Final Project – Autonomous Mobile Robot Navigation in Manufacturing and Warehousing

Dean Rogers
College of Engineering and Mines
University of North Dakota
Grand Forks ND, USA
dean.rogers@und.edu

***Abstract*—The purpose of this project was to develop a basic solution for using mobile robots in manufacturing and warehousing applications to reduce manpower requirements for material handling, picking, and internal transportation. This project uses a small two-wheeled mobile robot with a 360-degree LIDAR scanner mounted on it to generate a map of the area and navigate to a specified location. The algorithm developed uses the LIDAR scanner to obtain 2D range data which is then passed through a cluster analysis to remove noise. That data is then sent to an occupancy mapping algorithm to generate a map for the path finding algorithm. Once the robot path has been determined, the robot drives along the path until it reaches the goal point. While a feedback algorithm would significantly increase the accuracy of the robot navigation, the Kalman filter algorithm was not able to be implemented in this project.**

Keywords—LIDAR, occupancy mapping, mapping, localization, path planning, navigation

I. Introduction

Autonomous robotics, intelligent machines capable of performing tasks without explicit human control, require the input of sensors to provide information about the world around them in order to make decisions and perform tasks. As the robots operate in a complex 3D world, sensor data must be analyzed and matched to the physical world to allow the robot to navigate effectively.

In the manufacturing and resale industries, there is a flow of goods through a warehouse. While optimizations can be made to increase efficiency, such as reducing travel distances and minimizing retraced steps, there is a limit to the amount of time that can be saved. Being a fairly simple repetitive process, picking and transportation of goods in a warehouse could be further optimized through the use of autonomous mobile robotics.

By removing the human element in picking and transportation, cost can be reduced as human labor can be focused on the more technical or less repetitive tasks such as assembly or packaging. While human labor is being used to assemble goods or package orders, an autonomous robot can be assigned a task and, without human interaction, prepare the goods required for the next order, allowing the workers to continue seamlessly. Not only will the save time for the workers and money in labor for the company, but this can also reduce the risk of human error. The high repeatability of a task performed by a robot reduces the time spent correcting mistakes and increases order accuracy and thus customer satisfaction.

The tasks to be performed by an autonomous mobile robot in this project are to simulate picking goods from a specified location and delivering them to another. The robot will start from a random location, determine where it is, plan a route to the part location, navigate to that location while avoiding stationary obstacles, plan a route to the delivery location, and navigate to that location while avoiding obstacles.

Autonomous mobile robots must use a sensor to determine their location in their environment and to detect obstacles. Often used are 2D scanners which provide a set of points in the horizontal plane that correspond to vertical physical surfaces. One method of converting this data into a form that can be used by a robot for path planning is occupancy mapping. By converting the area scanned into a grid and determining which cells are likely to be occupied and which are not, path planning algorithms can use that data to navigate around obstacles and through a map.

In this project, a basic 360-degree 2D LiDAR scanner was used to collect the range data. The unit used is the YDLIDAR X4, a small entry-level scanner with a range of 0.12m to 10m. The scanner interfaces using a USB cable so the scans can be read into the provided software for visualization or read with a program to perform manipulation of the data. To read and work with the scan results, a Python program was created that makes use of libraries such as PyLidar3, for connecting to the scanner, MatPlotLib, for plotting the results, and Pandas, for handling the data.

II. Hardware and Software Setup

A. Hardware Used

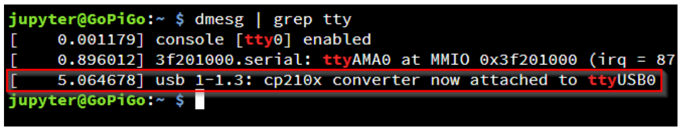
- GoPiGo Robot
  - GoPiGo Robot Chassis
  - GoPiGo Electronic Board
  - Raspberry Pi Computer (3B+)
  - DexterOS microSD Card
  - Rechargeable Battery
- YDLIDAR X4 LIDAR Scanner

- USB Battery Pack
- USB-A to USB-C Cable
- USB-A to MicroUSB Cable

B. *Lidar Scanner Setup*

Before connecting to the LIDAR scanner, it should be mounted to the robot. If possible, it would be best to drill holes in the top part of the chassis which correspond to the legs of the LIDAR scanner, however, if that is not an option, small pieces of wire can be used to tie the scanner to the holes along the edge of the top of the chassis. The scanner also requires external power, so an additional battery pack is required. Mount the battery pack to the robot and plug the power USB cable into the battery pack and the power port on the LIDAR scanner. The physical setup of the robot is shown in Appendix H.

In order to communicate with the YDLIDAR X4 scanner, the correct serial communication port must be specified in the code. The robot must be turned on and connected to from a computer. In the Code > Python section, open a new terminal. With the scanner plugged in to one of the USB ports on the RaspberryPi, use the command ‘`dmesg | grep tty`‘ to find the port that the scanner is connected to. The port will start with “tty”, as shown in Fig. 1.



```
jupyter@GoPiGo:~ $ dmesg | grep tty
[    0.001179] console [tty0] enabled
[    0.896012] 3f201000.serial: ttyAMA0 at MMIO 0x3f201000 (irq = 87
[    5.064678] usb 1-1.3: cp210x converter now attached to ttyUSB0
jupyter@GoPiGo:~ $
```

Fig. 1. Ubuntu Terminal - Find LIDAR Serial Port

C. *Python Code Setup*

For the Python code to run and the LIDAR scanner to be used, the PyLidar3 library must first be installed on the RaspberryPi. First try using the command ‘`pip install PyLidar3`’ in the terminal. If that does not work, navigate to the GitHub page for the PyLidar3 library, download the package, unzip the folder, and upload the files to a file location on the RaspberryPi. Once uploaded, change the directory of the terminal to match the directory where the setup.py file is located for the PyLidar3 library. Then run the command ‘`python3 setup.py install`’ in the terminal to install the library.

With the library installed, create a new Python3 notebook on the RaspberryPi and paste the code from Appendixes A-G into it. Change the port variable to match the name determined in step *B* and add the prefix “/dev/” to the port name.

III. Algorithm Implementation

The algorithm for this project was implemented using Python in a Jupyter Notebook using the easygopigo3, PyLidar3, Pandas, Numpy, MatPlotLib, time, and math libraries. The easygopigo3 library is used to send commands to the robot to move. The PyLidar3 library is used to send commands to and receive data from the YDLIDAR X4 scanner. The Pandas library is used for storing the data and is particularly useful for manipulating data quickly. The Numpy library is used in conjunction with the Pandas library to perform manipulations of the data. MatPlotLib is used to display the graphs of the data. The time library is used for specifying the scan time of the LIDAR scanner. The math library is used in calculations done on the scan data.

A. *Import Libraries and Initialize Variables*

Before the program starts, all required libraries are imported, and variables are initialized. While it isn’t always necessary to do this at the start of the program, it helps keep it organized and makes modifying the variables easier as the program gets larger. The code to import the libraries and initialize the variables is shown in Appendix A.

B. *Define Functions*

Since the graph search algorithms are recursive, they need to be set up using functions. The functions for the graph search algorithms, as well as for plotting the progress of the algorithms, are defined at the beginning of the program so they can be used later. Since these functions also use some of the libraries imported in part A, the functions must be defined after the libraries are imported. The code defining the functions is shown in Appendix B.

A recursive function is one that can call itself, which results in the function being able to be repeated multiple times where the results of the later iterations can affect the results of the earlier iterations. This is required in the map search algorithms since it is possible for the path the algorithm is taking to get ‘stuck’, meaning there aren’t any more unoccupied points it can move to from the current point, but it hasn’t reached the end point. Using a recursive function in this scenario allows the algorithm to retrace its steps and search for other possibilities at each step until it finds a new path.

1) *Breadth-First Search*

The breadth-first search works in all directions at once, by searching all points adjacent to the previous search points at the same time. This creates multiple search paths that branch out at each iteration. As the search spreads out, the number of points searched increases and spreads out across the map until the end point is reached. The algorithm then works backward through the path that reached the end point first. This algorithm weights each cell evenly, so the resulting path is the shortest path not including diagonals.

2) *Depth-First Search*

The depth-first search works on one path at a time, searching until it either finds the end point, or gets stuck. If it gets stuck it backtracks and tries to find a new path to follow. This method creates one path at a time, so once the end point is reached, it returns the path it was following. This algorithm weights each cell evenly, but since the order of the directions it searches in is arbitrary, it may find the shortest path, excluding diagonals, or it may find and extraordinarily long path.

3) *Directional Depth-First Search*

The directional depth-first search uses the same principle of the depth-first search where a single path is explored at a time, however, it differs in how it determines which cell to search next. The depth-first search just uses an arbitrary setting, such as search down first, then right, then up, then left. The directional depth-first search, on the other hand, compares the location of the current point to the end point and moves in the direction with the greatest distance to go toward the end point.

C. *Perform Scans*

The first step in creating an occupancy map is obtaining the range scan data. The test program provided in the PyLidar3 documentation was modified to fit the needs of the project better. The serial port was added as a static variable rather than a prompt. Additionally, the scan time, plot maximum, and minimum measurement threshold were added as variables. The scan data is also added into a Pandas DataFrame for use in the occupancy mapping algorithm. The updated code is shown Appendix C.

Three parameters are used in the scanning code: Minimum Distance Threshold, Buffer and Scan Time. The scan time just specifies how long the scanner runs, changing the number of scans performed. This value was set to 4 seconds which returns about seven to 10 scans. This provides additional information for the cluster analysis and helps remove noise without creating excess data to analyze. The buffer is simply subtracted from all the distance measurements to create a buffer around all the objects detected to help prevent the robot from colliding with the walls or obstacles.

The Minimum Distance Threshold specifies the minimum distance between the sensor and the measurement that will be recorded. The effect of this parameter is shown in Figs 3 and 4. This parameter essentially removes all data points within a radius of the origin point. This parameter was set at 200 millimeters as it included as much data as possible, while removing the noise that appears near the scanner. With the minimum scanning distance of the YDLIDAR X4 being 120 millimeters, the value selected removes only the points very close to the minimum which are highly likely to be noise.

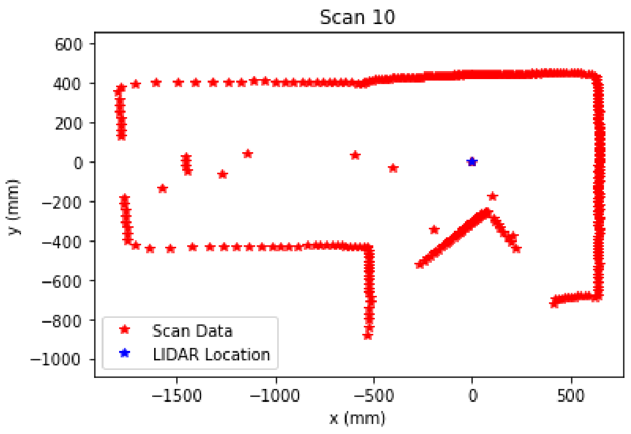


Fig. 2. LIDAR Scan with Min Distance Threshold = 200mm

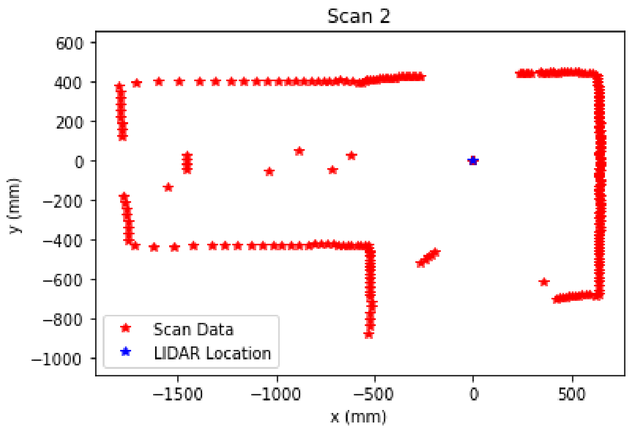


Fig. 3. LIDAR Scan with Min Distance Threshold = 500mm

An additional optimization to increase the accuracy of the data, and thus the occupancy map, was to drop the first scan performed. Many of the first scans are full of noise and show very little of the actual features in the area. This is likely due to the acceleration of the scanner as it starts up, so once it is spinning at a constant velocity, the scans are much cleaner. This can be shown in comparing Fig. 4 to Fig. 5 which are scans 2 and 1 respectively.

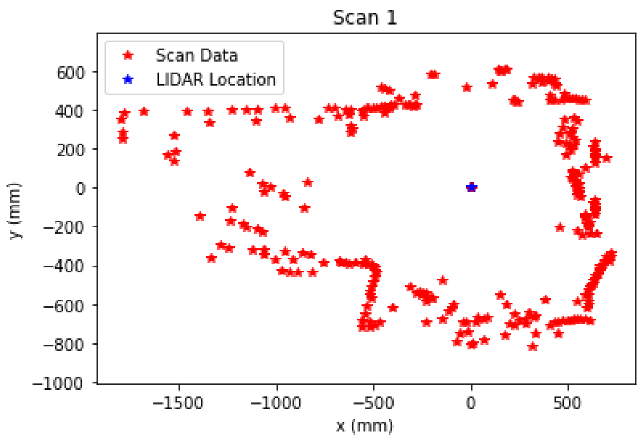


Fig. 4. LIDAR Scan - First Scan Showing Excess Noise Due to Scanner Acceleration

D. *Cluster Analysis*

In each scan, there are a small number of points that do not correspond to physical objects in the area scanned. These points are considered noise and will negatively affect the performance of the occupancy mapping algorithm by providing false positives of an object. To help combat this, a cluster analysis is performed which groups nearby points into cells and removes points in cells that don't contain enough points.

While this method alone is fairly effective, it can have issues if noise is concentrated in a certain area on a scan, or if objects that are further away from the sensor have points on a line but spread out. In the first scenario, the dense cluster of noise may not get removed if it doesn't fall below the threshold. In the second scenario, valid points can be removed when they are spread apart and fall below the threshold. To help prevent these two issues, multiple scans are taken while keeping the scanner stationary. These scans are then stacked on top of each other, and the point clouds of the actual objects get denser, while the noise stays sparse as it is random. The Python code to perform the cluster analysis can be found in Appendix D.

The cluster analysis algorithm uses two parameters: Cluster Grid Size, and Cluster Threshold. The grid size parameter specifies the size of the grid squares used to group the data points. Larger grid sizes tend to reduce the number of points removed as noise, while smaller grid sizes increase the number of points removed. This is shown in Figs 6, 7, and 8. The grid size selected for this exercise was 0.1 meters for this environment; the ideal grid size could change depending on the overall size, complexity, or noise of the environment being scanned.

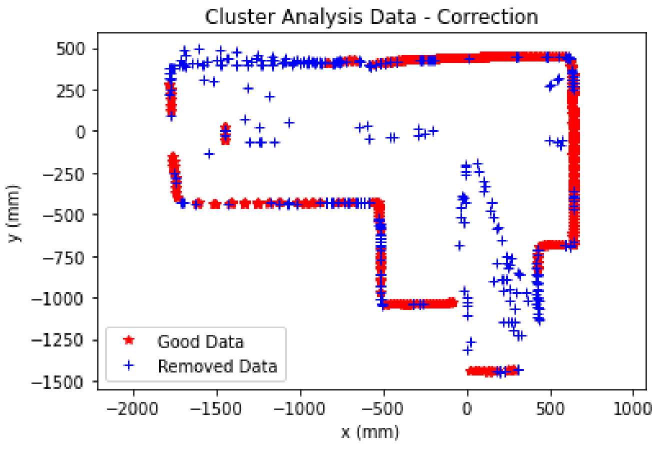


Fig. 5. Cluster Analysis - Grid Size=0.01m, Cluster Threshold=0.7

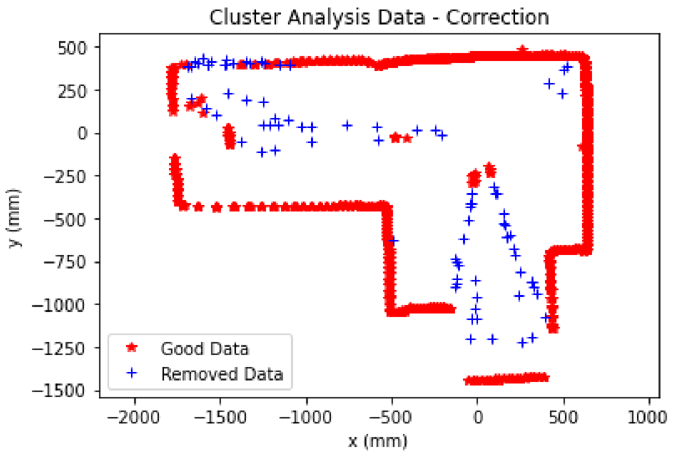


Fig. 6. Cluster Analysis - Grid Size=0.1m, Cluster Threshold=0.7

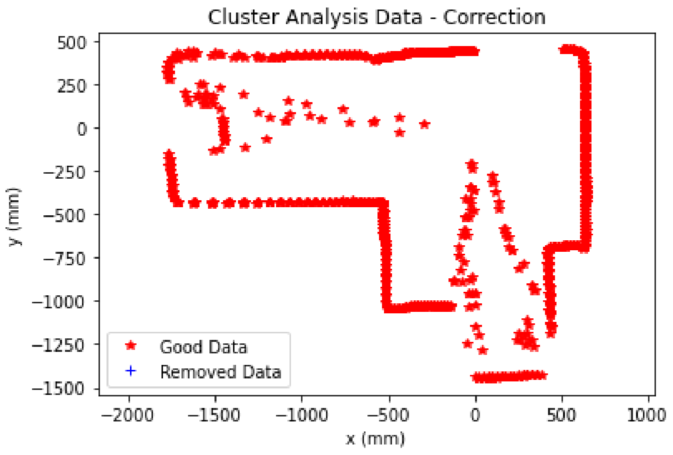


Fig. 7. Cluster Analysis - Grid Size=0.5m, Cluster Threshold=0.7

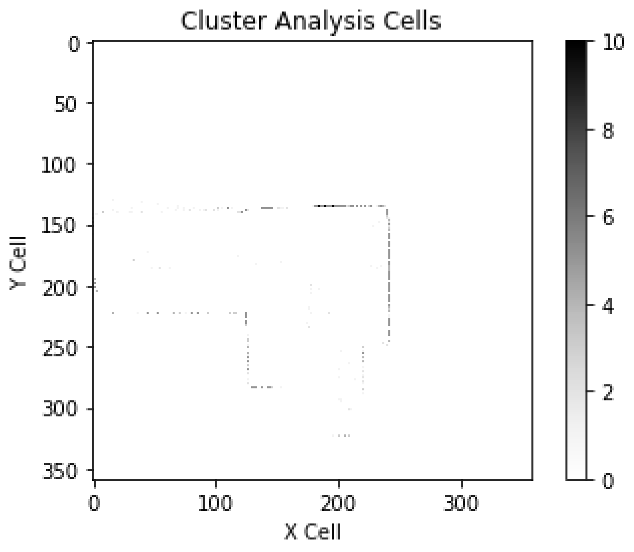


Fig. 8. Cluster Analysis Heatmap - Grid Size=0.01m

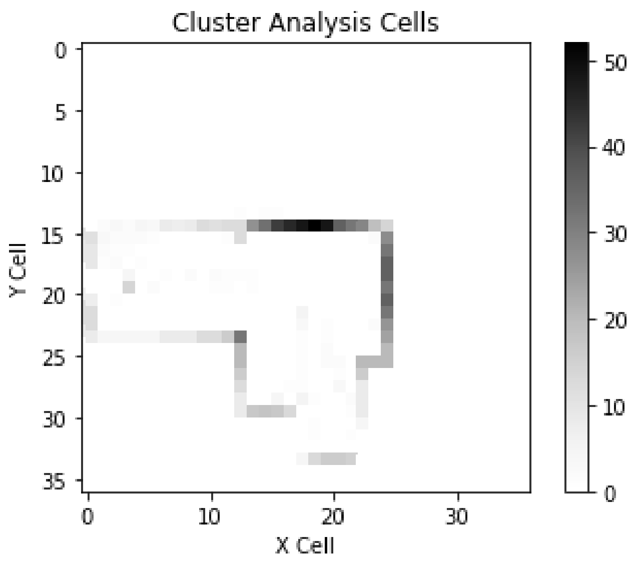


Fig. 9. Cluster Analysis Heatmap - Grid Size=0.1m

The second parameter used in the cluster analysis is the Cluster Threshold. This specifies the minimum ratio of points in a cell to the number of scans included in the cluster analysis. The higher this ratio, the more points get removed as noise. The value selected for this exercise is 0.5, which removes a good portion of the noise. It does leave some of the noise, however, increasing this threshold starts to remove points that are correct. Additionally, the occupancy mapping algorithm is able to remove some noise so the noise that is left doesn't affect the result significantly. The effect of the Cluster Threshold is shown in Figs 11, 12, and 13.

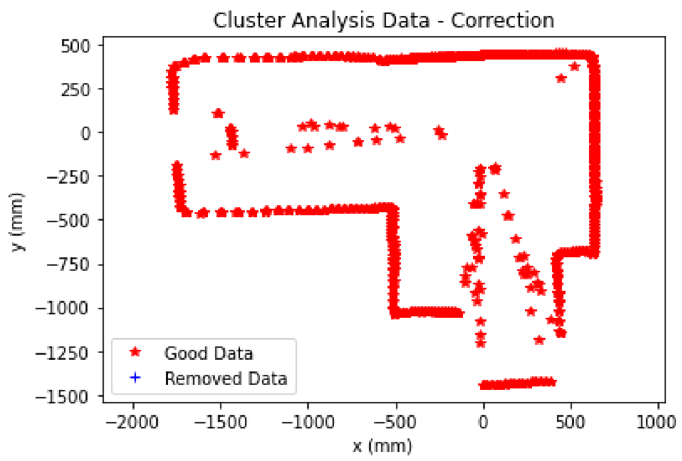


Fig. 10. Cluster Analysis - Grid Size=0.1m, Cluster Threshold=0.1

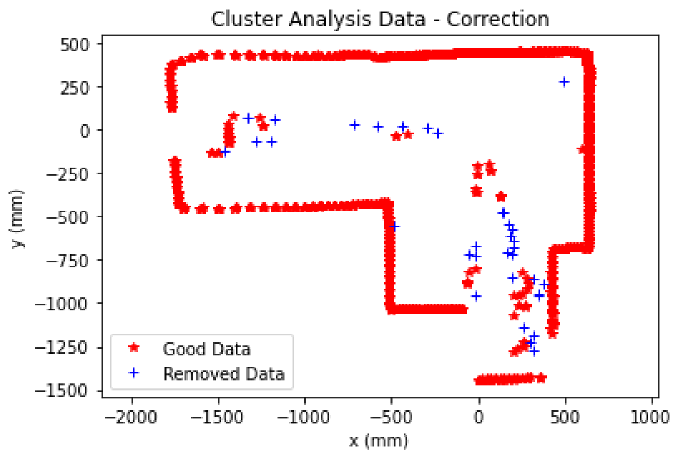


Fig. 11. Cluster Analysis - Grid Size=0.1m, Cluster Threshold=0.5

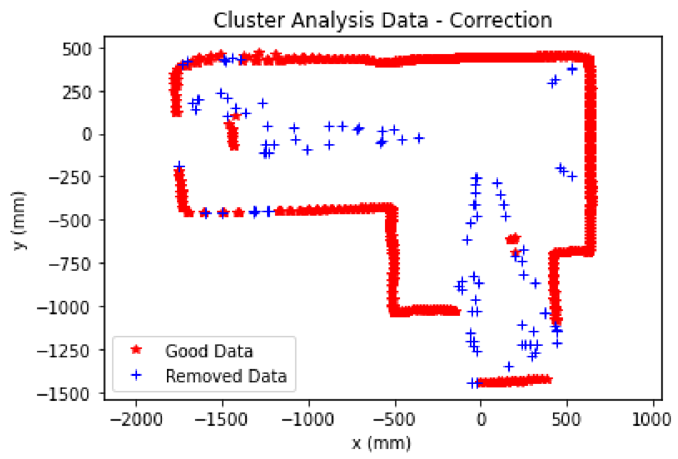


Fig. 12. Cluster Analysis - Grid Size=0.1m, Cluster Threshold=1

E. *Occupancy Mapping*

To generate the occupancy map, a grid is created that encompasses the overall size of the scanned area. Two values are stored for each cell in the grid, M and C. M is the evidence of the cell being occupied, and C is the number of times the cell has been observed.

To populate this grid, each measurement is analyzed individually. The M and C values of the cell that contains the measurement location get incremented. For all the cells that the light beam would have passed through to reach to measurement location, the M value is decremented while the C value is incremented. The concept is, the higher the value of the M/C value for a cell, the more likely it is to be occupied. A threshold can then be used to specify which cells are considered occupied and which are not.

This algorithm was implemented using a Pandas DataFrame to hold the list of cells, their locations, and their M and C values. This was then joined with a DataFrame containing all the measurements such that all measurements are compared to all cells. Then the distances between the measurement and the cell center, and the measurement path and the cell center are calculated. Based on a distance threshold, the M and C values for each cell are incremented, decremented, or ignored. The occupancy mapping Python script is included in Appendix E.

Other common algorithms for incremental occupancy mapping, such as Bresenham's Line and ray marching, work by traversing the ray path from the query point to the measurement location [1]. These methods would require an iterative solution which, especially over large datasets, is time-consuming. Even though the method implemented in this exercise requires larger datasets to compare every cell location to every measurement, it does not require iterative solution and remains suitably fast.

Three parameters are used to control the occupancy mapping algorithm: Tessellation Grid Size, Ray Cast Threshold, and Occupancy Threshold.

1) *Tessellation Grid Size*

The Tessellation Grid Size, much like the Cluster Grid Size, specifies the size of the cells that the area is split into. Decreasing the size can help increase the accuracy of the location of detected objects, however, it can increase the opportunity for false negatives in certain situations. The effect of changing the Tessellation Grid Size is shown in Figs 14, 15, and 16. The Tessellation Grid Size value was chosen to be 0.08 meters for this exercise as it provided the best resolution while keeping gaps to a minimum.

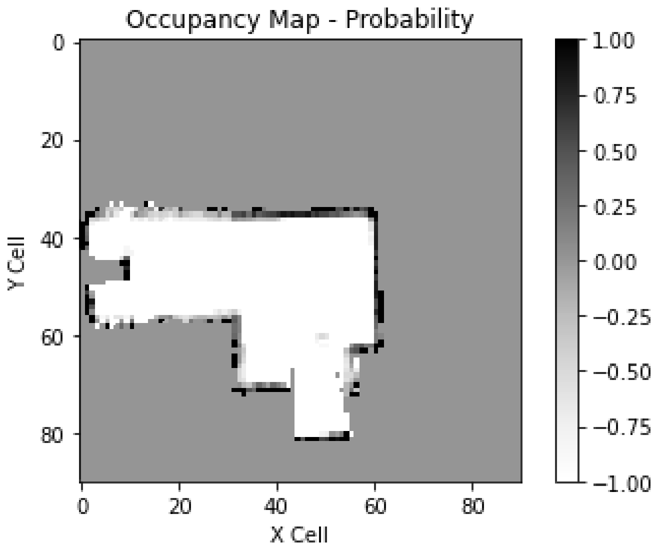


Fig. 13. Occupancy Probability Map - Tessellation Grid Size=0.04m

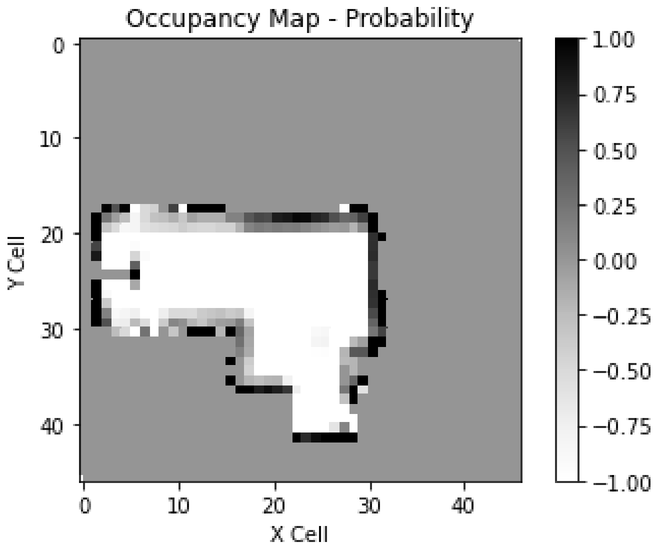


Fig. 14. Occupancy Probability Map - Tessellation Grid Size=0.08m

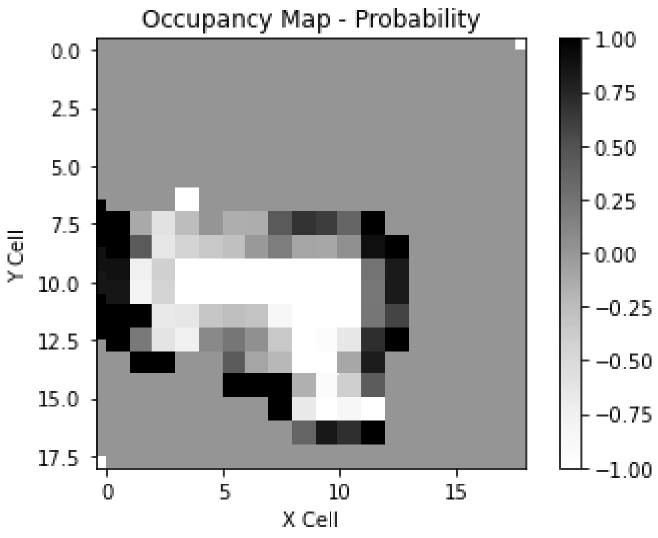


Fig. 15. Occupancy Probability Map - Tessellation Grid Size=0.2m

2) *Ray Cast Threshold*

The method of determining which cells the ray of light, used to measure each distance, passed through is called ray casting. This determines that the cells the ray passed through are empty and the cell where the object was found, and thus a measurement returned, is occupied.

While calculating exactly which cells each ray passed through would provide the most accurate results, it would be too computationally expensive. Instead, the rays are considered to pass through a cell if they come within a certain radius of the center of the cell. That radius is the Ray Cast Threshold. The effects of the Ray Cast Threshold are shown in Figs 17, 18, and 19. The value chosen for this exercise was setting the Ray Cast Threshold equal to twice the Tessellation Grid Size to provide a buffer between obstacles and the path generated for the robot. This provides some overlap between cells which can help close some gaps but may also reduce the certainty of some cells. As can be seen in Fig. 17, too small a value causes many false negatives, while Fig. 19 shows how too large a value creates many false positives.

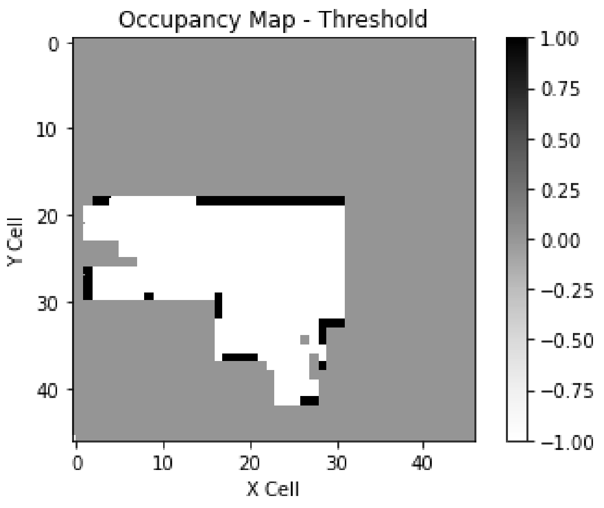


Fig. 16. Occupancy Threshold Map – Ray Cast Threshold = Half the Tessellation Grid Size

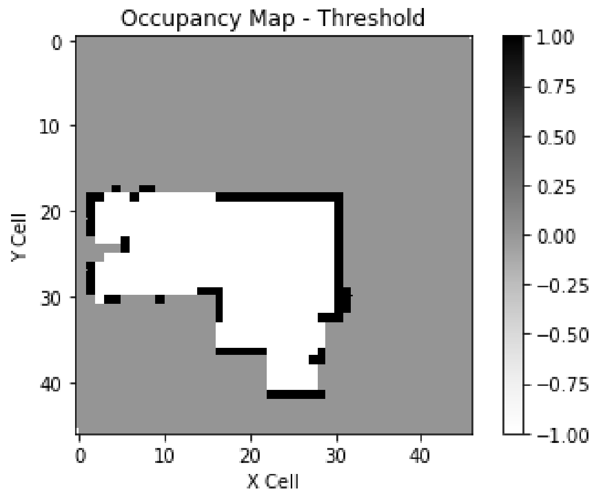


Fig. 17. Occupancy Threshold Map – Ray Cast Threshold = Tessellation Grid Size

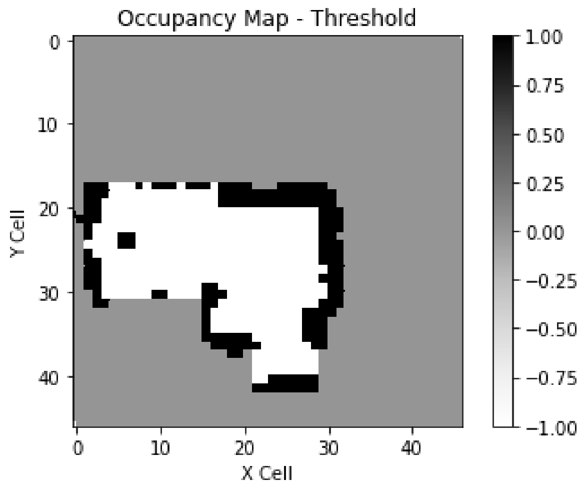


Fig. 18. Occupancy Threshold Map – Ray Cast Threshold = Twice the Tessellation Grid Size

3) *Occupancy Threshold*

While the probability map can be useful as, in uncertain areas it would provide the robot a place to investigate further to increase the certainty of occupancy, it is also helpful in path planning to have a threshold which determines if each cell is to be considered occupied. This can be visualized by comparing Figs 14 – 16 to 17 – 19. Figs 14 – 16 show the probability of each cell as a shade of gray, such the maps have a fading effect near objects, while in Figs 17 – 19 unoccupied cells are white, occupied cells are black, and uncertain cells are a single shade of gray.

The effect of changing the Occupancy Threshold is shown in Figs 20, 21, and 22. The value can only be between -1 and 1, where -1 is completely unoccupied and 1 is completely occupied. For this exercise, the Occupancy Threshold was chosen to be -0.3 which provides the most complete occupancy map without including many false positives.

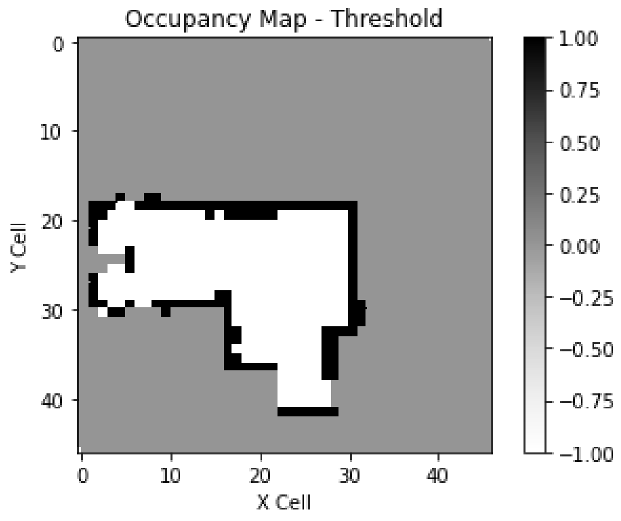


Fig. 19. Occupancy Threshold Map - Occupancy Threshold = -0.6

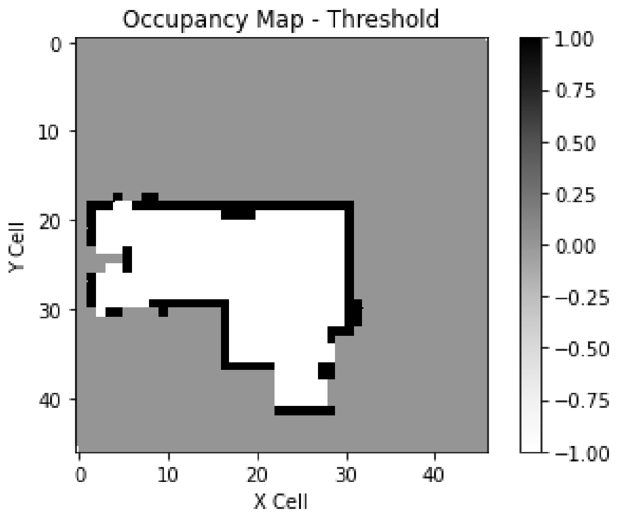


Fig. 20. Occupancy Threshold Map - Occupancy Threshold = -0.3

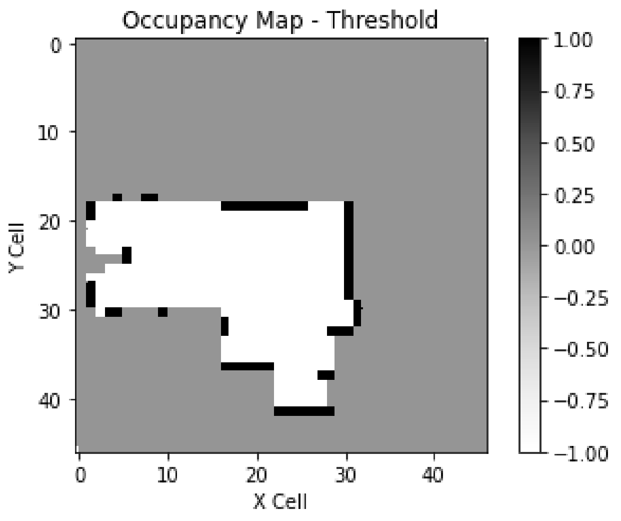


Fig. 21. Occupancy Threshold Map - Occupancy Threshold = 0.4

F. *Path Planning*

The path planning algorithm used in this project is the breadth-first search due to its consistency and how the paths it generates have minimal turns for the robot to perform. The breadth-first algorithm provides very consistent results where, if every cell is considered to require the same distance travelled, it finds the shortest route possible. This algorithm tends to form an L-shaped path, where the path moves out from the start point then turns left to the goal point. This is due to the order of checking for available points in the algorithm.

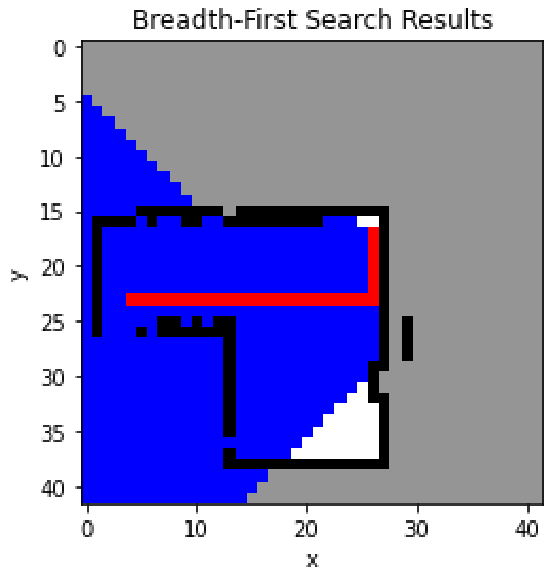


Fig. 22. Breadth-First Search - Start=(3,23) End=(26,17)

The algorithm considers uncertain spaces as available so it can plot a path through them and while it could be made safer by preventing that, on larger maps that could mean that the algorithm isn't able to find a solution where there are areas outside of the vision of the sensor. This does mean that the algorithm can create impossible paths such as the one shown in Fig. 22. In the figures, the blue cells are cells that were visited by the algorithm and the red cells are the path determined by the algorithm. The definition of the path planning function is shown in Appendix B. and the execution of the path planning algorithm is shown in Appendix F.

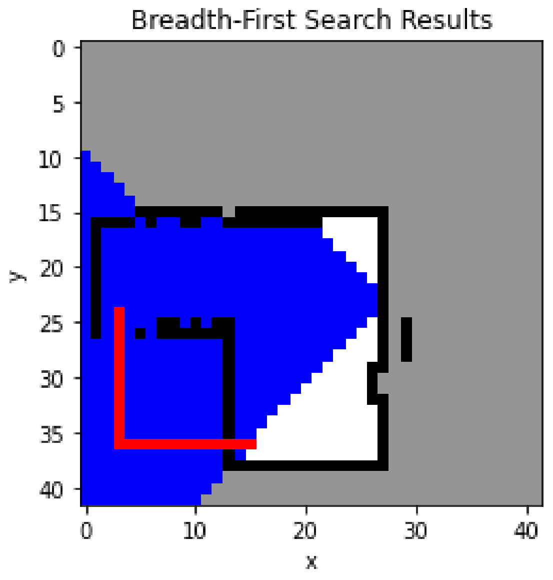


Fig. 23. Breadth-First - Start=(3,23) End=(15,36)

G. *Navigation*

To navigate, each step in the path generated by the path planning algorithm is compared to the current state of the robot. First, the angle is determined and if different than the current robot angle, the robot turns to match the required angle. Then the location is compared and if it is different the robot moves in a straight line forward to the next point. The code for this is shown in Appendix G.

IV. Algorithm Results

Up to the navigation step, algorithm provides a sufficient level of accuracy to assist with basic robot navigation, however, without a robot localization algorithm to act as a feedback loop, the navigation of the robot is not accurate. As the robot travels, it gets slightly off course, so each further movement generally takes it further off course.

The LIDAR scanner is adequate for basic robot navigation but did have some issues that negatively affect the results of the algorithm. First, the first scan performed is often full of noise and must be discarded. Second, every scan has random noise and many of the scans had repetitive noise. While the random noise can usually be removed with multiple scans passed through a cluster analysis, the repetitive noise was not removed. This is shown near the center of the area in Fig. 7. This noise could be caused by ambient light in the environment.

In the occupancy mapping algorithm, there are some false negatives in areas where the measurements are more spread apart such as along walls that are approaching parallel with the sensor beam. This causes a large change in linear distance between measurements with only a small angle change. This is shown on the left side of the top wall in Fig 12. These false negatives could potentially be remedied with a continuous occupancy mapping scheme was being used which could update low-confidence areas as the robot moved around the area. Using this process would change the optimal values of many of the parameters.

Since the Kalman filter is not functional, the navigation in this project is simply done from the initial path plan. This does result in inaccuracy as the robot is not able to drive the exact distance or turn to the exact angle specified, especially since the experiment is performed on carpet which causes significant variance in the actual distance moved. Some testing was performed to determine the relationship between the distance or angle sent to the robot and the actual distance or angle moved, and both relations came out to be almost the same. For simplicity, both the turn and drive commands used the same mapping equation. This method performs relatively well, however, as distance traveled is increased the error in the navigation is increased. Additionally, there is not obstacle avoidance implemented on the robot due to the lack of a functional Kalman filter to localize the robot to determine a new path around the new obstacle.

The Kalman filter, which uses statistical analysis to match extracted lines from 2D scan data to a predefined map, would allow for significantly more accurate robot navigation. As the robot moves, additional scans could be performed which, when passed through the Kalman filter, could determine the robot's new location with a certain degree of certainty, and a new path to the goal point could be planned. This allows for significant correction of error introduced in the environment.

V. Conclusion and Future Work

The method of scanning and pre-processing the data works well and cleans up most of the noise that appears in the scan data. This helps improve the accuracy of the rest of the algorithms that use the data. While there are other methods for cluster analysis, this relatively simple method is sufficient for this application.

While the iterative method for creating an occupancy map implemented in this exercise provides sufficient results for basic robot navigation and obstacle avoidance, more complex statistical methods demonstrate significantly more accurate results. One such method, called the Gaussian Process (GP), has multiple advantages such as the ability to introduce dependencies between data points and therefore generate maps from relatively noisy or sparse data, as well as producing a variance plot which can highlight areas that require additional exploration by the robot. To help optimize the existing program, a set of KPIs (Key Performance Indicators) could be introduced. By comparing a pre-existing map to the occupancy map generated, the accuracy of the algorithm could be greatly increased by programmatically updating the parameters and comparing the KPIs. Some potential KPIs could include True Positive Rate, False Positive Rate, Precision, and False Discovery Rate. [2, 3]

The path planning algorithm has significant opportunity for improvement. The current algorithm, the breadth-first search, only creates paths consisting of vertical and horizontal lines. In most cases, there is a significantly shorter path between the starting and ending points if the robot was to travel along a diagonal. This method would be difficult to implement in the breadth-first search, but an adaptation of the depth-first search algorithm might be able to produce diagonal lines by analyzing the results of the algorithm and extracting diagonal lines from the resulting path.

The navigation of the robot in this project presents the greatest opportunity for improvement. Without the implementation of robot localization, the robot navigation will always be inaccurate regardless of the accuracy of the path planned and the precision of the robot. External factors affect how the robot moves and create at minimum small inaccuracies in the robot's movements. The implementation of the Kalman filter, while not perfect, would at least allow significant inaccuracies to be corrected by localizing the robot throughout its path and correcting the path where needed.

References

[1] C. Walsh and S. Karaman, "CDDT: Fast approximate 2d Ray casting for accelerated localization," arXiv.org, 07-Mar-2018. [Online]. Available: https://arxiv.org/abs/1705.01167. [Accessed: 20-Nov-2021].

[2] S. T. O'Callaghan and F. T. Ramos, "Gaussian process occupancy maps," The International Journal of Robotics Research, vol. 31, no. 1, pp. 42–62, 2012.

[3] P. Markiewicz and J. Porębski, "Developing occupancy grid with automotive simulation environment," Applied Sciences, vol. 10, no. 21, p. 7629, 2020.

Appendix

A. Python Code to Import Libraries and Initialize Variables

```python
from easygopigo3 import EasyGoPiGo3
import PyLidar3
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
import time
import math

tess_grd_size = 0.08    # meters
ray_cast_thresh = tess_grd_size*2
occ_thresh = -0.3      # Number between -1 and 1
clust_grd_size = 0.1   # meters
clust_thresh = 0.5     # ratio points in cell / # of scans
scan_time = 4          # seconds
buff = 0.05            # meters
port = "/dev/ttyUSB0"  # Serial Port that the LIDAR is connected to
min_thresh = 0.200     # meters
scanNum = 0
scans = pd.DataFrame(columns=["scanNum","x","y","grd_x","grd_y"])
x = np.zeros((360))
y = np.zeros((360))
```

B. Python Code to Define Functions

```
def equiv(angle):
    if isinstance(angle, pd.Series):
        equiv = pd.DataFrame(columns=['angle'], data = angle)
        equiv.loc[(((np.absolute(equiv['angle'])/np.pi) % 2) > 1), 'equiv'] = -1*(-
np.absolute(equiv['angle']) % np.pi)*np.sign(equiv['angle'])
        equiv.loc[(((np.absolute(equiv['angle'])/np.pi) % 2) <= 1) &
(np.absolute((np.absolute(equiv['angle']) % (2*np.pi)) - np.pi) < 0.000001), 'equiv'] =
np.pi*np.sign(equiv['angle'])
        equiv.loc[(((np.absolute(equiv['angle'])/np.pi) % 2) <= 1) &
(np.absolute((np.absolute(equiv['angle']) % (2*np.pi)) - np.pi) >= 0.000001), 'equiv'] =
(np.absolute(equiv['angle']) % np.pi)*np.sign(equiv['angle'])
        return equiv['equiv']
    else:
        equiv = -1*(-np.absolute(angle) % np.pi)*np.sign(angle) if ((np.absolute(angle)/np.pi)
% 2)>1 else (np.pi if (np.absolute(angle) % (2*np.pi)) == np.pi else (np.absolute(angle) %
np.pi))*np.sign(angle)
        return equiv

def plotHeatmaps(heatmap1, color1, data2=[], color2='bwr', data3=[], color3='autumn',
title="Heatmap", xlabel="x", ylabel="y"):
    heatmap2 = np.zeros([len(heatmap1), len(heatmap1)])
    heatmap3 = np.zeros([len(heatmap1), len(heatmap1)])

    for i in range(0, len(data2)):
        heatmap2[data2[i][0],data2[i][1]] = 1

    for j in range(0, len(data3)):
        heatmap3[data3[j][0],data3[j][1]] = 1

    heatmap2 = np.ma.masked_where(heatmap2 < 0.1, heatmap2)
    heatmap3 = np.ma.masked_where(heatmap3 < 0.1, heatmap3)
    plt.imshow(heatmap1, cmap=color1)
    plt.imshow(heatmap2, cmap=color2, interpolation='none')
    plt.imshow(heatmap3, cmap=color3, interpolation='none')
    plt.title(title)
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    plt.show()

def bfs(graph, start, end, visited=[], path=[], level=0, showPlots=False):   # Breadth-First
Search
    # Returns a (i,j)x2x2 jagged array: [Path (2xi), Cells Visited (2xj)]
    # graph: a (nxm) 2D array of 0's and 1's
    # start: coordinates of the start points in the graph. Size nx2.
    # end: coordinates of the end point in the graph. Size 1x2.
    # visited: a list of the points within the graph that have been visited
    # path: a list of the iteration number and points, and each of their previous points,
passed through to reach the start point. Size nx5

    nextStart = []
    fnd = False

    try:
        len(start[0])
    except:
        raise Exception("'start' array must be two-dimensional")

    if len(visited) == 0:
        visited.append(start[0])
    if showPlots:
        plotHeatmaps(graph, 'Greys', data2=visited, data3=[visited[0],end], title='Breadth-
First Search')

    for i in range(0, len(start)):
        for j in range(0,4):
            if j == 0:
                p = [start[i][0]+1,start[i][1]]
            elif j == 1:
                p = [start[i][0],start[i][1]+1]
            elif j == 2:
                p = [start[i][0]-1,start[i][1]]
            elif j == 3:
                p = [start[i][0],start[i][1]-1]

            if p[0] >= 0 and p[1] >= 0 and p[0] < len(graph) and p[1] < len(graph[0]):
                avail = True
                if len(visited) > 0:
                    for k in range(0, len(visited)):
                        if visited[k][0] == p[0] and visited[k][1] == p[1]:
                            avail = False

                if graph[p[0]][p[1]] > 0:
                    avail = False

                if avail:
                    fnd = True
                    path.append([level, start[i][0], start[i][1], p[0], p[1]])
                    nextStart.append(p)
                    visited.append(p)

                    if p == end:
                        curr = end
                        for k in range(len(path)-1, -1, -1):
                            if path[k][3:5] == curr:
                                curr = path[k][1:3]
                            else:
                                del path[k]
                        for l in path:
                            del l[0:3]
                        return path, visited

    if not fnd:
        return path, visited
    bfs(graph, nextStart, end, visited, path, level+1, showPlots=showPlots)
    return path, visited
```

C. *Python Code to Perform and Record the LIDAR Scans*

```
dataSource = input('Enter how the data should be obtained (File/Scan):')
if dataSource == 'File':
    scans = pd.read_csv (r'ScanData.csv')
elif dataSource == 'Scan':
    Obj = PyLidar3.YdLidarX4(port)
    if(Obj.Connect()):
        print(Obj.GetDeviceInfo())
        scans = pd.DataFrame(columns=["scanNum","rho","theta","x","y","grd_x","grd_y"])
        gen = Obj.StartScanning()
        t = time.time() # start time
        while (time.time() - t) < scan_time: # scan for specified amount of time
            data = next(gen)
            scanNum +=1
            for angle in range(0,360):
                data[angle] = data[angle] / 1000.0
                if scanNum != 1:
                    if(data[angle]>min_thresh):
                        x[angle] = (data[angle] - buff) * math.sin(math.radians(angle))
                        y[angle] = (data[angle] - buff) * math.cos(math.radians(angle))
                    else:
                        x[angle] = 0
                        y[angle] = 0

                    newRow = {
                        'scanNum':scanNum,
                        'rho':data[angle],
                        'theta':angle,
                        'x':x[angle],
                        'y':y[angle],
                        'grd_x':0,
                        'grd_y':0
                            }
                    scans = scans.append(newRow, ignore_index=True)
        Obj.StopScanning()
        Obj.Disconnect()

        scans.to_csv(r'ScanData.csv', index=False)

        for i in range(scanNum-1):
            plotData = scans[scans.scanNum == i+2]
            plotData = plotData[['x','y']]
            plt.figure(i+2)
            plt.plot(plotData.x, plotData.y, 'r*')
            plt.plot(0, 0, 'b*')
            plt.axis("equal")
            plt.xlabel("x (mm)")
            plt.ylabel("y (mm)")
            plt.legend(['Scan Data','Robot Location'])
            plt.title("Scan "+str(i+2))
    else:
        Obj.Disconnect()
        print("Error connecting to device")

else:
    print("Invalid Entry, please type one of: 'File' or 'Scan'")
```

D. Python Code to Perform Cluster Analysis

```
mx = scans.max()
scans = scans[scans['scanNum']==mx['scanNum']].reset_index(drop=True)
mx = scans.max()
mn = scans.min()
maxs = pd.DataFrame({'mx':[abs(mx['x']), abs(mn['x']), abs(mx['y']), abs(mn['y'])]})
grid_size = math.ceil(maxs.max()[0]/(clust_grd_size))

scans['grd_x'] = np.floor(scans.x/(clust_grd_size))
scans['grd_y'] = np.floor(scans.y/(clust_grd_size))

numScansDF = scans.groupby(['scanNum']).x.count().reset_index()
numScans = numScansDF.scanNum.count()

grd = scans.groupby(['grd_x','grd_y']).x.count().reset_index()
grd = grd[(grd.grd_x != 0) | (grd.grd_y != 0)]
grd.rename(columns={'x':'numPoints'}, inplace=True)

# Remove Datapoints below Threshold
scansCorr = pd.merge(scans, grd, on=['grd_x','grd_y'])
noise = scansCorr[scansCorr.numPoints/numScans <= clust_thresh].reset_index(drop=True)
scansCorr = scansCorr[scansCorr.numPoints/numScans > clust_thresh].reset_index(drop=True)

# Plot Data Correction
plt.figure(1)
plt.plot(scansCorr.x, scansCorr.y, 'r*')
plt.plot(noise.x, noise.y, 'b+')
plt.axis('equal')
plt.title('Cluster Analysis Data - Correction')
plt.legend(['Good Data','Removed Data'])
plt.ylabel('y (mm)')
plt.xlabel('x (mm)')

# Plot Heatmap
grid_heatmap = np.zeros([2*grid_size,2*grid_size])
for idx, rw in grd.iterrows():
    grid_heatmap[-1*int(round(rw['grd_y']-grid_size+1)), int(round(rw['grd_x']+grid_size))] =
rw['numPoints']
plt.figure(2)
plt.imshow(grid_heatmap, cmap='Greys')
plt.colorbar(plt.pcolor(grid_heatmap, cmap='Greys'))
plt.title('Cluster Analysis Cells')
plt.xlabel('X Cell')
plt.ylabel('Y Cell')
plt.show()
```

E. Python Code to Perform Occupancy Mapping

```
occ_grid_size = math.ceil(maxs.max()[0]/(tess_grd_size))*2

occ = pd.DataFrame(0, index=np.arange(occ_grid_size**2), columns=['occ_x','occ_y','M','C'])
occ['occ_x'] = (np.floor(occ.index/occ_grid_size)+1)*tess_grd_size
occ['occ_y'] = (np.mod(occ.index, occ_grid_size)+1)*tess_grd_size
occ['cent_x'] = occ['occ_x'] - tess_grd_size/2
occ['cent_y'] = occ['occ_y'] - tess_grd_size/2
occ['r'] = np.sqrt((occ['cent_x'] - ((occ_grid_size/2)*tess_grd_size))**2 + (occ['cent_y'] -
((occ_grid_size/2)*tess_grd_size))**2)
occ['pnt_angle'] = np.arctan2(occ['cent_y'] - ((occ_grid_size/2)*tess_grd_size), occ['cent_x']
- ((occ_grid_size/2)*tess_grd_size))
occ['j'] = 0

occ_lines = scansCorr.copy()
occ_lines = occ_lines[[
    'scanNum',
    'x',
    'y'
]]
occ_lines['x'] = occ_lines['x'] + ((occ_grid_size/2)*tess_grd_size)
occ_lines['y'] = occ_lines['y'] + ((occ_grid_size/2)*tess_grd_size)
occ_lines['a'] = (occ_lines['x'] - ((occ_grid_size/2)*tess_grd_size))
occ_lines['b'] = -(occ_lines['a']**2 / (occ_lines['y'] - ((occ_grid_size/2)*tess_grd_size)))
occ_lines.loc[occ_lines['a'] == 0, 'a'] = 1
occ_lines['c'] = -((occ_lines['a']*occ_lines['x']) + (occ_lines['b']*occ_lines['y']))
occ_lines['line_angle'] = np.arctan2(occ_lines['y'] - ((occ_grid_size/2)*tess_grd_size),
occ_lines['x'] - ((occ_grid_size/2)*tess_grd_size))
occ_lines['h_x'] = np.sqrt((occ_lines['x'] - ((occ_grid_size/2)*tess_grd_size))**2 +
(occ_lines['y'] - ((occ_grid_size/2)*tess_grd_size))**2)
occ_lines['j'] = 0

occ_lines = pd.merge(occ, occ_lines, on=['j'])
occ_lines.drop(columns=['j'], inplace=True)

occ_lines['d_line'] = (abs((occ_lines['a']*occ_lines['cent_x']) +
(occ_lines['b']*occ_lines['cent_y']) + occ_lines['c'])) / np.sqrt(occ_lines['a']**2 +
occ_lines['b']**2)
occ_lines['d_end'] = np.sqrt((occ_lines['cent_x'] - occ_lines['x'])**2 + (occ_lines['cent_y'] -
occ_lines['y'])**2)


# Convert lines and points to horizontal to determine if point is near line segment
occ_lines['pnt_h_x'] = ((occ_grid_size/2)*tess_grd_size) +
(occ_lines['r']*np.cos(occ_lines['pnt_angle'] - occ_lines['line_angle']))
occ_lines = occ_lines[
    (occ_lines['pnt_h_x'] >= ((occ_grid_size/2)*tess_grd_size)) &
    (occ_lines['pnt_h_x'] <= occ_lines['h_x'] + ((occ_grid_size/2)*tess_grd_size))
]

occ_lines = occ_lines[(occ_lines['d_line'] <= ray_cast_thresh) | (occ_lines['d_end'] <=
ray_cast_thresh)]
occ_lines['C'] = 1
occ_lines['M'] = -1
occ_lines.loc[occ_lines['d_end'] <= ray_cast_thresh, ['M']] = 1
```

```python
occ = occ_lines.groupby(['occ_x','occ_y','cent_x','cent_y'])['M','C'].apply(lambda x :
x.astype(int).sum()).reset_index()
occ['occupied'] = occ['M']/occ['C']

# Plot Heatmaps
occ_heatmap = np.zeros([occ_grid_size,occ_grid_size])
for idx, rw in occ.iterrows():
    occ_heatmap[-(int(round(rw['occ_y']/(tess_grd_size)))-1),
int(round(rw['occ_x']/(tess_grd_size)))-1] = rw['occupied']
plt.figure(1)
plt.imshow(occ_heatmap, cmap='Greys')
plt.colorbar(plt.pcolor(occ_heatmap, cmap='Greys'))
plt.title('Occupancy Map - Probability')
plt.xlabel('X Cell')
plt.ylabel('Y Cell')
plt.show()

for idx, rw in occ.iterrows():
    if rw['C'] == 0:
        occ_heatmap[-(int(round(rw['occ_y']/(tess_grd_size)))-1),
int(round(rw['occ_x']/(tess_grd_size)))-1] = 0
    elif rw['occupied'] > occ_thresh:
        occ_heatmap[-(int(round(rw['occ_y']/(tess_grd_size)))-1),
int(round(rw['occ_x']/(tess_grd_size)))-1] = 1
    else:
        occ_heatmap[-(int(round(rw['occ_y']/(tess_grd_size)))-1),
int(round(rw['occ_x']/(tess_grd_size)))-1] = -1
plt.figure(2)
plt.imshow(occ_heatmap, cmap='Greys')
plt.colorbar(plt.pcolor(occ_heatmap, cmap='Greys'))
plt.title('Occupancy Map - Threshold')
plt.xlabel('X Cell')
plt.ylabel('Y Cell')
plt.show()
```

F. Python Code to Execute Path Planning Functions

```
start = [occ_grid_size-(int(round((occ_grid_size/2)))-1), int(round((occ_grid_size/2)))-1]

end = [33, 40]
res = bfs(occ_heatmap, [start], end, showPlots=False)
plotHeatmaps(occ_heatmap, 'Greys', data2=res[1], data3=res[0], title='Breadth-First Search
Results')
```

G. Python Code to Move the Robot Along the Path

```
gpg = EasyGoPiGo3()

conv = [2.9022, 1.9972]
pose = [start[0], start[1], 0]
path = res[0].copy()

for i, rw in enumerate(path):

    if path[i][1] == pose[1] and path[i][0] == pose[0]:
        theta = pose[2]
    else:
        theta = equiv(np.arctan2(path[i][1] - pose[1], path[i][0] - pose[0]))
        gpg.turn_degrees(-(conv[0]*np.degrees(theta - equiv(pose[2])) + conv[1]))


    if path[i][1] - pose[1] != 0:
        gpg.drive_cm(conv[0]*abs(path[i][1] - pose[1])*tess_grd_size*100 + conv[1])
    else:
        gpg.drive_cm(conv[0]*abs(path[i][0] - pose[0])*tess_grd_size*100 + conv[1])

    pose = [path[i][0], path[i][1], theta]
```

H. Physical Robot Setup