gazebo Environment Map

Table 13. Center Location Start Data, Gazebo Environment

Center location	
Starting Coordinate *(row-column grid index)*	[277, 400]
Coverage efficiency (%)	76.40
Turn Count	288
Path Length (m)	268
Overlap	115
Sim Runtime (mins)	3.79

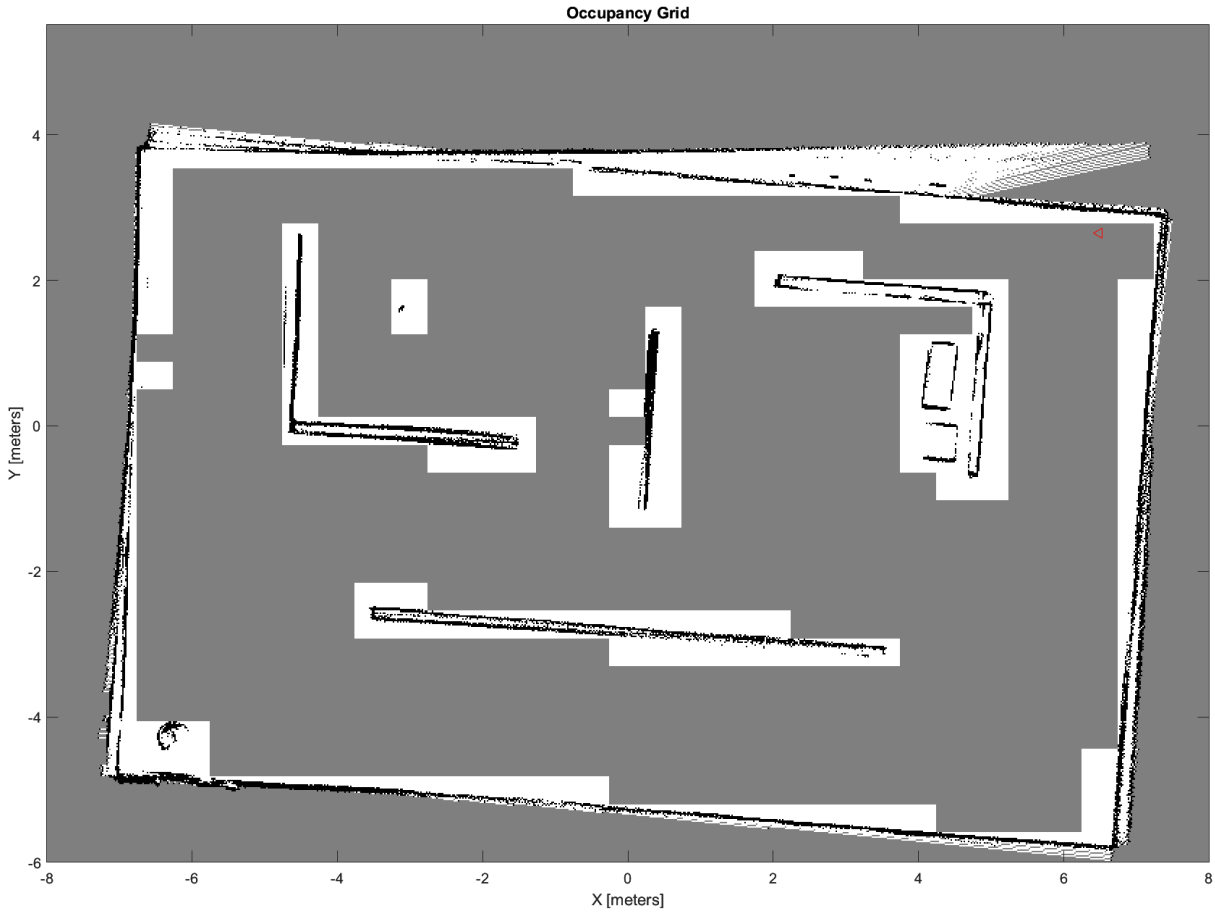


Figure 23. Center Start Coverage for Gazebo Environment

Figure 21 displays a lidar scan of the enclosed virtual environment in Gazebo World, where the row-column grid index [127, 100] served as the starting corner location and [277, 400] as the center location. The results revealed a 6% increase in the number of turns when starting from the center location, but a 2% reduction in path length and a 15% decrease in overlap during navigation. Although the coverage efficiency for both starting locations was similar, there was an overall performance improvement of 8% when starting at the center location. Nonetheless, the differences were relatively small, so the coverage performance was comparable. **Figure 22** and **Figure 23** illustrate images of the covered area after navigation for both starting locations.

5.3. ROS Simulation Results

List of Model parameters: ***A*** *= 70,* ***B*** *= 1,* ***D*** *= 1,* ***U*** *= 0.7,* ***I*** *= -100, 0, 100*

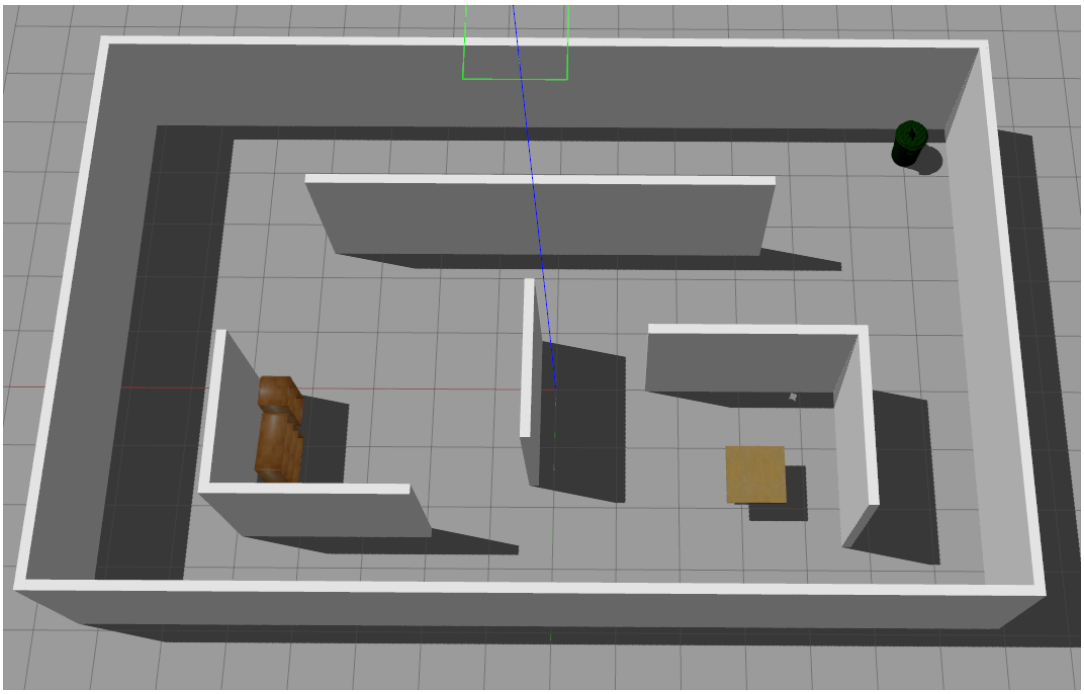


Figure 24. Enclosed Gazebo Environment

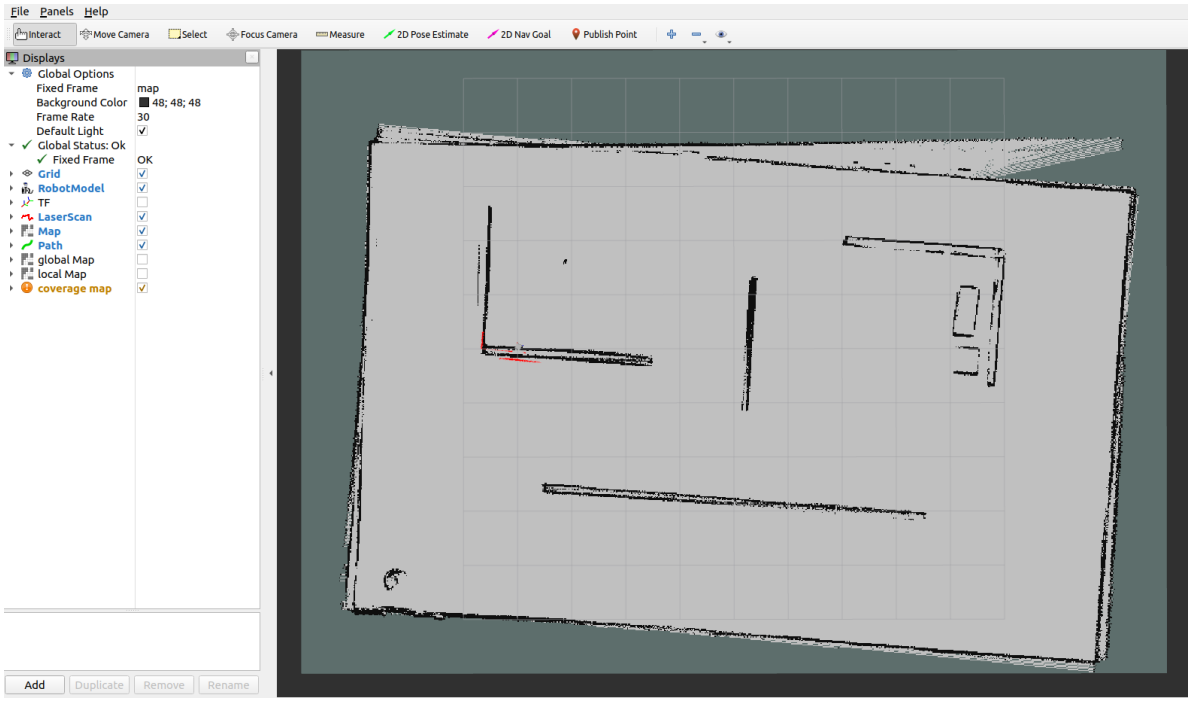


Figure 25. RViz Representation of Mapped Environment

Table 14. Corner Location Start Data, ROS simulation

Corner location			
Trial Runs	Run 1	Run 2	Run 3
Starting Coordinate *(row-column grid index)*	[103, 80]	[101,82]	[91,77]
Coverage efficiency (%)	81.53	80.32	80.24
Turn Count	329	312	275
Path Length (m)	331	303	276.78
Overlap	246	188	147
Sim Runtime (mins)	47.03	42.58	39.58

Table 15. Center Location Start Data, ROS simulation

Center location			
Trial Runs	Run 1	Run 2	Run 3
Starting Coordinate *(row-column grid index)*	[283, 400]	[284, 401]	[289, 402]
Coverage efficiency (%)	78.52	79.40	81.01
Turn Count	246	333	341
Path Length (m)	275	335	323
Overlap	149	258	228
Sim Runtime (mins)	38.28	47.63	47.36

Figure 24 shows the enclosed virtual environment built in Gazebo World. **Figure 25** shows the representation of the mapped environment in RViz. For this test there were three trial runs done to

get an estimate of the robot's performance. Based on these results, it appears that both the center and corner starting locations led to relatively similar coverage efficiency. However, the corner starting location generally led to fewer turns and less overlap, while the center starting location often led to shorter path lengths. The differences in performance between the two starting locations were relatively small, with only slight variations in coverage efficiency, turn count, and path length. Therefore, the choice of starting location may depend on specific factors such as the layout of the environment and the desired level of coverage efficiency.

Based on the findings presented, the optimal starting location for coverage efficiency and performance depends on the specific environment and obstacles present. In some cases, starting from the center location may lead to better performance, while in other cases starting from the corner location may be more advantageous. Additionally, the results suggest that starting from the center location can lead to a significant decrease in overlap, which is of particular interest in navigation tasks. However, it is important to note that the differences in performance between starting locations were relatively small in some cases, and coverage efficiency was similar for both starting locations. Therefore, the choice of starting location should be carefully considered based on the specific environment and objectives of the task at hand.

6. TECHNICAL CHALLENGES, RESEARCH CONTRIBUTION, & FUTURE WORK

6.1. Technical Challenges

Although the configuration of the physical system was successful, testing for consistent robot performance proved to be quite challenging due to the several issues detailed in the following sections.

6.1.1. ROS configuration

Setting up basic robot navigation in ROS was one of the most challenging aspects of this research. It was important to have a basic understanding of ROS and a good grasp of how to configure the robot navigation stack. Information on ROS navigation can be found in the ROS wiki [30]. Basic examples of how to set up the robot navigation stack are shown. While the scope of work for this thesis was limited to ROS simulation, two identical differential drive robots were built for the attempt at implementation on a physical system. Given the implementation of a custom-built robot, a deeper understanding of ROS was needed to properly configure this robot for navigation. This required knowledge in areas such as ROS topics, messages, and nodes, as well as how to interface with sensors and motors. More information on the physical robot ROS configuration can be found in [29]. However, to mitigate the challenges of setting up a custom-built robot on ROS, a pre-configured differential drive robot such as the TurtleBot can be acquired instead.

6.1.2. Configuration of Multi-Robots in RViz

Sufficient information on configuring a single robot for navigation in ROS exists. However, at the time of this research, not enough information was available on configuring multiple robots in ROS. For implementing multi-robot visualization in RVIZ, the coverage of both robots in a single map was needed. Both robots had identical setups, so spawning both robots on the ROS network resulted in issues such as the ROS node of one robot overriding that of the other, causing conflicts when sending data to the same ROS topics. Additionally, issues were encountered when spawning two identical robot models in RVIZ. To address these issues, the namespace feature in ROS was utilized, allowing for multiple instances of identical nodes/topics to be set up in different groups. These groups could then be linked together using a common node, with some topics requiring remapping. More information on ROS namespaces and remapping can be found in references [31][32]. Information on how to configure multi-robots in RVIZ is referenced in [33].

6.1.3. Robot localization

One of the challenges faced during this research was achieving proper localization of the robot on the map using the current setup. As outlined in the methodology section, the ROS package AMCL was utilized to correct odometry drift by matching laser scans from the LIDAR data onto the map. However, with the current robot setup, after multiple turns, specifically point turns, significant drift occurred in the laser scans from the map outlines. In some instances, the laser scans were offset from the map by approximately 90 degrees, leading to severe misinterpretation of the robot's location on the map. While the root cause of this issue could not be pinpointed, compensation for the drift was achieved by increasing the default values of the *odom_alpha* [22] parameter on *amcl*. Additionally, an alternative localization package, *gmcl* [34], was explored.

6.1.4. Robot Controller

As detailed in the methodology section, the autonomous robot controller utilized to generate driving commands for the robot was the *controllerPurePursuit* [28], a path-tracking algorithm provided by MATLAB for differential drive vehicles. However, a significant issue encountered was the controller's tendency to take a smooth, curved path forward to the next target location, making it more effective when navigating to locations further away. This approach was suboptimal for the coverage application since the robot needed to move in discrete (zig-zag) patterns over short distances.

During navigation, it was common for the controller to drive the robot through a long-curved path to a target location that was relatively close. This occurred more often when the target location was too close to the robot's current location, and a turn in either direction was required to reach the location. This issue caused the robot to attempt to drive to an undesired location in some cases, such as when the robot was in a corner and needed to navigate to the location directly behind it. The controller would opt to drive the robot further out in the forward direction before making a turn, causing the robot to drive into a wall. While the obstacle avoidance feature prevented the robot from hitting the wall, the controller still attempted to overshoot the trajectory when driving control was returned to it. This caused instances where the robot got stuck in a self-induced deadlock. Multiple recovery behaviors were attempted, but since the occurrence of this self-induced deadlock varied each time, a reliable solution that consistently worked could not be found. Increasing the controller's maximum angular velocity helped mitigate the issue, but it did not completely solve it.

Tuning the robot controller for the desired driving behavior was also challenging, but the parameters seen in the ROS simulation program were settled on. These parameters provided consistent driving performance.

6.1.5. Optimizing code parameters for obstacle recovery

During testing of the robot's navigation, scenarios were observed in which the robot became stuck in its environment, revealing a blind spot in the robot's setup. The robot perceives its environment through two methods: global perception using the provided map and local perception using live lidar scans. While lidar scans are used to update the map during navigation, they have limitations due to the use of a 2D lidar mounted at the top of the robot, which can only detect obstacles above the lidar scan plane. Consequently, the robot is susceptible to running into obstacles below the lidar scan plane, which is its blind spot. In some cases, the robot became stuck behind an obstacle in this blind spot, making it challenging to find a reliable recovery behavior as the conditions varied each time, and the robot had to account for an event it could not perceive.

The primary solution to this issue was to manually remove as many obstacles as possible from the robot's blind spot. Another solution involved using localization information to determine when the robot was stuck. If the robot's pose did not change for more than five seconds during navigation, a recovery behavior was initiated. This involved reversing the motion that caused the robot to become stuck, marking both the location it was stuck at and the intended destination as obstacles on the map. This approach was complex because it required considering the robot's previous driving commands to that point, and marking spots as obstacles could result in the robot becoming stuck multiple times while attempting to perform its recovery behaviors. This could lead to instances in which the navigation program trapped the robot in a deadlock.

In some instances, the robot's wheels continued to spin even though it was stuck, leading to false odometry data from the encoder values, which interfered with the localization correction done by the AMCL ROS package. Consequently, there were situations in which the robot was stuck, but the program could not register it as such because its pose changed within the five-second time interval.

6.1.6. Configuring ROS in MATLAB's Parallel Computing Toolbox

For a multi-robot navigation setup, knowledge of MATLAB's parallel computing toolbox is essential. Running multiple navigation codes simultaneously within a single MATLAB application is desirable. However, integrating ROS with MATLAB's parallel computing toolbox was a challenging task. Although a potential solution using *spmd* [35] was discovered, it requires further exploration, as detailed in [36]. Currently, to run multiple ROS nodes, separate MATLAB applications must be executed concurrently. The unsuccessful attempt to set up ROS with MATLAB's parallel computing toolbox affected obstacle detection in the navigation program. In an ideal scenario, obstacle detection should be run in parallel, where a node continually reads published lidar scans and interrupts the main navigation code when an obstacle is detected. However, the current program's obstacle detection is performed before sending driving commands to the robot, causing the robot to collide with undetected obstacles due to the navigation program's sequential nature.

6.1.7. Replicating results of the next position decision formula

The replication of the path selection algorithm specified in the original journal article has been a significant challenge in the progression of this research. The theoretical output of the algorithm should result in a zig-zag coverage path. However, when simulating the algorithm in MATLAB, it produced a spiral coverage path. Further analysis revealed that the spiral path is consistent with

the proposed algorithm, which is designed to make the least amount of turns from the robot's current heading direction. During simulation, the robot moves forward and makes a turn in either the left or right direction when it encounters an obstacle, such as a wall in the corner of a rectangular room. It continues moving forward in the new heading direction only if the spaces are uncovered, resulting in a spiral pattern along the uncovered spaces in the mapped environment.

To generate a zig-zag movement pattern, the robot must make at least two turns. The first turn is either left or right from the current heading when it arrives at a wall in the corner of a rectangular room. Then, it needs to make another left or right turn from its new heading if the spaces in those directions are uncovered. Although the spiral coverage path works well when the robot navigates an empty rectangular room without any obstacles, it fails to cover the entire area when obstacles are present. The spiral navigation path eventually loops around the walls of the environment indefinitely and misses some areas in the middle of the room.

Further research is necessary to determine how to replicate the zig-zag coverage path using the path selection algorithm. The analysis suggests that the algorithm's bias towards making the least number of turns should be modified to ensure the robot covers the entire area.

6.1.8. The discrepancy in navigation between robot scaling and initial simulation

During testing of the initial MATLAB simulation code, the robot was able to find a direct path to uncovered locations even when stuck in a deadlock situation, where all surrounding spaces were covered but other uncovered spaces existed elsewhere on the map. However, in the MATLAB code generated for scaling the real robot's diameter to its appropriate coverage area on the map, the direct pathfinding performance could not be replicated as reliably. In deadlock situations, the robot would eventually reach the uncovered spaces, but only after many random turns. Despite various adjustments to algorithm variables such as the passive decay rate and μ constant, which

influence the behavior of the neural propagation that determines the algorithm's pathfinding capabilities, this performance issue could not be resolved.

6.2. Research Contributions

- Developed a cost-effective and power-efficient proof of concept on an industrial robotic platform capable of covering an enclosed area in an orderly and structured manner.
    - By utilizing a single-board computer (SBC) instead of a mobile PC [37], the proposed system is cost-effective and consumes less power for efficient performance.
- Implemented a navigation system using a combination of the proposed neural network algorithm [9] and heuristic path selection strategy [16].
    - Combining these methods achieved the research objective of complete coverage path planning with higher coverage completeness, lower path repetition rate, and less path execution time.
- Expanded on the test conditions for the developed robotic system in [37] to accommodate more real-world environments by using lidar maps of real-world environments.

6.3. Future Work

This list of future work is derived from some of the technical challenges encountered.

- Upgrade to the latest ROS version, as the current version (ROS Noetic) may no longer be supported.
- Install additional perception sensors on the robot to enhance its understanding of its environment and location within it. Some examples of such sensors include:

    - GPS, which can help resolve localization issues in situations where the robot is stuck but its wheels continue to spin.
    - 3D LiDAR
    - Camera
    - Inertial Measurement Unit (IMU)
- Install sensors to monitor the robot's system vitals, such as:
    - Battery voltage
    - Power consumption
- Develop a precise point-and-go controller for the robot, which can improve its driving performance during coverage navigation in the environment.
- Develop recovery behaviors in the navigation program that can reliably guide the robot out of situations in which it is stuck. Recovery behaviors in the code could involve:
    - Maintaining a log of all driving commands until the robot reaches its next position.
    - Marking the location if the robot gets stuck on its way to the next position.
    - Reversing the logged driving commands to escape.
    - Clearing the marked spots once the robot has exited the stuck situation.
- Determine how to properly run ROS in the MATLAB Parallel Computing Toolbox to achieve simultaneous obstacle detection and avoidance during navigation and ensure proper scalability for multiple robots.
- Consider having each robot run its version of the navigation code and simply subscribe to the main service hosting the map for a more practical application.

7. CONCLUSION

In summary, this research presents a proof of concept for an industrial robotic platform that can achieve complete coverage path planning using the proposed neural network algorithm and heuristic path selection strategy. The results of this research validate the feasibility of using this approach in practical industrial applications and offer a promising avenue for further research in this area.

REFERENCES

1. MathWorks. (n.d.). occupancymap. Retrieved from https://www.mathworks.com/help/nav/ref/occupancymap.html

2. MathWorks. (n.d.). occupancymap (Robotics System Toolbox). Retrieved from https://www.mathworks.com/help/nav/ref/occupancymap.html#bvaw60t-6

3. MathWorks. (n.d.). occupancymap (Robotics System Toolbox). Retrieved from https://www.mathworks.com/help/nav/ref/occupancymap.html#bvaw60t-3

4. Paul Timothy (2019). Mapping and SLAM Course - 9 - Occupancy Grid Mapping Algorithm - YouTube. Retrieved from https://www.youtube.com/watch?v=VZZVhVn2bec

5. Tiziano Fiorenzani (2020). Autonomous Navigation with ROS for Beginner #4: SLAM and Occupancy Grid Mapping - YouTube. Retrieved from https://www.youtube.com/watch?v=cQh0gNfb6ro

6. syuntoku14. (2020). syuntoku14/fusion2urdf. GitHub. Retrieved from https://github.com/syuntoku14/fusion2urdf

7. Galceran, E. and M. Carreras, A survey on coverage path planning for robotics. Robotics and Autonomous Systems, 2013. 61(12): p. 1258-1276.

8. Palacín, J., et al. Measuring coverage performances of a floor cleaning mobile robot using a vision system. in Proceedings of the 2005 IEEE international conference on robotics and automation. 2005. IEEE.

9. Yang, S.X. and C. Luo, A neural network approach to complete coverage path planning. IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics), 2004. 34(1): p. 718-724.

10. Xu, P.-F., Y.-X. Ding, and J.-C. Luo, Complete Coverage Path Planning of an Unmanned Surface Vehicle Based on a Complete Coverage Neural Network Algorithm. Journal of Marine Science and Engineering, 2021. 9(11): p. 1163.

11. Luo, C. and S.X. Yang, A bioinspired neural network for real-time concurrent map building and complete coverage robot navigation in unknown environments. IEEE Transactions on Neural Networks, 2008. 19(7): p. 1279-1298.

12. Luo, C., et al. A computationally efficient neural dynamics approach to trajectory planning of an intelligent vehicle. in 2014 International Joint Conference on Neural Networks (IJCNN). 2014. IEEE.

13. Sun, B., et al., Complete coverage autonomous underwater vehicles path planning based on glasius bio-inspired neural network algorithm for discrete and centralized programming. IEEE Transactions on Cognitive and Developmental Systems, 2018. 11(1): p. 73-84.

14. MathWorks. (n.d.). BinaryOccupancyMap. Retrieved from https://www.mathworks.com/help/nav/ref/binaryoccupancymap.html

15. MathWorks. (n.d.). norm. Retrieved from https://www.mathworks.com/help/matlab/ref/norm.html

16. Lin, X., Dong, Y., Li, L., Li, Y., & Liu, S. (2019). A Complete Coverage Path Planning Method for Mobile Robot in Uncertain Environments. IEEE Access, 7, 66577-66588. doi: 10.1109/access.2019.2911689

17. Robotics Knowledgebase. (2017, October 15). ROS Tutorial #10: GMapping Demo. [Video]. YouTube. https://www.youtube.com/watch?v=5nZc5iSr5is&list=PLiiw0aSVHcAkF26qR6Q7x6RlLAL6-vuF3

18. ROS. (n.d.). Navigation Stack. Retrieved from http://wiki.ros.org/navigation

19. Victor056. (n.d.). bbl_ws. GitHub. https://github.com/victor056/bbl_ws

20. Microchip Technology Inc. (2017, October 31). Autonomous Mobile Robot Navigation using ROS. [Video]. YouTube. https://www.youtube.com/watch?v=S8pwfsK-F9w

21. Robotics Knowledgebase. (2017, September 20). ROS Tutorial #7: AMCL Localisation. [Video]. YouTube. https://www.youtube.com/watch?v=UBG1ibBNTiQ

22. ROS. (n.d.). amcl. Retrieved from http://wiki.ros.org/amcl

23. ROS. (n.d.). gmapping. Retrieved from http://wiki.ros.org/gmapping

24. ROS. (n.d.). map_server. Retrieved from http://wiki.ros.org/map_server

25. ROS. (n.d.). Navigating the ROS Wiki. Retrieved from http://wiki.ros.org/ROS/Tutorials/NavigatingTheWiki

26. ROS. (n.d.). rviz. Retrieved from http://wiki.ros.org/rviz

27. Foxglove. (n.d.). About Foxglove Studio. Retrieved from https://foxglove.dev/

28. MathWorks. (n.d.). Pure Pursuit Controller. Retrieved from https://www.mathworks.com/help/nav/ug/pure-pursuit-controller.html

29. Victor056. (n.d.). Multi Robot Coverage. Retrieved from https://github.com/victor056/multi_robot_coverage

30. ROS. (n.d.). Navigation. Retrieved from http://wiki.ros.org/navigation

31. Nootrix. (n.d.). ROS Namespaces: Basics and Tricks. Retrieved from https://robots.nootrix.com/diy-tutos/ros-namespaces/

32. ROS. (n.d.). Remapping arguments (roslaunch/XML). Retrieved from http://wiki.ros.org/roslaunch/XML/remap

33. ROS. (n.d.). Navigation Tutorials. Retrieved from https://www.youtube.com/watch?v=es_rQmlgndQ

34. ROS. (n.d.). gmcl. Retrieved from http://wiki.ros.org/gmcl

35. MathWorks. (n.d.). spmd. Retrieved from https://www.mathworks.com/help/parallel-computing/spmd.html

36. MathWorks. (n.d.). How to publish ROS message inside a parfor loop? Retrieved from https://www.mathworks.com/matlabcentral/answers/324560-how-to-publish-ros-message-inside-a-parfor-loop

37. Li, Y., Liu, J., Zhang, Y., & Zhu, Y. (2021). Neural-Dynamics-Driven Complete Area Coverage Navigation Through Cooperation of Multiple Mobile Robots. IEEE Transactions on Cybernetics, 51(3), 1167-1180. doi: 10.1109/TCYB.2020.2967082

38. Victor056. (n.d.). CompleteCoveragePathPlanning. Retrieved from https://github.com/victor056/CompleteCoveragePathPlanning

APPENDIX A

A-1: MATLAB Simulation Code

This program assumes a known map state. Code prompts the user to input a selection of available Excel grid maps. Generates a neural activity matrix of the same size as the selected map. Updates the entire neural activity level for each grid in the map and uses the heuristic method for path selection.

```
clear
clc

select = input('1-Rectangle obstacle, 2-Circular, 3-Generic, 4-100 park map: ')

if select == 1
   env = readmatrix("map builder.xlsx", "Sheet","Rectangle obstacle");
elseif select == 2
   env = readmatrix("map builder.xlsx", "Sheet","Circular obstacle");
elseif select == 3
   env = readmatrix("map builder.xlsx", "Sheet","test");
elseif select == 4
   env = readmatrix("map builder.xlsx", "Sheet","100 park");
end

map = occupancyMap(env);
n = zeros(map.GridSize(1), map.GridSize(2)); %generate array to store neural activity values
mat = occupancyMatrix(map); %an array containing the actual values of each grid in the occupancy map
show(map)
%initialize starting location
up_dir = '^r'; %indicator for robot heading direction on map
down_dir = 'vr'; %indicator for robot heading direction on map
left_dir = '<r';
right_dir = '>r';
hold on
startloc = [3,2]; %starting grid location
xy = grid2world(map,[startloc(1),startloc(2)]); %convert starting grid location to xy coordinates
plot(xy(1), xy(2), down_dir) %plot starting location xy coordinates on map

pc = startloc; %initialize starting grid location as current position

%go = input('press 1 to advance... '); %manually simulating motion (for debugging purposes)


go = 1; %waitforbuttonpress; %CHANGEE!!!
tic
```

```
 while go == 1
 %for go = 1:500
 %Building neural network of map
 for j = 1:1:map.GridSize(2) %{Reference; j is cols (goes left to right)}
    for i = 1:1:map.GridSize(1)  %{Reference; i is rows (goes up and down)}
        ab = grid2world(map, [i,j]); %current grid position for neural activity calculation
        status = checkOccupancy(map, [ab(1),ab(2)]); %check status of current grid position
        %convert grid values to 'I' values
        if status == -1
            I = 100;
        elseif status == 0
            I = 0;
        else
            I = -100;
        end
        I_plus = max([I 0]);
        I_neg = max([-1*I 0]);

        %check neural activity level of neighboring neurons, evaluate using ReLu and calculate the
Euclidean distance
        try
            North = max([n(i-1, j) 0]); %North = max([n(i, j-1) 0]);
            dNorth = 0.7/((norm(ab - grid2world(map, [i-1, j])))); %dNorth = norm(ab -
grid2world(map, [i, j-1]));
        catch
            North = 0;
            dNorth = 0;
        end

        try
            South = max([n(i+1, j) 0]); %South = max([n(i, j+1) 0]);
            dSouth = 0.7/(norm(ab - grid2world(map, [i+1, j]))); %dSouth = norm(ab -
grid2world(map, [i, j+1]));
        catch
            South = 0;
            dSouth = 0;
        end

        try
            West = max([n(i, j-1) 0]); %West = max([n(i-1, j) 0]);
            dWest = 0.7/(norm(ab - grid2world(map, [i, j-1]))); %dWest = norm(ab -
grid2world(map, [i-1, j]));
        catch
            West = 0;
            dWest = 0;
        end

        try
            East = max([n(i, j+1) 0]); %East = max([n(i+1, j) 0]);
            dEast = 0.7/(norm(ab - grid2world(map, [i, j+1]))); %dEast = norm(ab - grid2world(map,
[i+1, j]));
        catch
            East = 0;
            dEast = 0;
```

```
    end

    try
       NW = max([n(i-1, j-1) 0]); %NW = max([n(i-1, j-1) 0]);
       dNW = 0.7/(norm(ab - grid2world(map, [i-1, j-1]))); %dNW = norm(ab - grid2world(map, [i-1, j-1]));
    catch
       NW = 0;
       dNW = 0;
    end

    try
       NE = max([n(i-1, j+1) 0]); %NE = max([n(i+1, j-1) 0]);
       dNE = 0.7/(norm(ab - grid2world(map, [i-1, j+1]))); %dNE = norm(ab - grid2world(map, [i+1, j-1]));
    catch
       NE = 0;
       dNE = 0;
    end

    try
       SW = max([n(i+1, j-1) 0]); %SW = max([n(i-1, j+1) 0]);
       dSW = 0.7/(norm(ab - grid2world(map, [i+1, j-1]))); %dSW = norm(ab - grid2world(map, [i-1, j+1]));
    catch
       SW = 0;
       dSW = 0;
    end

    try
       SE = max([n(i+1, j+1) 0]); %SE = max([n(i+1, j+1) 0]);
       dSE = 0.7/(norm(ab - grid2world(map, [i+1, j+1]))); %dSE = norm(ab - grid2world(map, [i+1, j+1]));
    catch
       SE = 0;
       dSE = 0;
    end


    %store evaluated neural activity values for neighboring neurons (8x1 matrix/array)
    xplus = [North, South, West, East, NW, NE, SW, SE];

    %store euclidean distances (1x8 matrix/array)
    wij = [dNorth; dSouth; dWest; dEast; dNW; dNE; dSW; dSE];

    %calculate the weight
    weight = xplus * wij; % Matrix operation calculating for the weight (order of operation should not be changed)

    %set variables
    A = 80;
    B = 1;
    D = 1;
    %define & evaluate equations using ODE solver
```

```
        eqn = @(t,Xi) ((-A*Xi) + (B-Xi)*(I_plus + weight) - (D+Xi)*I_neg);
        [t,Xi] = ode45(eqn, [0:1], n(i,j));
        n(i,j) = Xi(end);

%         if Xi(end) < 0
%             n(i,j) = -0.6;%min(Xi);
%         elseif Xi(end) > 0.5
%             n(i,j) = 0.6;%max(Xi);
%         elseif (Xi(end) > 0) && (Xi(end) < 0.5)
%             n(i,j) = 0;%min(Xi);
%         end
    end
end


%check if map/environment is completely covered and end operation
if sum(n>=0.5, 'all') == 0
    go = 0;
end

%update status of current grid if map/environment is not completely covered
setOccupancy(map,pc,0,'grid')
pause(0.005)
show(map)


%--------------------PATH SELECTION ALGORITHM-----------------------------%
%Heuristic sequence
%left -> bottom -> top -> right -> right bottom -> right top


%Find all possible next positions
left = n(pc(1), pc(2)-1); %left as the next position
bottom = n(pc(1)+1, pc(2)); %bottom as the next position
top = n(pc(1)-1, pc(2)); %top as the next position
right = n(pc(1), pc(2)+1); %right as the next position
right_bottom = n(pc(1)+1, pc(2)+1); %right bottom as the next position
right_top = n(pc(1)-1, pc(2)+1); %right top as the next position


pn =[left, bottom, top, right, right_bottom, right_top]; %don't change order of this array

%check the status of all possible next locations
for value = 1:1:length(pn)
    if pn(value) < 0
        pn(value) = -1; %indicate cell is obstacle
    elseif pn(value) > 0.5
        pn(value) = 1; %indicate cell is uncovered
    elseif (pn(value) > 0) && (pn(value) < 0.5)
        pn(value) = 0; %indicate cell is covered
    end
end
```

```
if sum(pn==1) == 0 %check to for deadlock event
    %search for uncovered locations in closest neighbors using neural propagation
    kN = n(pc(1)-1, pc(2)) ;
    kS = n(pc(1)+1, pc(2)) ;
    kW = n(pc(1), pc(2)-1) ;
    kE = n(pc(1), pc(2)+1) ;
    kNW = n(pc(1)-1, pc(2)-1) ;
    kNE = n(pc(1)-1, pc(2)+1) ;
    kSW = n(pc(1)+1, pc(2)-1) ;
    kSE = n(pc(1)+1, pc(2)+1) ;
    search = [kN,kNE,kE,kSE,kS,kSW,kW,kNW];
    switch max(search)
        case search(1)
            pc = [pc(1)-1, pc(2)];
            gpc = grid2world(map, pc);
            %disp('north is next')
            plot(gpc(1), gpc(2),up_dir)
        case search(2)
            pc = [pc(1)-1, pc(2)+1];
            gpc = grid2world(map, pc);
            %disp('ne is next')
            plot(gpc(1), gpc(2),up_dir)
        case search(3)
            pc = [pc(1), pc(2)+1];
            gpc = grid2world(map, pc);
            %disp('east is next')
            plot(gpc(1), gpc(2),right_dir)
        case search(4)
            pc = [pc(1)+1, pc(2)+1];
            gpc = grid2world(map, pc);
            %disp('se is next')
            plot(gpc(1), gpc(2),down_dir)
        case search(5)
            pc = [pc(1)+1, pc(2)];
            gpc = grid2world(map, pc);
            %disp('south is next')
            plot(gpc(1), gpc(2),down_dir)
        case search(6)
            pc = [pc(1)+1, pc(2)-1];
            gpc = grid2world(map, pc);
            %disp('sw is next')
            plot(gpc(1), gpc(2),down_dir)
        case search(7)
            pc = [pc(1), pc(2)-1];
            gpc = grid2world(map, pc);
            %disp('west is next')
            plot(gpc(1), gpc(2),left_dir)
        case search(8)
            pc = [pc(1)-1, pc(2)-1];
            gpc = grid2world(map, pc);
            %disp('nw is next')
            plot(gpc(1), gpc(2),up_dir)
    end
```

```
     %once uncovered location is found revert back to heuristic path selection
else %continue on with heuristic path selection
   switch max(pn)
      case pn(1)
         pc = [pc(1), pc(2)-1];
         gpc = grid2world(map, pc);
         %disp('left is next')
         plot(gpc(1), gpc(2),left_dir)
      case pn(2)
         pc = [pc(1)+1, pc(2)];
         gpc = grid2world(map, pc);
         %disp('bottom is next')
         plot(gpc(1), gpc(2),down_dir)
      case pn(3)
         pc = [pc(1)-1, pc(2)];
         gpc = grid2world(map, pc);
         %disp('top is next')
         plot(gpc(1), gpc(2),up_dir)
      case pn(4)
         pc = [pc(1), pc(2)+1];
         gpc = grid2world(map, pc);
         %disp('right is next')
         plot(gpc(1), gpc(2),right_dir)
      case pn(5)
         pc = [pc(1)+1, pc(2)+1];
         gpc = grid2world(map, pc);
         %disp('right bottom is next')
         plot(gpc(1), gpc(2),down_dir)
      case pn(6)
         pc = [pc(1)-1, pc(2)+1];
         gpc = grid2world(map, pc);
         %disp('right top is next')
         plot(gpc(1), gpc(2),up_dir)
   end
end

mat = occupancyMatrix(map); %update grid values of the map
%mesh(n)
% Stop condition


end
toc
```

A-2: MATLAB Simulation Code with PRM

This program assumes an unknown map state. Code prompts the user to input a selection of available Excel grid maps. Generates a neural activity matrix of the same size as the selected map. Initializes the neural activity matrix to all zeros. Utilizes a rolling window to update neural activity level for each grid and uses the heuristic method for path selection.

```
clear
clc


select = input('1-Rectangle obstacle, 2-Circular, 3-Generic, 4-100 park map: ');

if select == 1
   env = readmatrix("map builder.xlsx", "Sheet","Rectangle obstacle");
elseif select == 2
   env = readmatrix("map builder.xlsx", "Sheet","Circular obstacle");
elseif select == 3
   env = readmatrix("map builder.xlsx", "Sheet","test");
elseif select == 4
   env = readmatrix("map builder.xlsx", "Sheet","100 park");
end

map = occupancyMap(env);
% map = occupancyMap(env,50);
% map = occupancyMap(map,11);
% map = occupancyMap(map,2);
show(map)

mat = occupancyMatrix(map); %an array containing the actual values of each grid in the occupancy map


map_prm = occupancyMap(env);
% map_prm = occupancyMap(env,50);
% map_prm = occupancyMap(map_prm,11);
% map_prm = occupancyMap(map_prm,2);

n = zeros(map.GridSize(1), map.GridSize(2)); %generate array to store neural activity values

%initialize starting location
up_dir = '^r'; %indicator for robot heading direction on map [0,0]
down_dir = 'vr'; %indicator for robot heading direction on map [1,1]
left_dir = '<r'; %[1,0]
right_dir = '>r'; %[0,1]
hold on
startloc = [8,2]; %starting grid location
xy = grid2world(map,[startloc(1),startloc(2)]); %convert starting grid location to xy coordinates
```

```
plot(xy(1), xy(2), up_dir) %plot starting location xy coordinates on map

pc = startloc; %initialize starting grid location as current position
count = 0;

prev_Orient = 0;
turnCount = 0;
path_length = 0;
overlap = 0;
pp = pc;

% prm = mobileRobotPRM;
% prm.Map = map_prm;
% prm.NumNodes = 100;
% prm.ConnectionDistance = 10;
% path = findpath(prm,[3.75,5.75],[5.75,3.75])
% show(prm)
go = waitforbuttonpress; %CHANGEE!!!
tic
while go == 1
%for go = 1:500
%Building neural network of map
for j = pc(2)-1:1:pc(2)+1 %{Reference; j is cols (goes left to right)}
   for i = pc(1)-1:1:pc(1)+1  %{Reference; i is rows (goes up and down)}
      ab = grid2world(map, [i,j]); %current grid position for neural activity calculation
      status = checkOccupancy(map, [ab(1),ab(2)]); %check status of current grid position
      %convert grid values to 'I' values
      if status == -1
         I = 100; %change to 0 if using '100 park' map
      elseif status == 0
         I = 00; %change to 100 if using '100 park' map
      else
         I = -100;
      end
      I_plus = max([I 0]);
      I_neg = max([-1*I 0]);

      %check neural activity level of neighboring neurons, evaluate using ReLu and calculate Euclidean distance
      try
         North = max([n(i-1, j) 0]); %North = max([n(i, j-1) 0]);
         dNorth = 0.7/((norm(ab - grid2world(map, [i-1, j])))); %dNorth = norm(ab - grid2world(map, [i, j-1]));
      catch
         North = 0;
         dNorth = 0;
      end

      try
         South = max([n(i+1, j) 0]); %South = max([n(i, j+1) 0]);
         dSouth = 0.7/(norm(ab - grid2world(map, [i+1, j]))); %dSouth = norm(ab - grid2world(map, [i, j+1]));
      catch
         South = 0;
```

```
            dSouth = 0;
        end

        try
            West = max([n(i, j-1) 0]); %West = max([n(i-1, j) 0]);
            dWest = 0.7/(norm(ab - grid2world(map, [i, j-1]))); %dWest = norm(ab - grid2world(map, [i-1, j]));
        catch
            West = 0;
            dWest = 0;
        end

        try
            East = max([n(i, j+1) 0]); %East = max([n(i+1, j) 0]);
            dEast = 0.7/(norm(ab - grid2world(map, [i, j+1]))); %dEast = norm(ab - grid2world(map, [i+1, j]));
        catch
            East = 0;
            dEast = 0;
        end

        try
            NW = max([n(i-1, j-1) 0]); %NW = max([n(i-1, j-1) 0]);
            dNW = 0.7/(norm(ab - grid2world(map, [i-1, j-1]))); %dNW = norm(ab - grid2world(map, [i-1, j-1]));
        catch
            NW = 0;
            dNW = 0;
        end

        try
            NE = max([n(i-1, j+1) 0]); %NE = max([n(i+1, j-1) 0]);
            dNE = 0.7/(norm(ab - grid2world(map, [i-1, j+1]))); %dNE = norm(ab - grid2world(map, [i+1, j-1]));
        catch
            NE = 0;
            dNE = 0;
        end

        try
            SW = max([n(i+1, j-1) 0]); %SW = max([n(i-1, j+1) 0]);
            dSW = 0.7/(norm(ab - grid2world(map, [i+1, j-1]))); %dSW = norm(ab - grid2world(map, [i-1, j+1]));
        catch
            SW = 0;
            dSW = 0;
        end

        try
            SE = max([n(i+1, j+1) 0]); %SE = max([n(i+1, j+1) 0]);
            dSE = 0.7/(norm(ab - grid2world(map, [i+1, j+1]))); %dSE = norm(ab - grid2world(map, [i+1, j+1]));
        catch
            SE = 0;
```

```
            dSE = 0;
        end


        %store evaluated neural activity values for neighboring neurons (8x1 matrix/array)
        xplus = [North, South, West, East, NW, NE, SW, SE];

        %store euclidean distances (1x8 matrix/array)
        wij = [dNorth; dSouth; dWest; dEast; dNW; dNE; dSW; dSE];

        %calculate the weight
        weight = xplus * wij; % Matrix operation calculating for the weight (order of operation should not be changed)

        %set variables
        A = 30;
        B = 1;
        D = 1;
        %define & evaluate equations using ODE solver
        eqn = @(t,Xi) ((-A*Xi) + (B-Xi)*(I_plus + weight) - (D+Xi)*I_neg);
        [t,Xi] = ode45(eqn, [0:1], n(i,j));
        n(i,j) = Xi(end);

%       if Xi(end) < 0
%           n(i,j) = -0.6;%min(Xi);2

%       elseif Xi(end) > 0.5
%           n(i,j) = 0.6;%max(Xi);
%       elseif (Xi(end) > 0) && (Xi(end) < 0.5)
%           n(i,j) = 0;%min(Xi);
%       end
    end
end


%check if map/environment is completely covered and end operation
if sum(n>=0.5, 'all') == 0
    go = 0;
    break
end

%update status of current grid if map/environment is not completely covered
setOccupancy(map,pc,0,'grid')
pause(0.005)
show(map)


%--------------------PATH SELECTION ALGORITHM------------------------------%
%Heuristic sequence
%left -> bottom -> top -> right -> right bottom -> right top


%list all possible next positions
left = n(pc(1), pc(2)-1); %left as the next position
```

```
bottom = n(pc(1)+1, pc(2)); %bottom as the next position
top = n(pc(1)-1, pc(2)); %top as the next position
right = n(pc(1), pc(2)+1); %right as the next position
right_bottom = n(pc(1)+1, pc(2)+1); %right bottom as the next position
right_top = n(pc(1)-1, pc(2)+1); %right top as the next position

pn =[left, bottom, top, right, right_bottom, right_top]; %don't change order of this array


%pn = [kNW,kW,kSW,kS,kN,kNE,kE,kSE];

%check the status of all possible next locations
for value = 1:1:length(pn)
   if pn(value) < 0
      pn(value) = -1; %indicate cell is obstacle
   elseif pn(value) > 0.5
      pn(value) = 1; %indicate cell is uncovered
   elseif (pn(value) > 0) && (pn(value) < 0.5)
      pn(value) = 0; %indicate cell is covered
   end
end



if sum(pn==1) == 0 %check to for deadlock event
%     count = count + 1;
%     if count == 20
% %   pc = runPRM(map,n,pc);
%     [row,col] = find(n>0.5);
%     startlocation = grid2world(map,pc);
%     endloc = grid2world(map,[row(1), col(1)]);
%     prm = mobileRobotPRM;
%     prm.Map = map_prm;
%     prm.NumNodes = 30;
%     prm.ConnectionDistance = 10;
%     path = findpath(prm,startlocation,endloc);
%     for i = 1:1:length(path)
%        grid_prm_path = world2grid(map,[path(i,1), path(i,2)]);
%        world_prm_path = grid2world(map, grid_prm_path);
%
%        setOccupancy(map,grid_prm_path,0.5,'grid')
%        pause(1)
%        show(map)
%
%        if i == 1
%           pose = up_dir;
%        elseif double([path(i-1,1), path(i-1,2)] > [path(i,1), path(i,2)]) == [0,0]
%           pose = up_dir;
%        elseif double([path(i-1,1), path(i-1,2)] > [path(i,1), path(i,2)]) == [0,1]
%           pose = right_dir;
%        elseif double([path(i-1,1), path(i-1,2)] > [path(i,1), path(i,2)]) == [1,0]
%           pose = left_dir;
%        elseif double([path(i-1,1), path(i-1,2)] > [path(i,1), path(i,2)]) == [1,1]
%           pose = down_dir;
```

```
%         end
%
%         plot(world_prm_path(1), world_prm_path(2),pose)
%
%     end
%     pc = [row(1), col(1)];
%     count = 0;
%     end

    %search for uncovered location in closest neighbors using neural propagation
    kN = n(pc(1)-1, pc(2)) ;
    kS = n(pc(1)+1, pc(2)) ;
    kW = n(pc(1), pc(2)-1) ;
    kE = n(pc(1), pc(2)+1) ;
    kNW = n(pc(1)-1, pc(2)-1) ;
    kNE = n(pc(1)-1, pc(2)+1) ;
    kSW = n(pc(1)+1, pc(2)-1) ;
    kSE = n(pc(1)+1, pc(2)+1) ;
%     Kp = n(pc(1),pc(2));
    search = [kN,kNE,kE,kSE,kS,kSW,kW,kNW];
    switch max(search)
%         case search(1)
%             pc = [pc(1), pc(2)];
%             gpc = grid2world(map, pc);
%             %disp('remain in current position')
%             plot(gpc(1), gpc(2),up_dir)

        case search(1)
            pc = [pc(1)-1, pc(2)];
            gpc = grid2world(map, pc);
            %disp('north is next')
            plot(gpc(1), gpc(2),up_dir)
            orientation = 1;

        case search(2)
            pc = [pc(1)-1, pc(2)+1];
            gpc = grid2world(map, pc);
            %disp('ne is next')
            plot(gpc(1), gpc(2),up_dir)
            orientation = 2;

        case search(3)
            pc = [pc(1), pc(2)+1];
            gpc = grid2world(map, pc);
            %disp('east is next')
            plot(gpc(1), gpc(2),right_dir)
            orientation = 3;

        case search(4)
            pc = [pc(1)+1, pc(2)+1];
            gpc = grid2world(map, pc);
            %disp('se is next')
            plot(gpc(1), gpc(2),down_dir)
            orientation = 4;
```

```
        case search(5)
            pc = [pc(1)+1, pc(2)];
            gpc = grid2world(map, pc);
            %disp('south is next')
            plot(gpc(1), gpc(2),down_dir)
            orientation = 5;

        case search(6)
            pc = [pc(1)+1, pc(2)-1];
            gpc = grid2world(map, pc);
            %disp('sw is next')
            plot(gpc(1), gpc(2),down_dir)
            orientation = 6;

        case search(7)
            pc = [pc(1), pc(2)-1];
            gpc = grid2world(map, pc);
            %disp('west is next')
            plot(gpc(1), gpc(2),left_dir)
            orientation = 7;

        case search(8)
            pc = [pc(1)-1, pc(2)-1];
            gpc = grid2world(map, pc);
            %disp('nw is next')
            plot(gpc(1), gpc(2),up_dir)
            orientation = 8;

    end

     %once uncovered location is found revert back to heuristic path selection
else %continue on with heuristic path selection
    switch max(pn)
        case pn(1)
            pc = [pc(1), pc(2)-1];
            gpc = grid2world(map, pc);
            %disp('left is next')
            plot(gpc(1), gpc(2),left_dir)
            count = 0;
            orientation = 7;
        case pn(2)
            pc = [pc(1)+1, pc(2)];
            gpc = grid2world(map, pc);
            %disp('bottom is next')
            plot(gpc(1), gpc(2),down_dir)
            count = 0;
            orientation = 5;
        case pn(3)
            pc = [pc(1)-1, pc(2)];
            gpc = grid2world(map, pc);
            %disp('top is next')
            plot(gpc(1), gpc(2),up_dir)
            count = 0;
```

```
            orientation = 1;
        case pn(4)
            pc = [pc(1), pc(2)+1];
            gpc = grid2world(map, pc);
            %disp('right is next')
            plot(gpc(1), gpc(2),right_dir)
            count = 0;
            orientation = 3;
        case pn(5)
            pc = [pc(1)+1, pc(2)+1];
            gpc = grid2world(map, pc);
            %disp('right bottom is next')
            plot(gpc(1), gpc(2),down_dir)
            count = 0;
            orientation = 4;
        case pn(6)
            pc = [pc(1)-1, pc(2)+1];
            gpc = grid2world(map, pc);
            %disp('right top is next')
            plot(gpc(1), gpc(2),up_dir)
            count = 0;
            orientation = 2;
    end
end

% count number of turns
if prev_Orient ~= orientation
    turnCount = turnCount + 1;
    prev_Orient = orientation;
end


% count number of overlaps
if checkOccupancy(map, pc, "grid") == 0
    overlap = overlap + 1;
end


path_dist = norm(grid2world(map, pc) - grid2world(map, pp));

path_length = path_length + path_dist; %calculate path length data

pp = pc;




mat = occupancyMatrix(map); %update grid values of the map

%mesh(n)
```

```
% Stop condition




end


toc
```

A-3: MATLAB Simulation Code for Obstacle free map

```
clear
clc

l = 10; %set length (# of rows) of grid map
w = 10; %set width (# of columns) of grid map
%res = 2;  %define number of cells per meter for grid map ( sets resolution)


map = occupancyMap(l,w,"grid"); %generate grid map dimensions


%set map boundaries
for x = 1:1:map.GridSize(1)
    setOccupancy(map,[1,x],1,"grid")
    setOccupancy(map,[x,1],1,"grid")
    setOccupancy(map,[map.GridSize(1),x],1,"grid")
    setOccupancy(map,[x,map.GridSize(2)],1,"grid")

    pause(0.005)
    show(map)
end

mat = occupancyMatrix(map); %an array containing the actual values of each grid in the occupancy map
n = zeros(map.GridSize(1), map.GridSize(2)); %generate array to store neural activity values
% clear
% clc
%
% env = zeros(20,20);
% map = occupancyMap(env);
% n = zeros(map.GridSize(1), map.GridSize(2)); %generate array to store neural activity values
% mat = occupancyMatrix(map); %an array containing the actual values of each grid in the occupancy map
% show(map)
%initialize starting location
up = '^r'; %indicator for robot heading direction on map
down = 'vr'; %indicator for robot heading direction on map
left = '<r';
right = '>r';
hold on
startloc = [2,2]; %starting grid location
xy = grid2world(map,[startloc(1),startloc(2)]); %convert starting grid location to xy coordinates
plot(xy(1), xy(2), down) %plot starting location xy coordinates on map

pc = startloc; %initialize starting grid location as current position

%go = input('press 1 to advance... '); %manually simulating motion (for debugging purposes)


go = waitforbuttonpress; %CHANGEE!!!
```

```
 while go == 1
 %for go = 1:500
 %Building neural network of map
 for j = 1:1:map.GridSize(2) %{Reference; j is cols (goes left to right)}
    for i = 1:1:map.GridSize(1)  %{Reference; i is rows (goes up and down)}
        ab = grid2world(map, [i,j]); %current grid position for neural activity calculation
        status = checkOccupancy(map, [ab(1),ab(2)]); %check status of current grid position
        %convert grid values to 'I' values
        if status == -1
            I = 100;
        elseif status == 0
            I = 0;
        else
            I = -100;
        end
        I_plus = max([I 0]);
        I_neg = max([-1*I 0]);

        %check neural activity level of neighboring neurons, evaluate using ReLu, and calcuate
euclidean distance
        try
            North = max([n(i-1, j) 0]); %North = max([n(i, j-1) 0]);
            dNorth = 0.7/((norm(ab - grid2world(map, [i-1, j])))); %dNorth = norm(ab -
grid2world(map, [i, j-1]));
        catch
            North = 0;
            dNorth = 0;
        end

        try
            South = max([n(i+1, j) 0]); %South = max([n(i, j+1) 0]);
            dSouth = 0.7/(norm(ab - grid2world(map, [i+1, j]))); %dSouth = norm(ab -
grid2world(map, [i, j+1]));
        catch
            South = 0;
            dSouth = 0;
        end

        try
            West = max([n(i, j-1) 0]); %West = max([n(i-1, j) 0]);
            dWest = 0.7/(norm(ab - grid2world(map, [i, j-1]))); %dWest = norm(ab -
grid2world(map, [i-1, j]));
        catch
            West = 0;
            dWest = 0;
        end

        try
            East = max([n(i, j+1) 0]); %East = max([n(i+1, j) 0]);
            dEast = 0.7/(norm(ab - grid2world(map, [i, j+1]))); %dEast = norm(ab - grid2world(map,
[i+1, j]));
        catch
            East = 0;
            dEast = 0;
```

```
    end

    try
       NW = max([n(i-1, j-1) 0]); %NW = max([n(i-1, j-1) 0]);
       dNW = 0.7/(norm(ab - grid2world(map, [i-1, j-1]))); %dNW = norm(ab - grid2world(map, [i-1, j-1]));
    catch
       NW = 0;
       dNW = 0;
    end

    try
       NE = max([n(i-1, j+1) 0]); %NE = max([n(i+1, j-1) 0]);
       dNE = 0.7/(norm(ab - grid2world(map, [i-1, j+1]))); %dNE = norm(ab - grid2world(map, [i+1, j-1]));
    catch
       NE = 0;
       dNE = 0;
    end

    try
       SW = max([n(i+1, j-1) 0]); %SW = max([n(i-1, j+1) 0]);
       dSW = 0.7/(norm(ab - grid2world(map, [i+1, j-1]))); %dSW = norm(ab - grid2world(map, [i-1, j+1]));
    catch
       SW = 0;
       dSW = 0;
    end

    try
       SE = max([n(i+1, j+1) 0]); %SE = max([n(i+1, j+1) 0]);
       dSE = 0.7/(norm(ab - grid2world(map, [i+1, j+1]))); %dSE = norm(ab - grid2world(map, [i+1, j+1]));
    catch
       SE = 0;
       dSE = 0;
    end


    %store evaluated neural activity values for neighboring neurons (8x1 matrix/array)
    xplus = [North, South, West, East, NW, NE, SW, SE];

    %store euclidean distances (1x8 matrix/array)
    wij = [dNorth; dSouth; dWest; dEast; dNW; dNE; dSW; dSE];

    %calculate the weight
    weight = xplus * wij; % Matrix operation calculating for the weight (order of operation should not be changed)

    %set variables
    A = 80;
    B = 1;
    D = 1;
    %define & evaluate equation using ODE solver
```

```
        eqn = @(t,Xi) ((-A*Xi) + (B-Xi)*(I_plus + weight) - (D+Xi)*I_neg);
        [t,Xi] = ode45(eqn, [0:1], n(i,j));
        n(i,j) = Xi(end);
%         if Xi(end) < 0
%             n(i,j) = -0.6;%min(Xi);
%         elseif Xi(end) > 0.6
%             n(i,j) = 0.6;%max(Xi);
%         elseif (Xi(end) > 0) && (Xi(end) < 0.6)
%             n(i,j) = 0;%min(Xi);
%         end
    end
end

%mesh(n)



if pc == startloc %if current position eqauls starting position
    gpc = grid2world(map, pc); %convert current location grid index to global xy values
    pp = [1,2]; %set previous position
    gpp = grid2world(map, pp); %convert previous location grid index to global xy values

    %Find all possible next positions
    pjN = grid2world(map, [pc(1)-1, pc(2)]); %North as the next position
    pjS = grid2world(map, [pc(1)+1, pc(2)]); %South as the next position
    pjW = grid2world(map, [pc(1), pc(2)-1]); %West as the next position
    pjE = grid2world(map, [pc(1), pc(2)+1]); %East as the next position
    pjNW = grid2world(map, [pc(1)-1, pc(2)-1]); %NW as the next position
    pjNE = grid2world(map, [pc(1)-1, pc(2)+1]); %NE as the next position
    pjSW = grid2world(map, [pc(1)+1, pc(2)-1]); %SW as the next position
    pjSE = grid2world(map, [pc(1)+1, pc(2)+1]); %SE as the next position

    %Evaluate for next position
    %next - current, current - previous
    %|atan2(ypj − ypc ,xpj −xpc ) − atan2(ypc − ypp ,xpc − xpp )|
    c = 0.5;

    r = abs(atan2(pjN(2)-xy(2),pjN(1)-xy(1)) - atan2(gpc(2)-gpp(2),gpc(1)-gpp(1)));
    yjN = ((1 - (r/pi)));
    kN = n(pc(1)-1, pc(2)) + c*yjN;

    r1 = abs(atan2(pjS(2)-xy(2),pjS(1)-xy(1)) - atan2(gpc(2)-gpp(2),gpc(1)-gpp(1)));
    yjS = ((1 - (r1/pi)));
    kS = n(pc(1)+1, pc(2)) + c*yjS;

    r2 = abs(atan2(pjW(2)-xy(2),pjW(1)-xy(1)) - atan2(gpc(2)-gpp(2),gpc(1)-gpp(1)));
    yjW = ((1 - (r2/pi)));
    kW = n(pc(1), pc(2)-1) + c*yjW;

    r3 = abs(atan2(pjE(2)-xy(2),pjE(1)-xy(1)) - atan2(gpc(2)-gpp(2),gpc(1)-gpp(1)));
    yjE = ((1 - (r3/pi)));
    kE = n(pc(1), pc(2)+1) + c*yjE;

    r4 = abs(atan2(pjNW(2)-xy(2),pjNW(1)-xy(1)) - atan2(gpc(2)-gpp(2),gpc(1)-gpp(1)));
```

```
yjNW = ((1 - (r4/pi)));
kNW = n(pc(1)-1, pc(2)-1) + c*yjNW;

r5 = abs(atan2(pjNE(2)-xy(2),pjNE(1)-xy(1)) - atan2(gpc(2)-gpp(2),gpc(1)-gpp(1)));
yjNE = ((1 - (r5/pi)));
kNE = n(pc(1)-1, pc(2)+1) + c*yjNE;

r6 = abs(atan2(pjSW(2)-xy(2),pjSW(1)-xy(1)) - atan2(gpc(2)-gpp(2),gpc(1)-gpp(1)));
yjSW = ((1 - (r6/pi)));
kSW = n(pc(1)+1, pc(2)-1) + c*yjSW;

r7 = abs(atan2(pjSE(2)-xy(2),pjSE(1)-xy(1)) - atan2(gpc(2)-gpp(2),gpc(1)-gpp(1)));
yjSE = ((1 - (r7/pi)));
kSE = n(pc(1)+1, pc(2)+1) + c*yjSE;

pn = [kN,kS,kW,kE,kNW,kNE,kSW,kSE]; %do not change the order of this array!!!
setOccupancy(map,gpc,0)
show(map)
pp = pc;
gpp = grid2world(map, pp);
switch max(pn)
   case pn(1)
      pc = [pc(1)-1, pc(2)];
      gpc = grid2world(map, pc);
      %disp('north is next')
      plot(gpc(1), gpc(2),up)
   case pn(2)
      pc = [pc(1)+1, pc(2)];
      gpc = grid2world(map, pc);
      %disp('south is next')
      plot(gpc(1), gpc(2),down)
   case pn(3)
      pc = [pc(1), pc(2)-1];
      gpc = grid2world(map, pc);
      %disp('west is next')
      plot(gpc(1), gpc(2),left)
   case pn(4)
      pc = [pc(1), pc(2)+1];
      gpc = grid2world(map, pc);
      %disp('east is next')
      plot(gpc(1), gpc(2),right)
   case pn(5)
      pc = [pc(1)-1, pc(2)-1];
      gpc = grid2world(map, pc);
      %disp('nw is next')
      plot(gpc(1), gpc(2),up)
   case pn(6)
      pc = [pc(1)-1, pc(2)+1];
      gpc = grid2world(map, pc);
      %disp('ne is next')
      plot(gpc(1), gpc(2),up)
   case pn(7)
      pc = [pc(1)+1, pc(2)-1];
      gpc = grid2world(map, pc);
```

```
            %disp('sw is next')
            plot(gpc(1), gpc(2),down)
        case pn(8)
            pc = [pc(1)+1, pc(2)+1];
            gpc = grid2world(map, pc);
            %disp('se is next')
            plot(gpc(1), gpc(2),down)
    end

    %pc = pn;
    %gpc = grid2world(map, pc);
else
    %Find all possible next positions
    pjN = grid2world(map, [pc(1)-1, pc(2)]); %North as the next position
    pjS = grid2world(map, [pc(1)+1, pc(2)]); %South as the next position
    pjW = grid2world(map, [pc(1), pc(2)-1]); %West as the next position
    pjE = grid2world(map, [pc(1), pc(2)+1]); %East as the next position
    pjNW = grid2world(map, [pc(1)-1, pc(2)-1]); %NW as the next position
    pjNE = grid2world(map, [pc(1)-1, pc(2)+1]); %NE as the next position
    pjSW = grid2world(map, [pc(1)+1, pc(2)-1]); %SW as the next position
    pjSE = grid2world(map, [pc(1)+1, pc(2)+1]); %SE as the next position

    %Evaluate for next position
    %next - current, current - previous
    %|atan2(ypj − ypc ,xpj −xpc ) − atan2(ypc − ypp ,xpc − xpp )|

    r = abs(atan2(pjN(2)-gpc(2),pjN(1)-gpc(1)) - atan2(gpc(2)-gpp(2),gpc(1)-gpp(1)));
    yjN = ((1 - (r/pi)));
    kN = n(pc(1)-1, pc(2)) + c*yjN;

    r1 = abs(atan2(pjS(2)-gpc(2),pjS(1)-gpc(1)) - atan2(gpc(2)-gpp(2),gpc(1)-gpp(1)));
    yjS = ((1 - (r1/pi)));
    kS = n(pc(1)+1, pc(2)) + c*yjS;

    r2 = abs(atan2(pjW(2)-gpc(2),pjW(1)-gpc(1)) - atan2(gpc(2)-gpp(2),gpc(1)-gpp(1)));
    yjW = ((1 - (r2/pi)));
    kW = n(pc(1), pc(2)-1) + c*yjW;

    r3 = abs(atan2(pjE(2)-gpc(2),pjE(1)-gpc(1)) - atan2(gpc(2)-gpp(2),gpc(1)-gpp(1)));
    yjE = ((1 - (r3/pi)));
    kE = n(pc(1), pc(2)+1) + c*yjE;

    r4 = abs(atan2(pjNW(2)-gpc(2),pjNW(1)-gpc(1)) - atan2(gpc(2)-gpp(2),gpc(1)-gpp(1)));
    yjNW = ((1 - (r4/pi)));
    kNW = n(pc(1)-1, pc(2)-1) + c*yjNW;

    r5 = abs(atan2(pjNE(2)-gpc(2),pjNE(1)-gpc(1)) - atan2(gpc(2)-gpp(2),gpc(1)-gpp(1)));
    yjNE = ((1 - (r5/pi)));
    kNE = n(pc(1)-1, pc(2)+1) + c*yjNE;

    r6 = abs(atan2(pjSW(2)-gpc(2),pjSW(1)-gpc(1)) - atan2(gpc(2)-gpp(2),gpc(1)-gpp(1)));
    yjSW = ((1 - (r6/pi)));
    kSW = n(pc(1)+1, pc(2)-1) + c*yjSW;
```

```
r7 = abs(atan2(pjSE(2)-gpc(2),pjSE(1)-gpc(1)) - atan2(gpc(2)-gpp(2),gpc(1)-gpp(1)));
yjSE = ((1 - (r7/pi)));
kSE = n(pc(1)+1, pc(2)+1) + c*yjSE;

pn = [kN,kS,kW,kE,kNW,kNE,kSW,kSE]; %do not change the order of this array!!!

pp = pc;
gpp = grid2world(map, pp);
setOccupancy(map,gpp,0)
pause(0.005)
show(map)

switch max(pn)
   case pn(1)
      pc = [pc(1)-1, pc(2)];
      gpc = grid2world(map, pc);
      %disp('north is next')
      plot(gpc(1), gpc(2),up)
   case pn(2)
      pc = [pc(1)+1, pc(2)];
      gpc = grid2world(map, pc);
      %disp('south is next')
      plot(gpc(1), gpc(2),down)
   case pn(3)
      pc = [pc(1), pc(2)-1];
      gpc = grid2world(map, pc);
      %disp('west is next')
      plot(gpc(1), gpc(2),left)

   case pn(4)
      pc = [pc(1), pc(2)+1];
      gpc = grid2world(map, pc);
      %disp('east is next')
      plot(gpc(1), gpc(2),right)
   case pn(5)
      pc = [pc(1)-1, pc(2)-1];
      gpc = grid2world(map, pc);
      %disp('nw is next')
      plot(gpc(1), gpc(2),up)
   case pn(6)
      pc = [pc(1)-1, pc(2)+1];
      gpc = grid2world(map, pc);
      %disp('ne is next')
      plot(gpc(1), gpc(2),up)
   case pn(7)
      pc = [pc(1)+1, pc(2)-1];
      gpc = grid2world(map, pc);
      %disp('sw is next')
      plot(gpc(1), gpc(2),down)
   case pn(8)
      pc = [pc(1)+1, pc(2)+1];
      gpc = grid2world(map, pc);
      %disp('se is next')
      plot(gpc(1), gpc(2),down)
```

```
    end
end

mat = occupancyMatrix(map); %update grid values of the map
%mesh(n)
% Stop condition
check = find(n >= 0.5);
if isempty(check) == 1
    go = 0;
end

end
```

A-4: ROS simulation navigation code

(This program utilizes various custom functions that can be found in the GitHub repository in the link: https://github.com/victor056/CompleteCoveragePathPlanning)

ROS simulation navigation code

Connect to ROS Master

```
rosshutdown()
prompt = "ipaddress: ";
ipaddress = input(prompt, "s");
rosinit(ipaddress, 11311)
```

Clear program workspace

```
clc
clear
delete((timerfind))
```

Create ROS subscribers

```
sub.scan = rossubscriber("/scan");
sub.map = rossubscriber("/map","nav_msgs/OccupancyGrid","DataFormat","struct"); % global map
sub.coverageMap = rossubscriber("/coverage_map","nav_msgs/OccupancyGrid","DataFormat","struct");
sub.amcl = rossubscriber("/amcl_pose","geometry_msgs/PoseWithCovarianceStamped","DataFormat","struct");
```

Create ROS publishers

```
pub.map = rospublisher("/coverage_map","nav_msgs/OccupancyGrid","DataFormat","struct");
pub.cmd_vel = rospublisher("/cmd_vel","geometry_msgs/Twist","DataFormat","struct");
```

Read and display original global map published to ROS

```
pause(1)
msg.map = sub.map.LatestMessage;
pause(1)
map = rosReadOccupancyGrid(msg.map);
% show(map);
```

Acquire initial pose of the robot (2d pose estimate)

```
msg.initialpose = sub.amcl.LatestMessage;
pause(1)
pos = msg.initialpose.Pose.Pose.Position;
position.initial = [pos.X, pos.Y];
position.current = position.initial;
position.current_grid_index = world2grid(map,position.current)
```

Republish map to '/coverage_map' topic ***(optional)***

```
map_msgType = rosmessage("nav_msgs/OccupancyGrid","DataFormat","struct");
map_msg = rosWriteOccupancyGrid(map_msgType,map);
send(pub.map,map_msg)
pause(0.0002)
```

Read coverage map from '/coverage_map' topic,

```
msg.coverageMap = sub.coverageMap.LatestMessage;
map_coverage = rosReadOccupancyGrid(msg.coverageMap);
```

Acquire and store uncovered cells

```
mat = occupancyMatrix(map_coverage);
uncovered = length(find(mat <=0.5));
```

Initialize neural network matrix

```
nn_mat = zeros(map_coverage.GridSize(1), map_coverage.GridSize(2));
```

Update map to mark initial position as covered

```
mapUpdate(map_coverage, position.current_grid_index, 0.5)
```

Re-publish map

```
map_msg = rosWriteOccupancyGrid(map_msgType,map_coverage);
send(pub.map,map_msg)
pause(0.0002)
```

Initialize robot controller

```
controller = controllerPurePursuit;
controller.DesiredLinearVelocity = 0.15;
```

```
controller.MaxAngularVelocity = 2.0;
```

Create timer objects

```
% deadlockTimer = timer;
% set(deadlockTimer, 'ExecutionMode', 'fixedSpacing')
% set(deadlockTimer, 'StartFcn', 'tic')
% set(deadlockTimer, 'TimerFcn', 'duration = toc;')
% set(deadlockTimer, 'StopFcn', 'duration = toc;')

idle = timer;
set(idle, 'ExecutionMode', 'fixedSpacing')
set(idle, 'StartFcn', 'tic')
set(idle, 'TimerFcn', 'stopped = toc;')
set(idle, 'StopFcn', 'stopped = toc; stopped = 0;')
```

Initialize variables for data collection

```
prev_Orient = 0;
turnCount = 0;
path_length = 0;
overlap = 0;
```

Acquire and transform current lidar scans to map frame

```
tftree = rostf;
pause(1);
```

Initialize check variable for deadlock condition

```
searching = 0;
```

Set initial velocities to zero

```
cmd_vel = rosmessage("geometry_msgs/Twist","DataFormat","struct");
cmd_vel.Linear.X = 0;
cmd_vel.Angular.Z = 0;
send(pub.cmd_vel,cmd_vel);
```

Start timer and begin navigation loop

```
nav_start_time = tic;
incomplete = true;
while incomplete
```

```
cartScanDataTransformed = laserScanTransform (tftree,sub.scan, 'lidar_1');
```

Read updated coverage map from '/coverage_map' topic

```
msg.coverageMap = sub.coverageMap.LatestMessage;
map_coverage = rosReadOccupancyGrid(msg.coverageMap);
% show(map_coverage)
```

Set stop condition if robot is stuck in deadlock

```
if searching > 150
    disp('in deadlock')
    break
```

Plan next movement for complete coverage

1. Pass in current grid index position
2. Overlay transformed lidar scans on coverage map
3. Calculate and next goal pose using BINN
4. Check for deadlock event

```
else
    [position.next_grid_index, orientation, deadlock, nn_mat, path_dist] =
nextPosition(msg.coverageMap, position.current_grid_index, nn_mat,
cartScanDataTransformed);

    path_length = path_length + path_dist; %calculate path length data

    % count number of turns
    if prev_Orient ~= orientation
        turnCount = turnCount + 1;
        prev_Orient = orientation;
    end

    switch deadlock
        case 0
            searching = 0;
        case 1
            searching = searching + deadlock;
    end
%     disp(searching)
```

Navigate to goal

```
    target = grid2world(map_coverage, position.next_grid_index);
```

```
    % count number of overlaps
    if checkOccupancy(map_coverage, target) == -1
       overlap = overlap + 1;
    end


    release(controller);
    controller.LookaheadDistance = 1.5;
    controller.Waypoints = target;


    [amcl.X, amcl.Y, amcl.th] = amclPose(sub.amcl);
    current = [amcl.X, amcl.Y];

    stopped = 0;

    while norm(target - current) > 0.1

       % Send cmd_vel
          % if obstacle in the way perform obstacle avoidance
          % else set linear & angular velocity to [v, omega]

       scan = lidarScan(sub.scan.LatestMessage);
       right_obstacle = removeInvalidData(scan,'RangeLimits',[0.01 0.15], 'AngleLimits',[-pi,
0]);
       %disp(right_obstacle.Count)
       if (right_obstacle.Count > 0) %|| (front_right_obstacle.Count > 0)
          %disp('obstacle right')
          %stop
          cmd_vel.Linear.X = 0.0;
          cmd_vel.Angular.Z = 0.0;
          send(pub.cmd_vel,cmd_vel);
          pause(0.2)

          %back up
          cmd_vel.Linear.X = -0.1;
          cmd_vel.Angular.Z = 0.0;
          send(pub.cmd_vel,cmd_vel);
          pause(0.5)

          while right_obstacle.Count > 0
             %turn left
             cmd_vel.Linear.X = 0.00;
             cmd_vel.Angular.Z = 1.8;
             send(pub.cmd_vel,cmd_vel);
```

```
            pause(0.5)
            %reachedTarget = 0;

            %check if obstacle is still in the detection zone
            scan = lidarScan(sub.scan.LatestMessage);
            right_obstacle = removeInvalidData(scan,'RangeLimits',[0.01 0.15], 'AngleLimits',[-
pi, 0]);
        end

        cmd_vel.Linear.X = 0.0;
        cmd_vel.Angular.Z = 0.0;
        send(pub.cmd_vel,cmd_vel);
        pause(0.2)
        [amcl.X, amcl.Y, amcl.th] = amclPose(sub.amcl);
    end

    scan = lidarScan(sub.scan.LatestMessage);
    left_obstacle = removeInvalidData(scan,'RangeLimits',[0.01 0.15], 'AngleLimits',[0, pi]);
    %disp(right_obstacle.Count)
    if (left_obstacle.Count > 0) %|| (front_left_obstacle.Count > 0)
        %disp('obstacle left')
        %stop
        cmd_vel.Linear.X = 0.0;
        cmd_vel.Angular.Z = 0.0;
        send(pub.cmd_vel,cmd_vel);
        pause(0.2)

        %back up
        cmd_vel.Linear.X = -0.1;
        cmd_vel.Angular.Z = 0.0;
        send(pub.cmd_vel,cmd_vel);
        pause(0.5)

        while left_obstacle.Count > 0
            %turn left
            cmd_vel.Linear.X = 0.00;
            cmd_vel.Angular.Z = -1.8;
            send(pub.cmd_vel,cmd_vel);
            pause(0.5)
            %reachedTarget = 0;

            %check if obstacle is still in the detection zone
            scan = lidarScan(sub.scan.LatestMessage);
            left_obstacle = removeInvalidData(scan,'RangeLimits',[0.01 0.15], 'AngleLimits',[0,
pi]);
        end
```

```
            cmd_vel.Linear.X = 0.0;
            cmd_vel.Angular.Z = 0.0;
            send(pub.cmd_vel,cmd_vel);
            pause(0.2)
            [amcl.X, amcl.Y, amcl.th] = amclPose(sub.amcl);
        end

  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

        if strcmp(idle.running,'off')
            [amcl.X, amcl.Y, amcl.th] = amclPose(sub.amcl);
            start(idle) %stop this timer
            reference = [amcl.X, amcl.Y];
        end

        % Compute the controller outputs, i.e., the inputs to the robot
        [v, omega] = controller([amcl.X, amcl.Y, amcl.th]);

        cmd_vel.Linear.X = v;
        cmd_vel.Angular.Z = omega;
        send(pub.cmd_vel,cmd_vel);


        % check the current pose
        [amcl.X, amcl.Y, amcl.th] = amclPose(sub.amcl);

        % update current positon
        current = [amcl.X, amcl.Y];
        covered = world2grid(map_coverage, current);
        mapUpdate(map_coverage, covered, 0.5)


        % After 5s cehck robot location if robot is idle for >5s re-compute path
        if stopped > 5
  %             disp('check for idle')
            displacement = norm(current - reference);
            if displacement <= 0.05
                disp('stuck in idle')

                % stop
                cmd_vel.Linear.X = 0.0;
                cmd_vel.Angular.Z = 0.0;
                send(pub.cmd_vel,cmd_vel);
                pause(1)
```

```
            % back up
            cmd_vel.Linear.X = -0.15;
            cmd_vel.Angular.Z = 0.0;
            send(pub.cmd_vel,cmd_vel);
            pause(1.5)

            % move forward
            cmd_vel.Linear.X = -0.1;
            cmd_vel.Angular.Z = 0.0;
            send(pub.cmd_vel,cmd_vel);
            pause(1)

            % turn right
            cmd_vel.Linear.X = 0.0; %0.1;
            cmd_vel.Angular.Z = 2; %0.0;
            send(pub.cmd_vel,cmd_vel);
            pause(1.5)

            % turn left
            cmd_vel.Linear.X = 0.0; %0.1;
            cmd_vel.Angular.Z = 1; %0.0;
            send(pub.cmd_vel,cmd_vel);
            stop(idle)%stop(timerfind)
            pause(1)

            % stop
            cmd_vel.Linear.X = 0.0;
            cmd_vel.Angular.Z = 0.0;
            send(pub.cmd_vel,cmd_vel);
            pause(1)

            stop(idle)%stop(timerfind)
            pause(0.15)

 %              setOccupancy(map_coverage,position.next_grid_index,1,'grid') %come back to
this & determine if next position should be still labeled as an obstacle
 %              consider setting both current and next location to obstacles
            mapUpdate(map_coverage,covered, 1)
            reachedTarget = 0;
            break
        else
            stop(idle)%stop(timerfind)
            pause(0.15)
 %              disp(idle.running)
        end
```

```
        end

        reachedTarget = 1;

    end
```

stuck in idle

stuck in idle

```
    cmd_vel.Linear.X = 0.0;
    cmd_vel.Angular.Z = 0;
    send(pub.cmd_vel,cmd_vel);

    % stop timer if it is still running
    if strcmp(idle.running,'on')
        stop(idle)
        pause(0.15)
    end

    if reachedTarget == 1
        position.current_grid_index = position.next_grid_index;
    end

end
```

Update and re-publish map

```
map_msg = rosWriteOccupancyGrid(map_msgType,map_coverage);
send(pub.map,map_msg)
pause(0.0002)
```

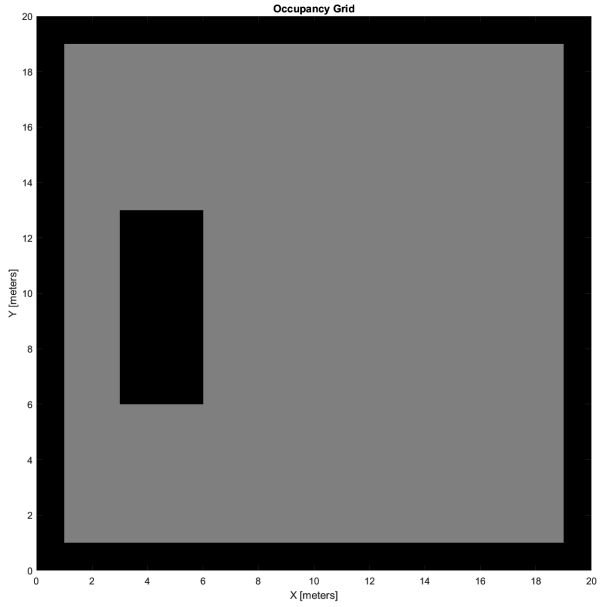
Stop condition to check if map/environment is completely covered

```
if sum(nn_mat>=0.5, 'all') == 0
    incomplete = false;
end

end

%coverage duration
nav_stop_time = (toc(nav_start_time))/60
```

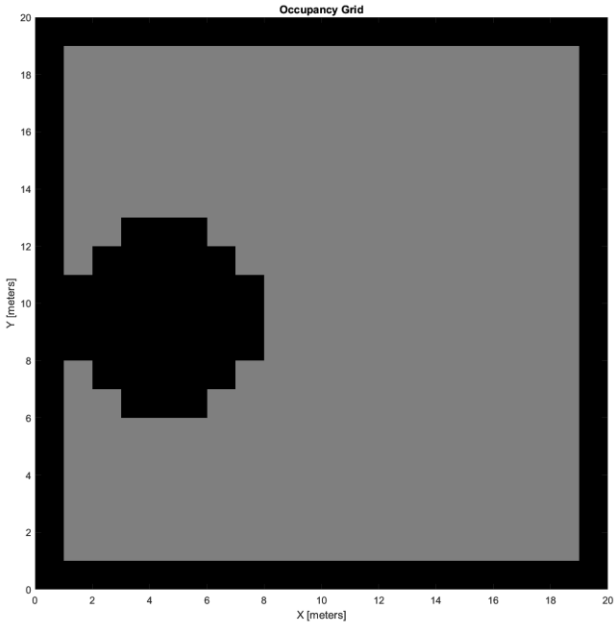
```
%calculate coverage efficiency
mat = occupancyMatrix(map_coverage);
leftover = length(find(mat <=0.5));
coverage_eff = (1 - (leftover/uncovered)) * 100

path_length
turnCount
overlap
clc
delete(idle)
```

APPENDIX B

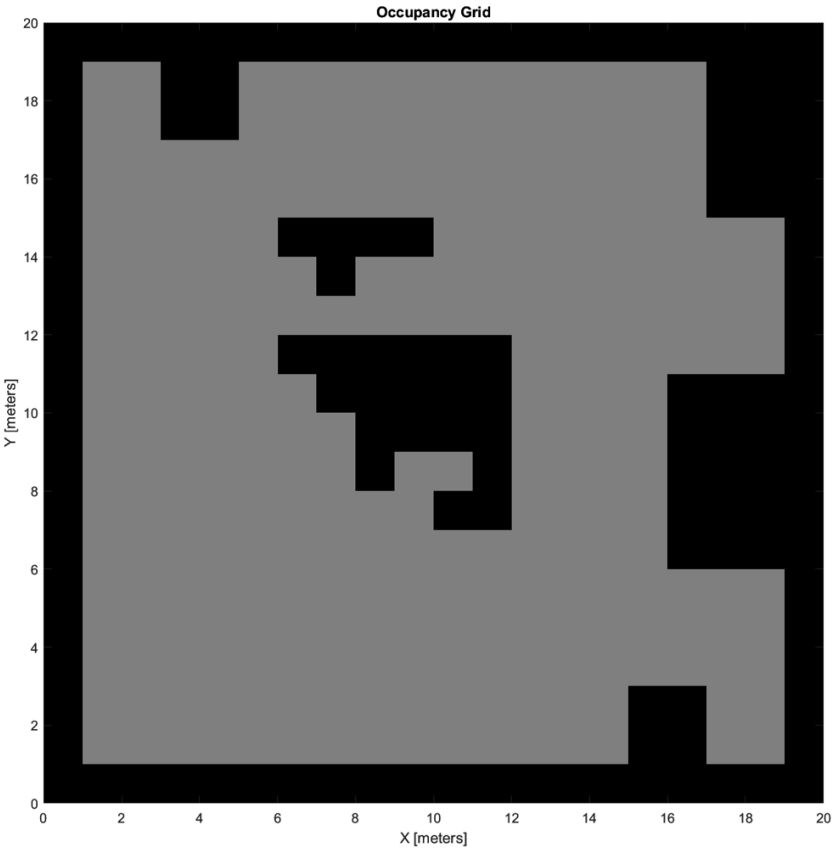
B-1: Pre-made excel maps for MATLAB simulation testing.



Rectangular obstacle

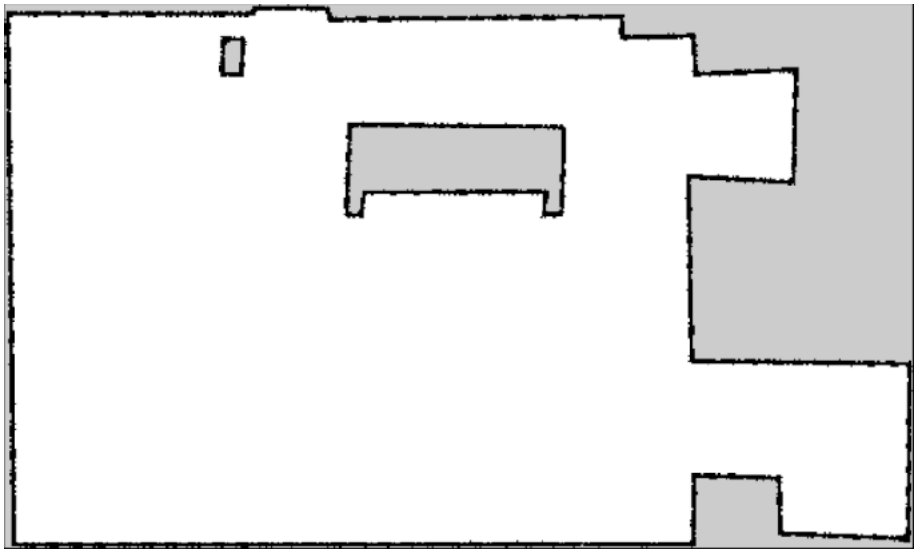


Circular Obstacle



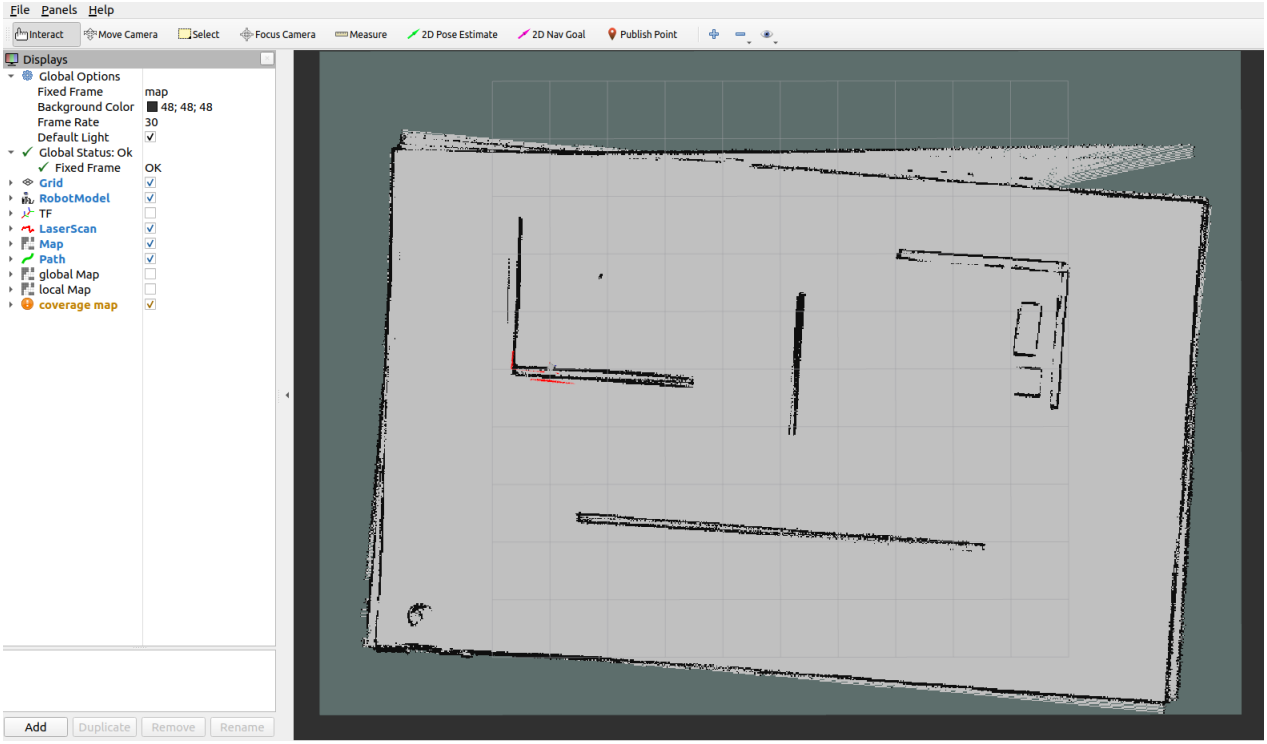
Generic Obstacle

B-2: ROS simulation mapped environments.

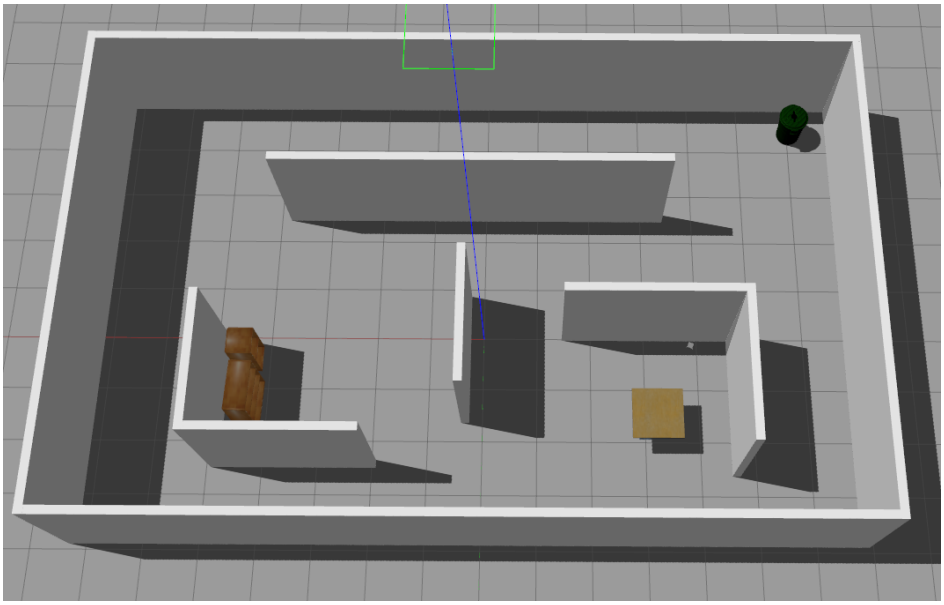


Apartment living room.

Lab space (DMD lab)



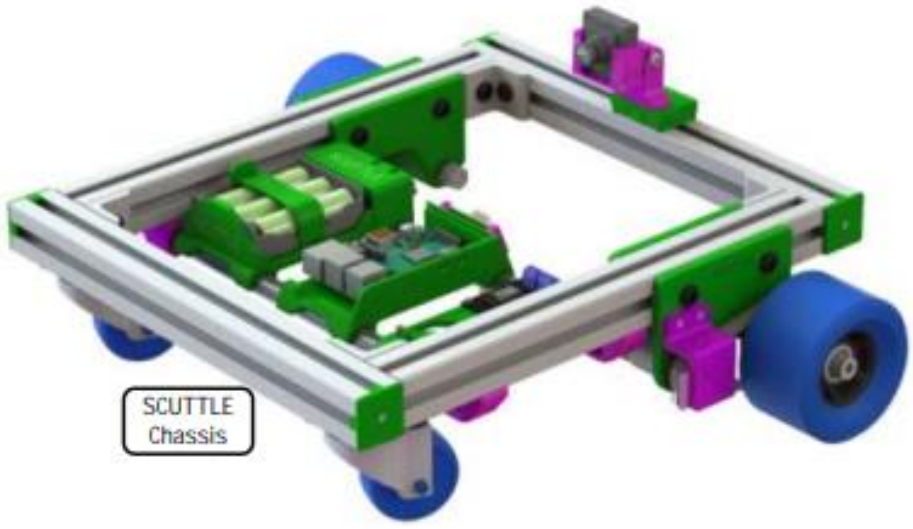
Gazebo world Map in RViz



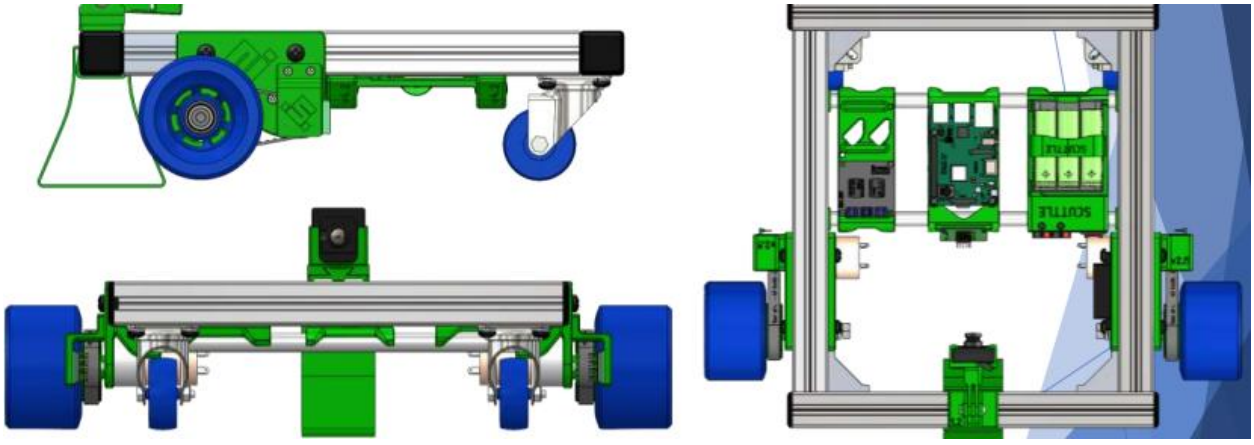
Gazebo World Environment.

APPENDIX C

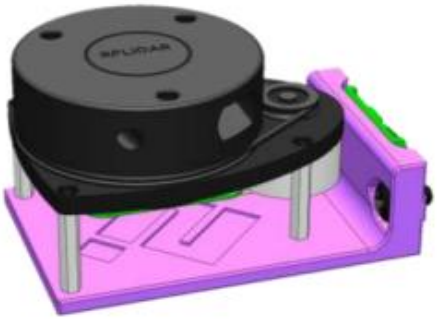
C-1: Robot Hardware components



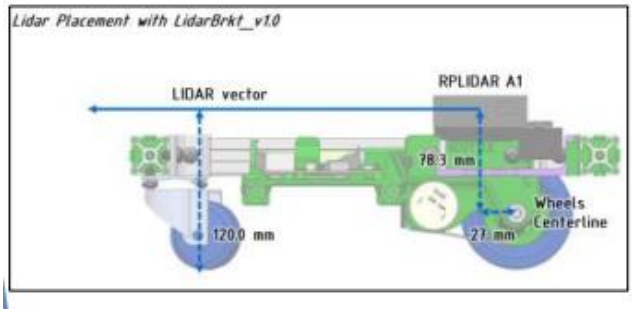
SCUTTLE CHASSIS



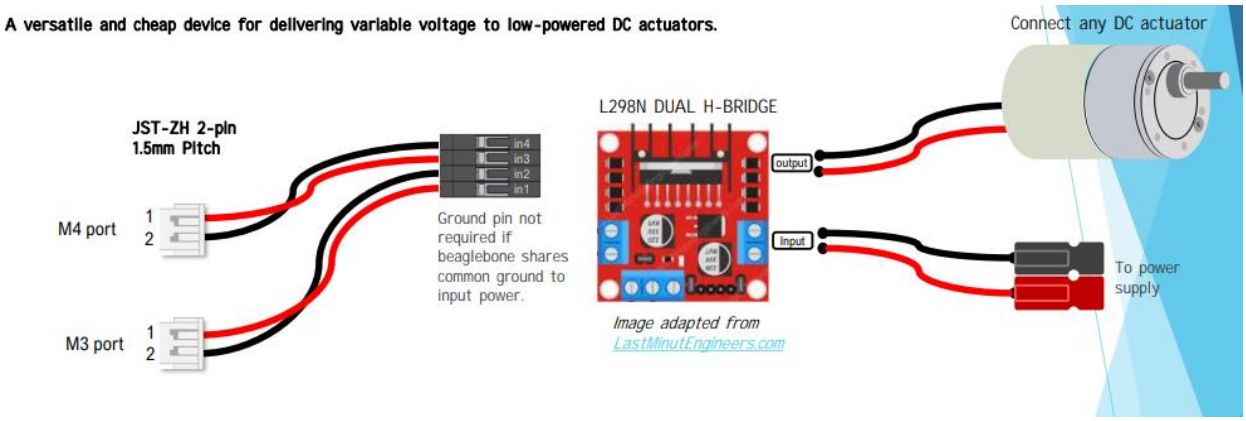
SCUTTLE CHASSIS VIEWS



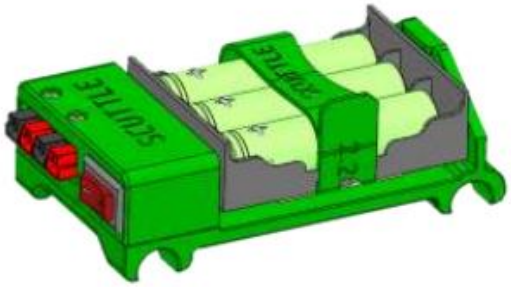
RPLIDAR A1 with Bracket



RP-LIDAR



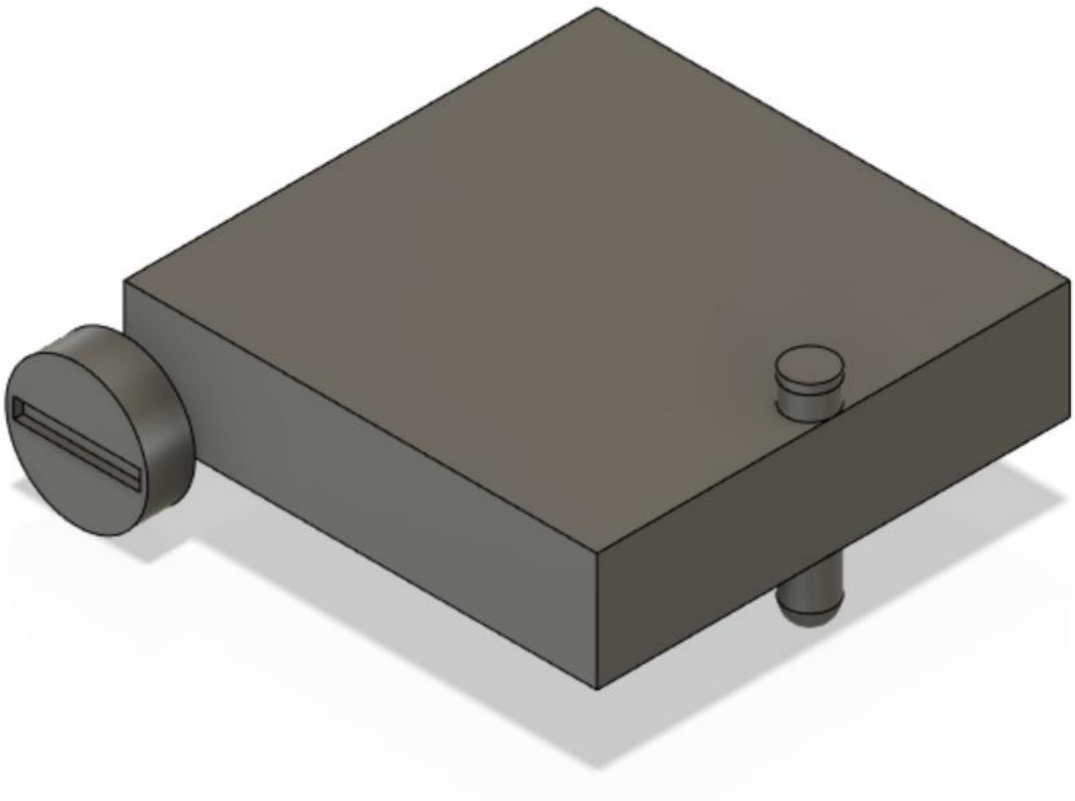
MOTOR & DRIVER COMPONENTS



BATTERY PACK

RASPBERRY PI

APPENDIX D



ROS simulation robot model