NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

AUTONOMOUS OBSTACLE AVOIDANCE AND CONTROL USING VOXEL SEGMENTATION OF 3D LIDAR DATA

by

Justin T. Bracci

September 2021

Thesis Advisor: Xiaoping Yun
Co-Advisor: James Calusdian

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			*Form Approved OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** September 2021	**3. REPORT TYPE AND DATES COVERED** Master's thesis

4. TITLE AND SUBTITLE AUTONOMOUS OBSTACLE AVOIDANCE AND CONTROL USING VOXEL SEGMENTATION OF 3D LIDAR DATA	**5. FUNDING NUMBERS**
6. AUTHOR(S) Justin T. Bracci	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Naval Information Warfare Center Pacific, San Diego, CA 92152	**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release. Distribution is unlimited.	**12b. DISTRIBUTION CODE** A

13. ABSTRACT (maximum 200 words)

The purpose of this work was to determine if a single 3D lidar sensor could provide enough data to conduct obstacle detection and avoidance for a small ground-based autonomous vehicle in an indoor environment. This work was based on previous Naval Postgraduate School work with simultaneous localization and mapping using a 2D lidar sensor and a 3D time of flight camera. A voxel-based point cloud filtering method was used to interpret data and classify objects as large, small, or negative. The data was then used as an input to a control algorithm using a potential field control model to navigate around the identified obstacles. The classification and control algorithm was proven successful through four separate experiments, and a definition for a small object was developed. Areas for future study were identified to include the development of a localization method using a single 3D lidar sensor, the implementation of the obstacle avoidance algorithm on an autonomous platform with six degrees of freedom, and the development of a path planning algorithm based on an initial point cloud.

14. SUBJECT TERMS light detection and ranging, lidar, 3D lidar, simultaneous localization and mapping, SLAM, voxel, point cloud, autonomous navigation, segmentation	**15. NUMBER OF PAGES** 121
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UU

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release. Distribution is unlimited.

AUTONOMOUS OBSTACLE AVOIDANCE AND CONTROL USING VOXEL SEGMENTATION OF 3D LIDAR DATA

Justin T. Bracci
Captain, United States Marine Corps
BSME, California Polytechnic State University San Luis Obispo, 2014

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
September 2021

Approved by: Xiaoping Yun
Advisor

James Calusdian
Co-Advisor

Douglas J. Fouts
Chair, Department of Electrical and Computer Engineering

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

The purpose of this work was to determine if a single 3D lidar sensor could provide enough data to conduct obstacle detection and avoidance for a small ground-based autonomous vehicle in an indoor environment. This work was based on previous Naval Postgraduate School work with simultaneous localization and mapping using a 2D lidar sensor and a 3D time of flight camera. A voxel-based point cloud filtering method was used to interpret data and classify objects as large, small, or negative. The data was then used as an input to a control algorithm using a potential field control model to navigate around the identified obstacles. The classification and control algorithm was proven successful through four separate experiments, and a definition for a small object was developed. Areas for future study were identified to include the development of a localization method using a single 3D lidar sensor, the implementation of the obstacle avoidance algorithm on an autonomous platform with six degrees of freedom, and the development of a path planning algorithm based on an initial point cloud.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

DoF	degree of freedom
FOV	field of view
GNSS	global navigation satellite systems
INS	inertial navigation systems
ISR	information, surveillance, and reconnaissance
lidar	light detection and ranging
PDF	probability density function
ROI	region of interest
ROS	robot operating system
SLAM	simultaneous localization and mapping
S&T	science and technology
SUAV	small unmanned aerial vehicle
ToF	time of flight

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I would like to thank my advisors, Professor Xiaoping Yun and Dr. James Calusdian, for all the help and support that they offered me throughout the thesis process. Professor Yun was a constant source of guidance in both technical and practical areas and ensured my research stayed on track and relevant. Dr. Calusdian was a constant presence in the lab and a patient sounding board for my ideas and concerns.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

The automotive industry in recent years has seen a surge in the development of obstacle avoidance and navigation techniques for autonomous vehicles. Among the multitude of different sensors used in this endeavor, light detection and ranging sensors (lidar) have become the pre-eminent tool to provide data about an environment. Recent technological developments by companies such as Velodyne and Ouster have seen a rise in the availability of 3D, 360-degree field of view lidar sensors [1], [2]. These sensors provide a wealth of data that can be used in solving the problem of autonomous navigation and obstacle avoidance. The viability of using a single 3D lidar sensor to detect different types of objects is investigated in this thesis. Unlike the automotive industry, the focus of this research is on small, indoor unmanned vehicles. In this chapter, the motivation behind this research, past and related work in this field, and ultimately the purpose and goal of this thesis are presented.

A. MOTIVATION

Current autonomous vehicle control and obstacle avoidance are achieved using a variety of sensors. The use of multiple sensors operating simultaneously to provide input data to a single system, also known as sensor fusion [3], requires a physically large autonomous platform and power source to accommodate all the sensors. This constraint limits the deployment of autonomous vehicles at the tactical level, specifically in the realm of autonomous simultaneous localization and mapping (SLAM) in complex indoor environments [4]. The 2018 United States Marine Corps Science and Technology (S&T) Strategic Plan identifies joint capabilities area two as the development of battlespace awareness and calls for a comprehensive information, surveillance, and reconnaissance (ISR) enterprise to provide intelligence to warfighters at every level [5]. The intelligence collection capabilities of autonomous vehicles at the tactical level will directly align with intelligence S&T objectives two and five. Furthermore, S&T Objective four calls for the development of advanced robotic systems in support of ground maneuver [5].

In previous thesis work, Payne identified the need for Marine Corps tactical teams to have prior knowledge of a building or small, enclosed urban space before entering the environment and proposed an autonomous SLAM solution [4]. The results from Payne's work identified the need for a more robust obstacle avoidance algorithm using emerging 3D lidar technology [4]. Most autonomous vehicle work with 3D lidar sensors has been done for the automotive industry, therefore a specific analysis of the capability of 3D lidar sensors to identify obstacles in a confined indoor or urban environment is needed.

B. PREVIOUS WORK

The use of lidar sensors to detect obstacles is a common practice and has been implemented on numerous different types of platforms. Lidar sensors are classified as active sensors and are able to return position data of an object relative to the sensor [6]. As previously stated, this thesis builds on past work at the Naval Postgraduate School (NPS) in the realm of autonomous navigation and SLAM. The work of Miyakawa [7] explored the capabilities of an autonomous vehicle to navigate in an outdoor environment using a potential field model and Global Navigation Satellite Systems/Inertial Navigation Systems (GNSS/INS) to identify and avoid obstacles. Miyakawa utilized two 2D lidar sensors to detect large and small obstacles as well as obstacles below an established ground plane such as stairs, curbs, and holes [7]. Payne, in addition to SLAM, implemented a similar obstacle avoidance method using a 2D lidar sensor and a 3D Time-of-Flight (ToF) camera or scanner-less lidar [4].

Outside NPS, other research has been conducted on the use of 3D lidar sensors as part of a larger sensor fusion suite for autonomous navigation and obstacle avoidance [8]–[10]. Further research has been conducted on the implementation of 3D lidar sensors on small unmanned aerial vehicles (SUAV) for obstacle avoidance [11], [12]. Most of this research involves multiple sensors mounted on large autonomous platforms accomplishing other tasks in conjunction with obstacle avoidance and navigation such as SLAM.

The research presented in [9] is of particular interest because of the use of voxels to interpret the environment surrounding the sensor. A commonly encountered obstacle when working with lidar data sets is the large size of lidar data and consequently the large

amount of computing power needed to process the data [4], [13]. If the size of the data sets can be reduced, then the computing power needs can also be reduced. Simplifying the interpretation of the environment with the use of voxels has the potential to speed up computation times to better achieve real time obstacle detection and avoidance at the cost of accuracy and detail.

The research detailed in [14] looked specifically at the use of 3D lidar sensors to identify what is defined as a negative obstacle or an obstacle that is "below the ground, with a negative height" [14]. Some examples of negative obstacles include holes, large surface cracks, and edges of stairs and floor levels. The conclusion drawn from this research and the afore mentioned research is that any obstacle detection sensor would need to identify three types of obstacles: large, small, and negative height obstacles.

C. PURPOSE AND GOAL

The purpose of this thesis is to determine if a single 3D lidar sensor can provide sufficient data to conduct obstacle detection and avoidance for a small ground-based autonomous vehicle in an indoor environment. The size and scope of obstacles that can be detected by the sensor will be determined. A voxel-based 3D point cloud manipulation algorithm will be developed to provide real-time obstacle detection and avoidance. The greater purpose of the obstacle detection and avoidance algorithm is to facilitate the navigation of an autonomous vehicle in an indoor environment to conduct SLAM.

The ultimate goal of this thesis is to produce a working algorithm that, coupled with a single 3D lidar sensor, has the capability to ensure that a ground-based autonomous robot is capable of obstacle detection and avoidance when moving from a designated start and end point. Ideally the robot would be reliant on the single 3D lidar sensor to accomplish all navigation needs. However, for the sake of simplicity and focus, a dead-reckoning method for localization was implemented using the built-in wheel encoders of the robot as detailed in Chapter II. Three-dimensional point cloud localization techniques exist to support this requirement as detailed in [4]. The algorithm developed in this research will account for six degrees of freedom to allow for implementation on a SUAV autonomous platform.

This thesis is composed of six chapters and two appendices. Chapter II covers the hardware and software used for this research and includes an in-depth explanation of the lidar sensor. Chapter III covers the small object identification capabilities of the lidar sensor and provides a definition for small obstacles in the case of this research. In Chapter IV, an analysis of the method used for the identification of negative height obstacles and subsequent test results is presented. Chapter V includes an overall description of the robot control algorithm and obstacle classification technique used. The results of practical application of the robot in an indoor environment are presented in Chapter VI. Finally, Chapter VII includes a summary, conclusion, and recommendations for future study. Appendices A and B include the code for the implementation of the control algorithm.

II. HARDWARE AND SOFTWARE

A. HARDWARE

1. Pioneer 3-DX Mobile Robot

The pioneer 3-DX (P3-DX) mobile robot is a three-wheeled mobile robot made by Adept Mobile robots and comes ready to run with a suite of front and rear facing sonar sensors, wheel encoders, a front and rear facing segmented bumper array, and a programmable microcontroller [15]. The P3-DX is shown in Figure 1. The NPS Controls Laboratory has developed a series of MATLAB functions to interface and control the P3-DX [16] and the robot has been used in multiple past NPS theses [4], [7], [17]. While the P3-DX is not the most up to date model, the experience gained from working with the robot has ensured ease of use and fusion between the robot and other hardware and software. The three-wheel design of the robot allows for a turn radius of 0 cm [15] making operating in tight, indoor environments ideal. The P3-DX has a limited ability to traverse rough terrain, as highlighted later in Chapter III.

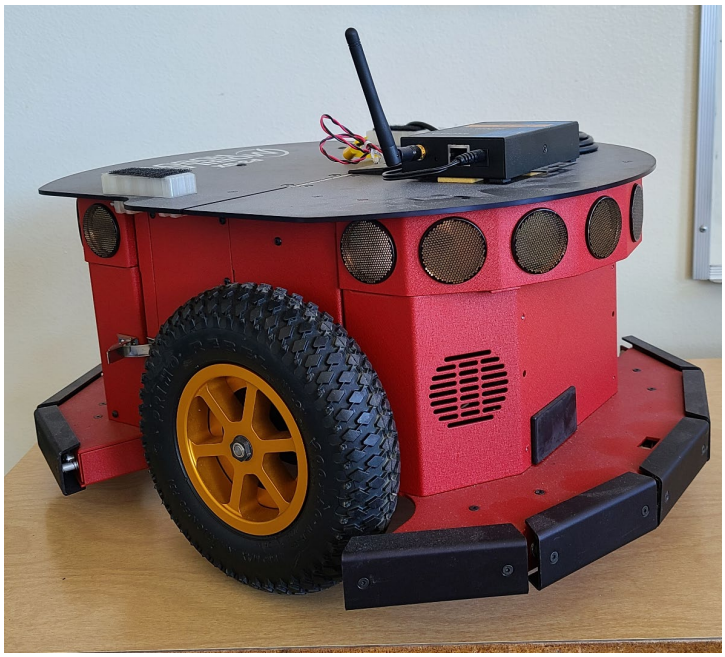


Figure 1. Pioneer 3-DX

2. Samsung R580 Laptop

Based on the observations in [4] regarding the computational requirements associated with large lidar point cloud data sets, a Samsung R580 laptop was chosen to interface with the robot. The R580 was chosen based on availability and the 512MB GeForce GT 310M discrete graphics card on the computer. In theory the use of the graphics card might be able to mitigate some of the computational requirements of working with large point clouds. However, after installation, the Ubuntu 20.04 LTS suite was revealed to not support the GeForce 310M graphics card. As no other laptops were readily available, the Samsung R580 was used and the 2.26 GHz intel Core i5 processor, 4GB of RAM, and 500GB hard drive of the R580 proved sufficient for the conduct of this research. The R580 was connected to the P3-DX using a serial to USB connector and connected to the OS1-16 using an ethernet connection. The laptop itself was mounted on top of the P3-DX and to the rear of the OS1-16 sensor to mitigate obstruction of the sensor. Shown in Figure 2 is the complete robot setup with the R580 clearly visible on top of the robot.

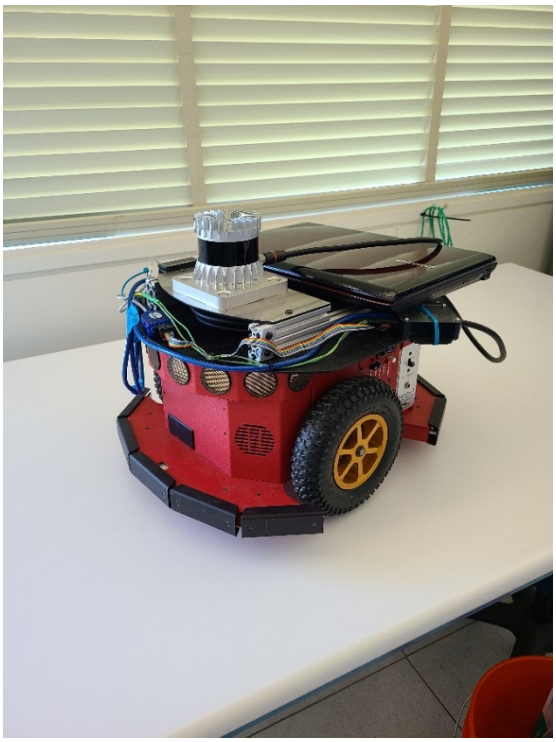


Figure 2. P3-DX Robot with Lidar Sensor and R580 Control Computer

3. Microsoft Surface Pro 6

A Microsoft Surface Pro 6 was used as the primary computer to interface wirelessly with the R580 and thus the P3-DX robot. The Microsoft Surface Pro had a 1.6 GHz Intel Core i5 processor with 8 GB of RAM and a 128 GB hard drive. The Microsoft Surface Pro ran Microsoft Windows 10 64 bit and the built in Windows 10 secure shell (SSH) client [18] was used to connect to and send commands to the R580 laptop. The Microsoft Surface Pro was also capable of wirelessly receiving output data from the OS1-16 sensor via the robot operating system (ROS) network and displaying the point cloud from the sensor on the Surface Pro. This point cloud could be adjusted according to the needs of the user and was primarily used to view the filtered point cloud that the robot was using for obstacle avoidance and for troubleshooting purposes. This capability was also in keeping with the overall motivation for this research to have a wireless operator monitor the data collected by the robot as detailed in [4]. MATLAB 2020b was used to program and implement this feature. A detailed description of the control interface between the robot, lidar sensor, Microsoft Surface, etc., can be found in Chapter V.

4. DC 12V to 24V Step Up Converter

A 5-amp, 120-watt 12-volt to 24-volt DC step up converter was used to supply power to the 24-volt OS1-16 sensor from the 12-volt P3-DX.

5. Ouster OS1-16 Gen 1 Mid-Range High-Resolution Imaging Lidar

The OS1-16 lidar sensor is a high resolution 360-degree field of view lidar sensor designed for all-weather indoor and outdoor use [19]. The OS1-16 transmits data to the Samsung R580 laptop via an ethernet connection and the data is read into MATLAB 2020b as a ROS point cloud message. A comprehensive understanding of the capabilities and limitations of the sensor is necessary to inform the decision for the definition of a detectable small object as detailed in Chapter III. The relevant characteristics of the lidar sensor [19] are provided in Table 1. Shown in Figure 3 is the Ouster OS1-16 used in this research.

Table 1. OS1-16 Characteristics. Adapted from [19].

Parameter	Value	Comments
Maximum Range	50 m	90% detection probability with a target of 10% Lambertian reflectivity
Minimum Range	0.8 m	For returned point cloud data
Range Accuracy	±5 cm	For Lambertian targets
Precision	0.8 → 1 m ±1 cm 1 → 20 m ±1.1 cm 20 → 50 m ±3 cm >50 m ±5 cm	10% Lambertian reflectivity; 1 standard deviation
Range Resolution	0.3 cm	
Field of View	Vertical: +16.6° to -16.6° (33.2°) Horizontal: 360°	
Angular Sampling Accuracy	Vertical: ±0.01° Horizontal: ±0.01°	

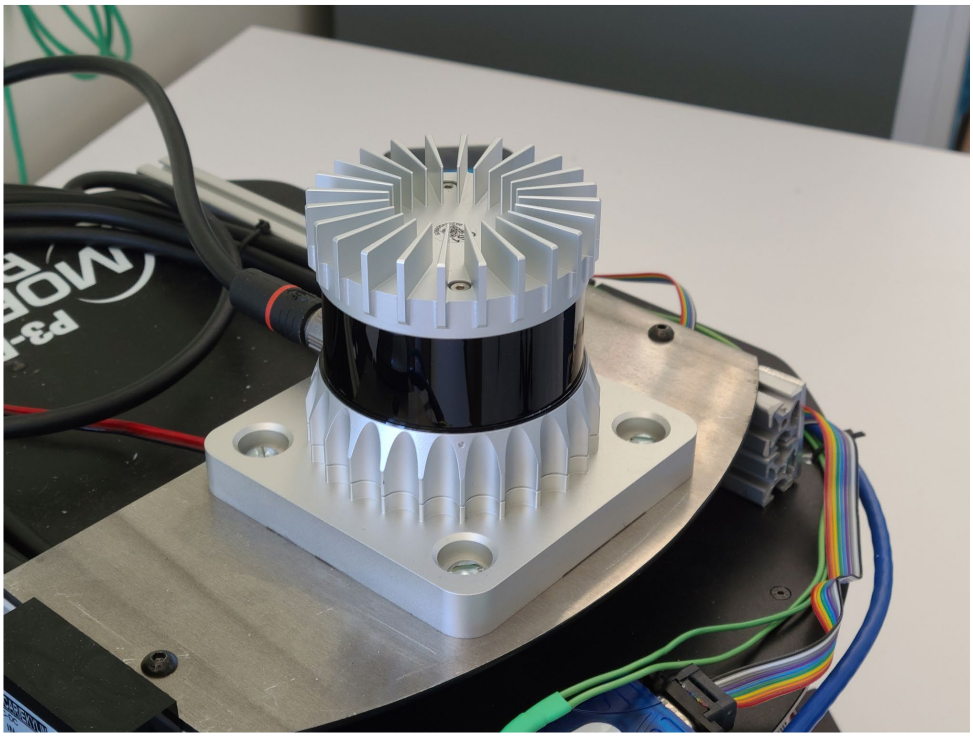


Figure 3. Ouster OS1-16

a. Maximum Range

The published maximum range of the sensor is representative of the furthest distance that the sensor can receive point cloud data from a target with 10% Lambertian reflectivity. In the industry, the standard tends to be between 90% and 10% Lambertian reflectivity with the more conservative maximum range value in relation to a target with low Lambertian reflectivity [20]. The published maximum range can be misleading when considering the differences between detection and identification as discussed later in Chapter III. There are three primary variables that affect the range of a sensor: type of target, the probability of detection, and the amount of sunlight or "noise" [20]. A more detailed explanation of range as it pertains to signal strength is given in Figure 4.

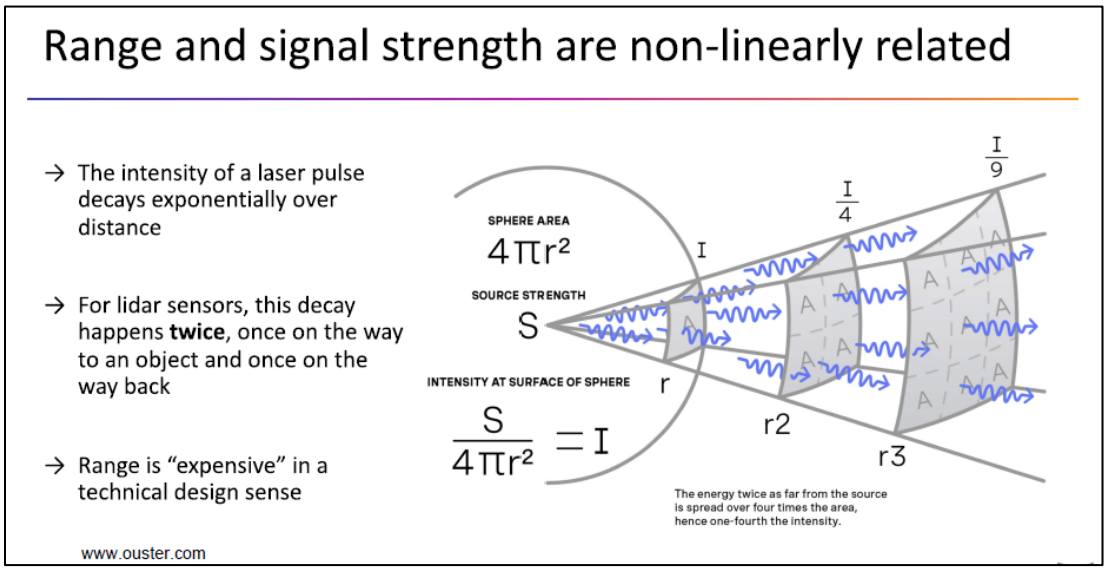


Figure 4. Signal Strength and Range. Source: [20].

Lambertian reflectance is a categorization of a surface based on the reflective properties of that surface. As stated by Tatum, "the radiance of a Lambertian surface is independent of the angle from which it is viewed" [21]. Most natural surfaces are Lambertian [22]. A surface with 100% Lambertian characteristic is considered to be ideal matte, and a surface with a low Lambertian characteristic would be considered very reflective [23]. Examples of Lambertian objects across the range of Lambertian reflectance

are given in Figure 5, and examples of the three primary types of targets that a lidar sensor is expected to observe are given in Figure 6 [20].

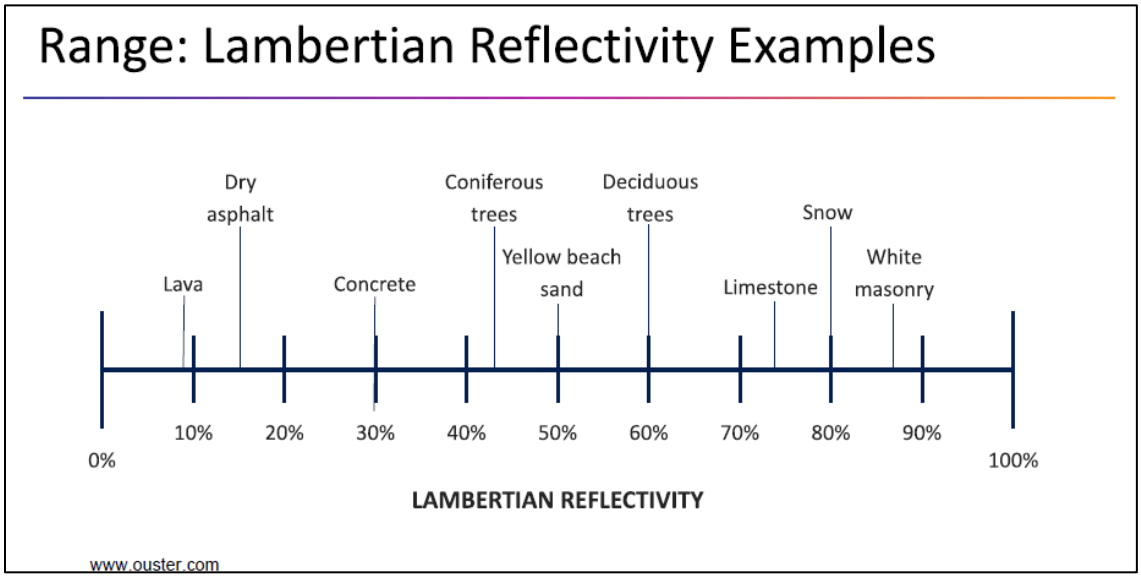


Figure 5. Range of Lambertian Reflectance. Source: [20].

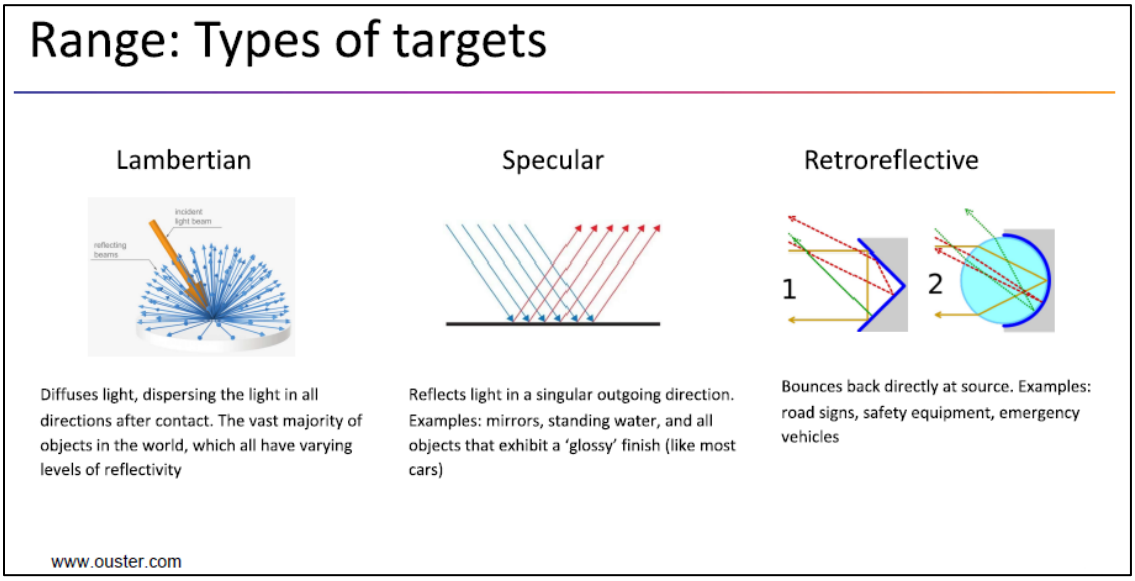


Figure 6. Expected Lidar Targets. Source: [20].

b. Minimum Range

The minimum range is the minimum distance from an object that the sensor can be and return point cloud data. Any part of an object that is inside of this minimum range will not be seen by the sensor.

c. Range Accuracy

Accuracy is an evaluation of how close a measurement is to the true value of that measurement [20]. In a practical sense, the accuracy of the sensor determines what can visually be interpreted as wall straightness [20]. Shown in Figure 7 is an example of an uncalibrated sensor and the resulting curve in what is in reality a straight wall.

d. Precision

Precision is a metric for how repeatable consecutive measurements are relative to each other [20]. The value given in Table 1 for precision is the straight line distance between two consecutive points from the same beam. In practical terms, the precision can be thought of as the thickness of a wall [20]. A sensor with high precision will produce relatively thin walls. The parameters of precision and accuracy are illustrated in Figure 7.

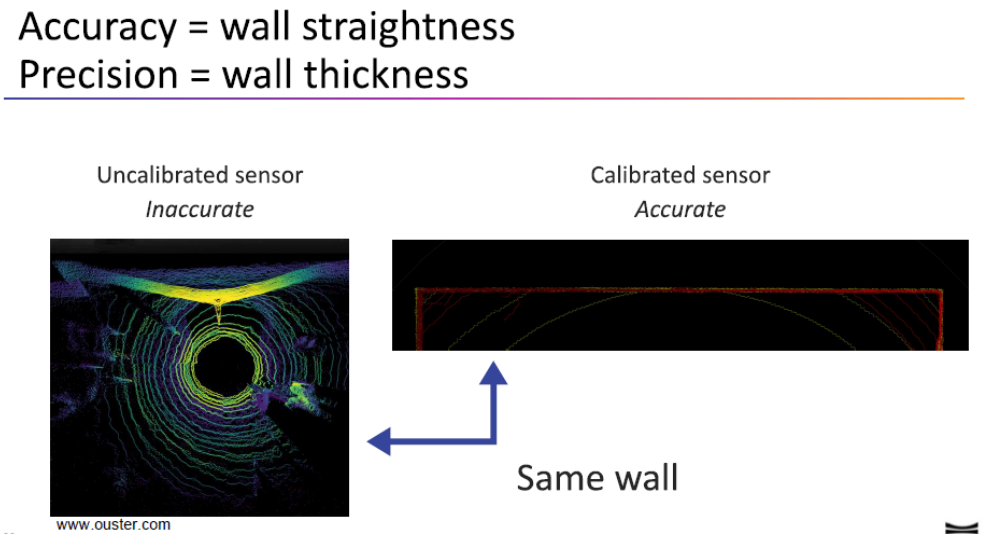


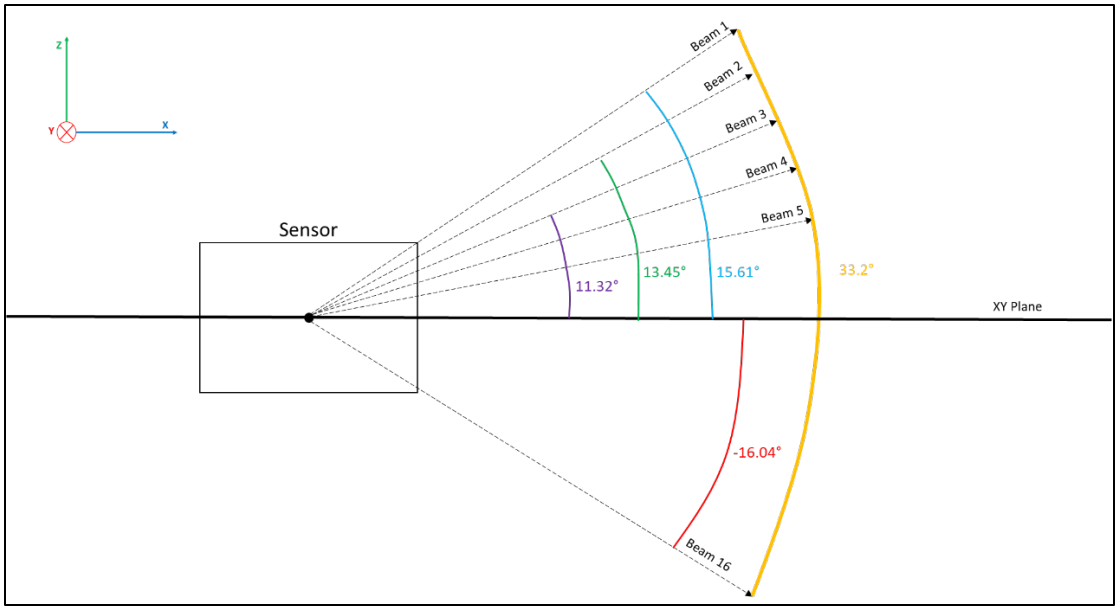
Figure 7. Accuracy and Precision. Source: [20].

e. Field of View

The overall field of view of the lidar sensor is the elevation covered by the beams of the sensor. In the case of the OS1-16, there are 16 total beams covering the total field of view of 33.2° [19]. The angular elevation of each beam relative to the XY plane of the sensor, as shown in Figure 8, is given in Table 2.

Table 2. Beam Angle

Beam (from top to bottom)	Angle (in degrees)
1	15.61
2	13.45
3	11.32
4	9.21
5	7.11
6	5.02
7	2.94
8	0.85
9	-1.24
10	-3.33
11	-5.42
12	-7.51
13	-9.62
14	-11.73
15	-13.86
16	-16.04



Beams 6 through 15 omitted for simplicity and clarity.

Figure 8. Sensor Beam Orientation

f. Angular Sampling Accuracy

The angular sampling accuracy is the horizontal and vertical angular accuracy of the returned angle for each point identified by the sensor. The vertical resolution of the sensor is determined by the number of beams and the angular spacing between each of the beams. In the case of the OS1-16, the angular resolution varies depending on the two beams in question but is about 2.1° between adjacent beams. The horizontal resolution of the sensor is variable, depending on the chosen rotation rate of the sensor and can be set to 512, 1024, or 2048. Given the limitations imposed by the processing power available during this research, the data collected during this work was limited to a horizontal resolution of 1024.

g. Range Resolution

The range resolution of a sensor defines how well the sensor can differentiate between two targets that are very close together [24]. In the case of the OS1-16, this translates to two targets with the same bearing but different angles relative to the sensor [24], as shown in Figure 9, where ΔR is the range resolution.

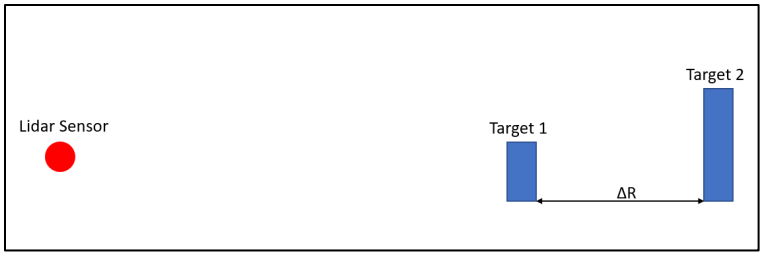


Figure 9. Range Resolution

h. Lidar Data Interpretation

The data from the sensor can be visually interpreted as an NxM matrix with N representing the number of beams and M representing the chosen horizontal resolution of the sensor. This can be visually interpreted as a panoramic view of a fixed resolution depth, signal, and ambient image [25], as shown in Figure 10. A panoramic photo of the scan shown in Figure 10 is given in Figure 11.

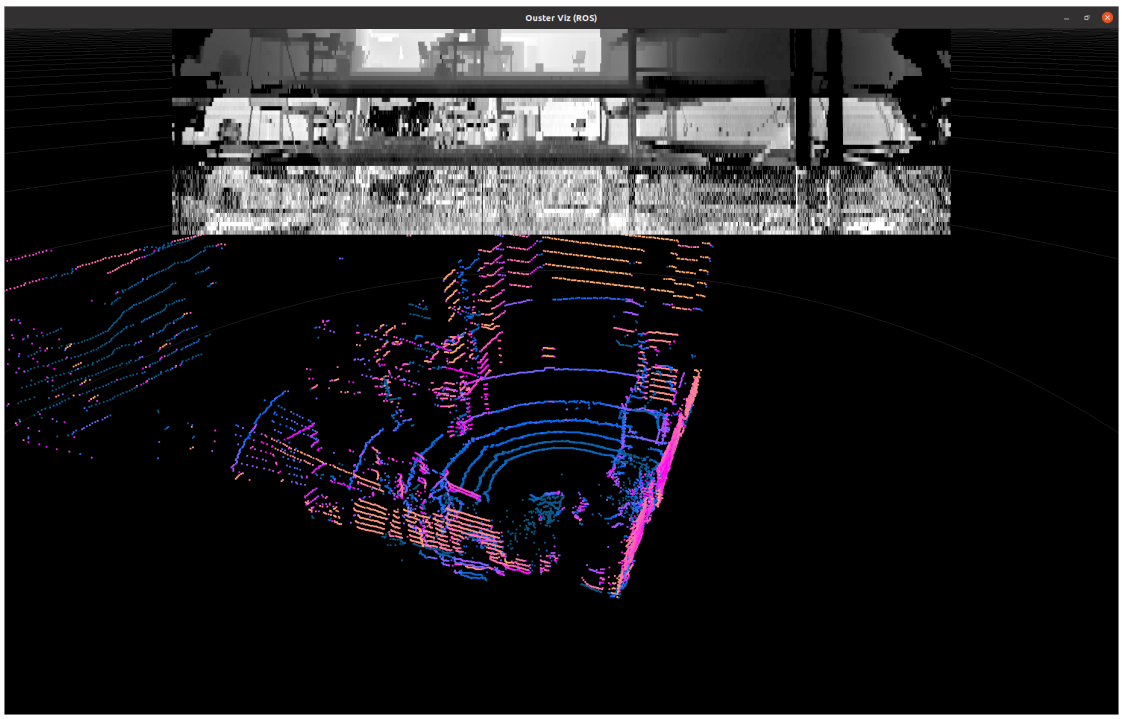


Figure 10. OS1-16 Fixed Resolution Depth Image and Corresponding Point Cloud

Figure 11. Panoramic Photo

B. SOFTWARE

1. MATLAB

MATLAB is a programming platform centered around a matrix-based language that can be used by scientists, engineers, and analysts to perform computational mathematics relative to an innumerable number of areas of study [26]. MATLAB is ideally suited for the easy development of algorithms and for the proof of concepts and ideas. MATLAB also allows for the use of innumerable toolboxes, each geared towards a specific area of study. Of particular note is the newly introduced MATLAB Lidar Toolbox which provides numerous tools for processing, interpreting, and visualizing point cloud data for autonomous systems [27]. All the algorithms described and developed in this thesis were created and implemented using MATLAB version 2020b. The use of MATLAB also allowed for easy access to the lidar sensor data with the use of ROS and the previously developed P3-DX interface functions [16]. The MATLAB ROS Toolbox allows for the integration of the disparate systems of the robot [28]. Of note, the MATLAB ROS Toolbox was originally developed for the version of ROS, ROS Melodic. This research utilized the latest version of ROS, ROS Noetic. No major issues were discovered except that in order to initiate the ROS environment in MATLAB, the default version of Python in Ubuntu 20.04 had to be changed to Python 2. The default version of Python used with Ubuntu 20.04 is Python 3.

2. The Robot Operating System (ROS)

The Robot Operating System or ROS is a free software suite designed to allow for the simple and seamless integration of multiple sensors and systems to form a more complex robotic system [29]. Like MATLAB, ROS is composed of a collection of tools

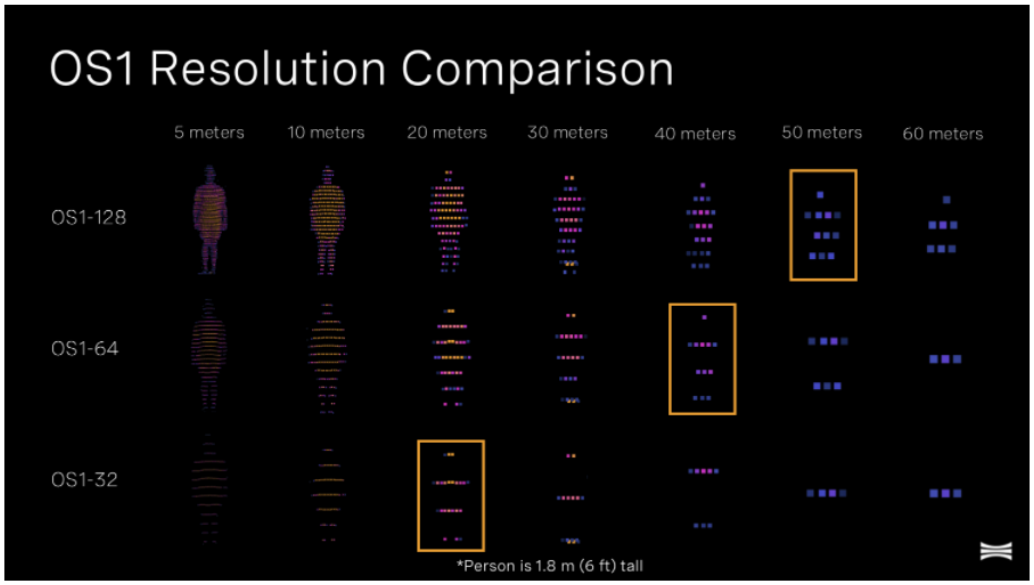
that allow for a user to integrate different systems into a single cloud like network environment [28], [29]. The basic structure of ROS is the ROS master environment which contains multiple nodes or sensors/systems [28]. Each of these nodes can publish data or receive data from what are known as topics [28]. A more in-depth analysis of this framework is given in Chapter V. For the scope of this research, ROS enabled the lidar sensor to transmit data in a usable format to MATLAB for implementation in the obstacle avoidance algorithm. ROS also allowed for the Microsoft Surface laptop to wirelessly receive the lidar point cloud data.

III. SMALL OBJECT DETECTION AND IDENTIFICATION

The objective of this phase of the research was to determine how small of an object the OS1-16 could detect and clearly define what constituted a small object. For this thesis, the term *place* will refer to the sensor receiving a laser return from one of the laser beams or beams, and providing a set of XYZ coordinates in the sensor frame for that point.

A. DETECTION AND IDENTIFICATION

Ouster, the company that produces the OS1-16, has produced their own interpretation of small object detection. Based on the widespread use of lidar sensors for autonomous driving [30], Ouster has identified a small object as a 1.8 m tall pedestrian [31]. Reliable detection and identification according to Ouster is defined as the ability to place four horizontal lines on a 1.8 m tall pedestrian [31]. A comparison of three different Ouster lidar sensors with different vertical resolutions and the effective range of each sensor based on the Ouster definition of detection is shown in Figure 12.



High resolution increases a sensor's effective range. These OS1 lidar sensors all have the same maximum range, but their effective range is determined by their resolution.

Figure 12. Resolution and Detection. Source: [31].

1. Detection

This work defines detection as the ability of a sensor to identify the presence of an obstacle. For this work and in the case of the OS1-16 sensor, detection is established by a threshold of at least one beam placing at least two points on an object. An example of the detection of an object is given in Figure 13. In this case a 3.0 cm x 3.2 cm x 1.6 cm block was detected by beam 10 approximately 0.85 m from the sensor. As can be seen in Figure 13, the presence of the object is indicated by the two points returned from beam 10. This data is enough for the sensor and any attached systems to be alerted to the presence of a potential obstacle, but not enough to identify the lidar return as an obstacle. This concept is important because depending on the environment, a lidar sensor can return multiple erroneous points that may or may not be true obstacles. This phenomenon is discussed further in Chapter VI.

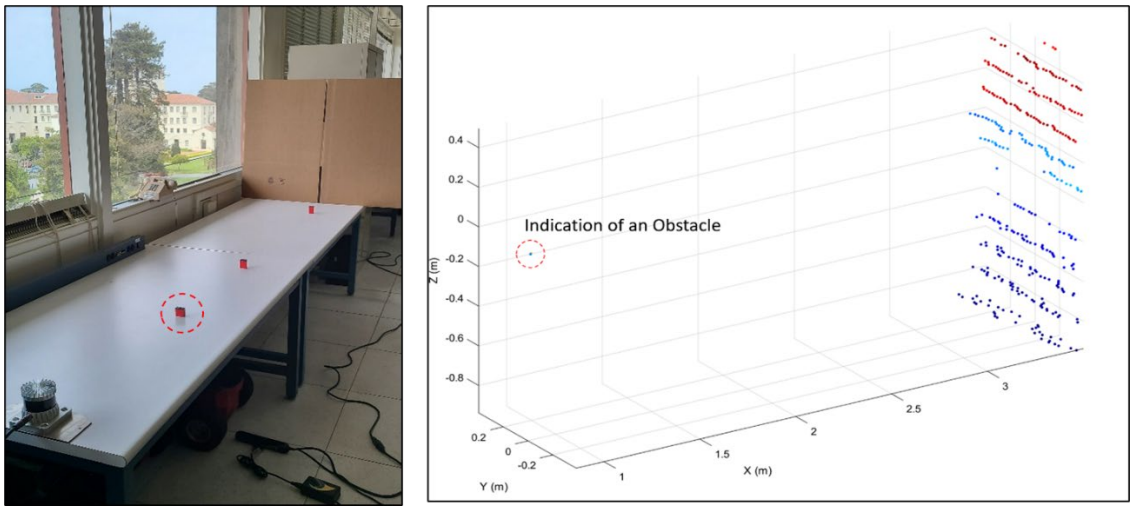


Figure 13. Indication of Object at 0.8 m Distance Picture and Point Cloud

2. Identification

In this work, identification is defined as the ability of the sensor to detect and identify an object as an obstacle. For this work, the identification of an obstacle is defined as at least two beams placing at least two points each on an object for a total of four points.

Based on this definition of identification, the smallest object that would be detectable by the sensor would be an object large enough for the sensor to place at least four points from two different beams on the object. An example of this concept is given in Figure 14.

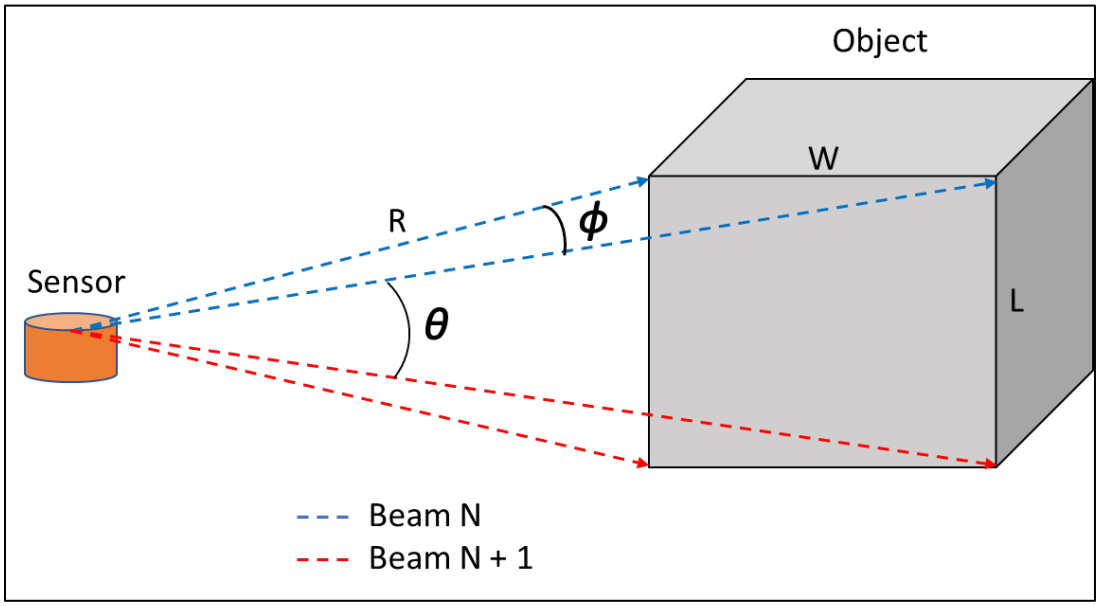


Figure 14. Identification of an Object

B. SMALL OBJECT DEFINED

The definition of a small object depends on environmental considerations, the employment of the sensor, and the physical capabilities of the sensor. Chapter II covered the capabilities and limitations of the sensor in depth, including angular sampling accuracy and range accuracy, which directly influence the values of W and L, as seen in Figure 14. For this work, the sensor was mounted on a mobile platform with a ground clearance of approximately 3.9 cm. Any object greater than 3.9 cm is an obstacle for the mobile platform. Based on this evaluation, the sensor needed to be able to detect an object with a height of at least 3.9 cm.

To define the dimensions of the smallest detectable object, the minimum values of W and L from Figure 14 needed to be determined. The value of ϕ is determined from

$$\phi = \frac{360°}{\text{Horizontal Resolution}}. \tag{1}$$

The value of θ varies based on which two beams are being considered and is derived from the angular position values given in Table 2, Chapter II. The values of W and L are then determined using the two equations below

$$\mathrm{L} = 2\mathrm{R}\sin\left(\frac{\theta}{2}\right) \tag{2}$$

and

$$\mathrm{W} = 2\mathrm{R}\sin\left(\frac{\phi}{2}\right) \tag{3}$$

respectively.

The following two equations, taken from [32], were used to find the variation of W and L as

$$\Delta\mathrm{L} = \left|\frac{\partial f}{\partial \mathrm{R}}\right|\Delta\mathrm{R} + \left|\frac{\partial f}{\partial \theta}\right|\Delta\theta \tag{4}$$

and

$$\Delta\mathrm{W} = \left|\frac{\partial f}{\partial \mathrm{R}}\right|\Delta\mathrm{R} + \left|\frac{\partial f}{\partial \phi}\right|\Delta\phi \tag{5}$$

based on the published sensor data from Table 1, Chapter II, repeated here in Table 3 for convenience. The partial derivatives with respect to R, θ, and ϕ are given by

$$\left|\frac{\partial f}{\partial \mathrm{R}}\right| = \left|2\sin\left(\frac{\theta}{2}\right)\right|, \tag{6}$$

$$\left|\frac{\partial f}{\partial \theta}\right| = \left|\mathrm{R}\cos\left(\frac{\theta}{2}\right)\right|, \tag{7}$$

and

$$\left|\frac{\partial f}{\partial \phi}\right| = \left|\mathrm{R}\cos\left(\frac{\phi}{2}\right)\right|. \tag{8}$$

Table 3. Sensor Parameters for the Calculation of Width and Length Variance

Parameter	Value
ΔR	0.05 m
$\Delta\,\phi$	0.01°
$\Delta\theta$	0.01°

The values of W and L with corresponding deviations were calculated for each pair of beams, with R varied from 0.8 m to 10 m in 0.1 m increments using the MATLAB script given in Appendix A. A sample of the resulting data table is given in Table 4. Based on this data, the ideal smallest detectable object that the sensor is capable of placing four points on from two beams has the dimensions of W = 0.0049 m and L = 0.030 m when R = 0.8 m and W = 0.061 m and L =0.380 m when R = 10.0 m. These dimensions represent the smallest detectable object under ideal conditions where an object is centered between two beams. With the sensor mounted on a moving platform, objects can be brought in and out of this optimum detection location.

Table 4. Width and Length Calculations for Beams 15 and 16

Distance R (m)	Vertical Distance (cm)	Vertical Error in cm (+-)	Horizontal Distance (cm)	Horizontal Error in cm (+-)
0.8	3.043670616	0.204189521	0.490873082	0.044642136
0.9	3.424129443	0.205934534	0.552232217	0.046387457
1.0	3.804588269	0.207679548	0.613591353	0.048132778
1.1	4.185047096	0.209424561	0.674950488	0.049878099
1.2	4.565505923	0.211169575	0.736309623	0.05162342
1.3	4.94596475	0.212914588	0.797668758	0.053368741
1.4	5.326423577	0.214659601	0.859027894	0.055114062
1.5	5.706882404	0.216404615	0.920387029	0.056859383
1.6	6.087341231	0.218149628	0.981746164	0.058604704
1.7	6.467800058	0.219894642	1.043105299	0.060350025
1.8	6.848258885	0.221639655	1.104464435	0.062095346

Distance R (m)	Vertical Distance (cm)	Vertical Error in cm (+-)	Horizontal Distance (cm)	Horizontal Error in cm (+-)
1.9	7.228717712	0.223384669	1.16582357	0.063840667
2.0	7.609176539	0.225129682	1.227182705	0.065585988
2.1	7.989635366	0.226874696	1.28854184	0.067331309
2.2	8.370094193	0.228619709	1.349900976	0.06907663
2.3	8.75055302	0.230364722	1.411260111	0.070821952
2.4	9.131011847	0.232109736	1.472619246	0.072567273
2.5	9.511470674	0.233854749	1.533978381	0.074312594
2.6	9.891929501	0.235599763	1.595337517	0.076057915
2.7	10.27238833	0.237344776	1.656696652	0.077803236
2.8	10.65284715	0.23908979	1.718055787	0.079548557
2.9	11.03330598	0.240834803	1.779414923	0.081293878
3.0	11.41376481	0.242579816	1.840774058	0.083039199

C. SENSOR CHARACTERISTICS WHEN MOUNTED ON THE ROBOT

After fully exploring the characteristics of the OS1-16, the next step was to analyze the effect on sensor output due to the physical location of the sensor on the robot. Based on the previous analysis of the sensor and the orientation of the beams, the sensor has the best chance of detecting a small object on the ground plane the closer the sensor itself is to the ground plane. For this reason, the sensor was mounted as close to the ground plane as possible without hindering the 360° field of view of the sensor.

A graphical representation of the sensor mounted on the robot and the associated angular orientation between the beams and the ground plane is given in Figure 15. The distance from the sensor to the point where each individual beam strikes the ground plane is known as the ground strike distance. Each beam has a different ground strike distance that is determined by the physical orientation of the sensor and the angular orientation of the beam. In an ideal configuration, the XY plane of the sensor would be parallel to the ground plane as shown in Figure 16. Because of the uneven nature of the robot wheels, the XY plane of the sensor is offset from the ground plane by approximately 1.54°. Therefore, the value of θ, as shown in Figure 15, must be adjusted to account for the 1.54° offset. The beam ground strike derived from Figure 15 is given by

$$\text{Beam Ground Strike} = \text{H}\tan\left(\theta\right) = \text{H}\tan\left(90-\beta\right). \tag{9}$$

In this case, β is the published angle of the beam from the XY plane of the sensor as given in Table 2, Chapter II. The real-world beam ground strike distances for beams 16–10 are given in Table 5. Under ideal conditions, beam 9 should also strike the ground plane but because of the 1.54° offset, the elevation of the beam is above the XY plane of the sensor and is therefore projected upwards in the positive Z direction.

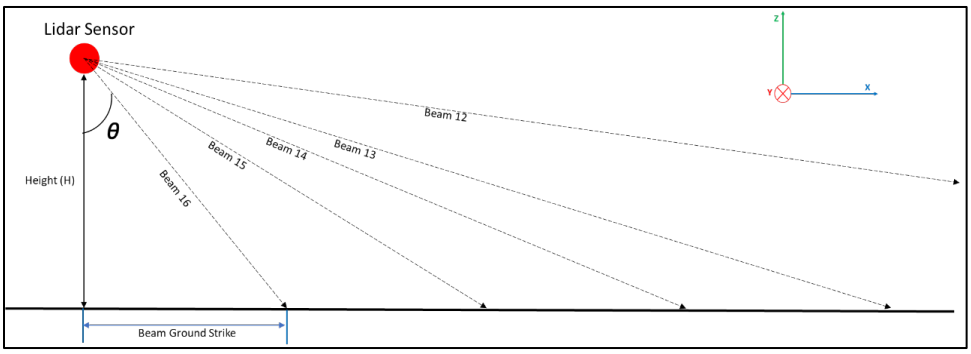


Figure 15. Beam Orientation with Sensor on Robot

Table 5. Beam Ground Strike Location

Beam	Ground Strike Distance (m)
10	10.031
11	4.6223
12	2.9979
13	2.2086
14	1.7441
15	1.4354
16	1.2122

The initial hypothesis drawn from the data in Table 5 was that given a single point cloud, an object cannot be seen by the sensor unless the object lay in the direct path of one of the beams. Any object with a small enough height in the Z direction that lies in the dead space between two beams has the potential to be missed by the sensor. This concept is illustrated in Figure 15 as the space between points where two beams strike the ground plane. Three test distances were determined from which the minimum object dimensions

at those distances could be identified. The three test distances determined were 0.8 m, 1.5 m, and 3.0 m.

The theoretical location of beams 16 to 9 in the Z direction at each test distance as shown in Figure 16 was calculated using

$$h = \frac{BGS - L}{\tan(\theta)}. \tag{10}$$

The theoretical results of this analysis are presented in Table 6.

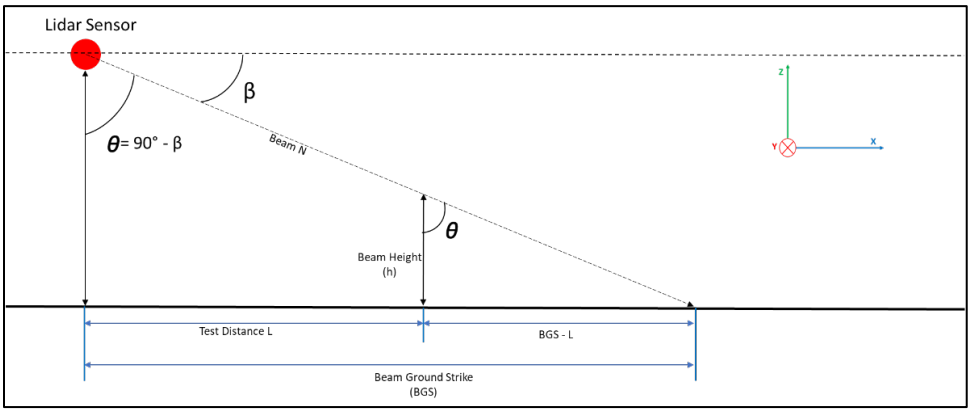


Figure 16. Beam Height Representation

Table 6. Beam Vertical Locations

Beam	Vertical location of Beam (m) when L = 0.8 m	Vertical location of Beam (m) when L = 1.5 m	Vertical location of Beam (m) when L = 3.0 m
9	0.317688828	0.321354053	0.329208107
10	0.288498751	0.266622657	0.219745315
11	0.259242015	0.211766279	0.110032557
12	0.229840095	0.156637678	-0.000224644*
13	0.199942278	0.100579272	-0.112341456
14	0.169701423	0.043877668	-0.225744665
15	0.138779241	-0.014101423	-0.341702846
16	0.106605933	-0.074426377	-0.462352753

*Negative values indicate that the beam has hit the ground plane before the distance L

As indicated in Table 6, beam 16, the lowest looking beam, will theoretically only be able to detect an object approximately 10 cm in height that is 0.8 m from the sensor. At the 1.5 m distance, beam 14 will theoretically be able to see an object that is approximately 4 cm or greater in height.

D. VERIFICATION OF THEORETICAL RESULTS

To determine the validity of the theoretical calculations, an experiment was devised to either confirm or deny the theoretical results as well as define the dimensions of the smallest identifiable object. The robot with the sensor was positioned on a flat, homogenous surface, as shown in Figure 17. A flat, square box large enough to provide multiple point returns from multiple beams was positioned at varying distances from the sensor. The resulting point cloud of this box was then analyzed and the two beams which provided returns on the box and were closest to the ground plane where isolated as shown by the point cloud in Figure 18.

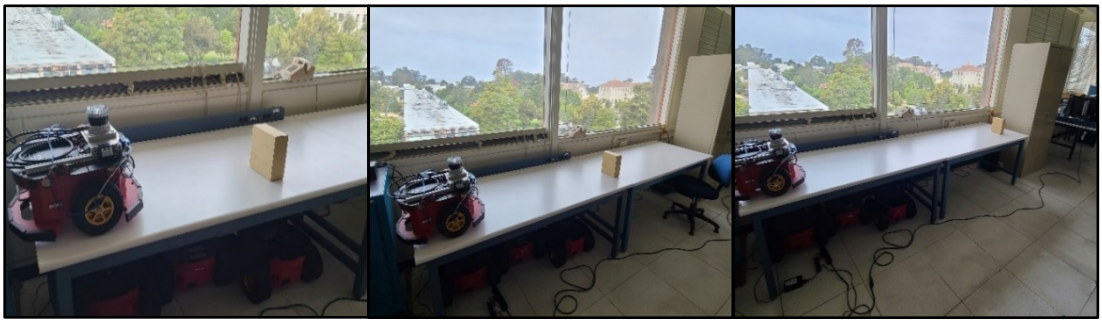


Figure 17. Small Object Test Setup at 0.8 m, 1.5 m, and 3.0 m

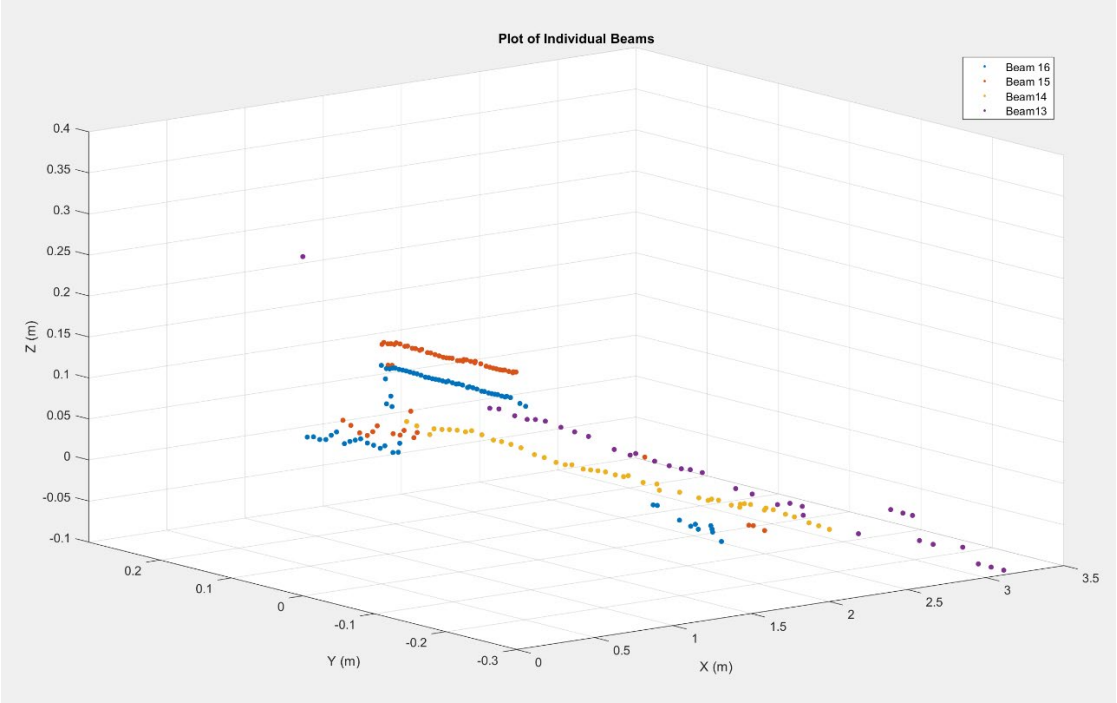


Figure 18. Point Cloud of Target Box at 0.8 m

While keeping with the definition of detection, four points, two from two separate beams, were chosen to be analyzed. For consistency, the two beams closest to the ground plane were chosen and the four points either side of the Z-axis were analyzed, as shown in Figure 19. One hundred point clouds were taken for each of the three test distances and the mean and standard deviation of the dimensions between these four points determined. The probability density functions (PDFs) of the dimensions at the corresponding test distances are given by Figure 20, Figure 21, and Figure 23, respectively. The numerical values for the mean and standard deviation along with a confidence interval for each test are given in Table 7.

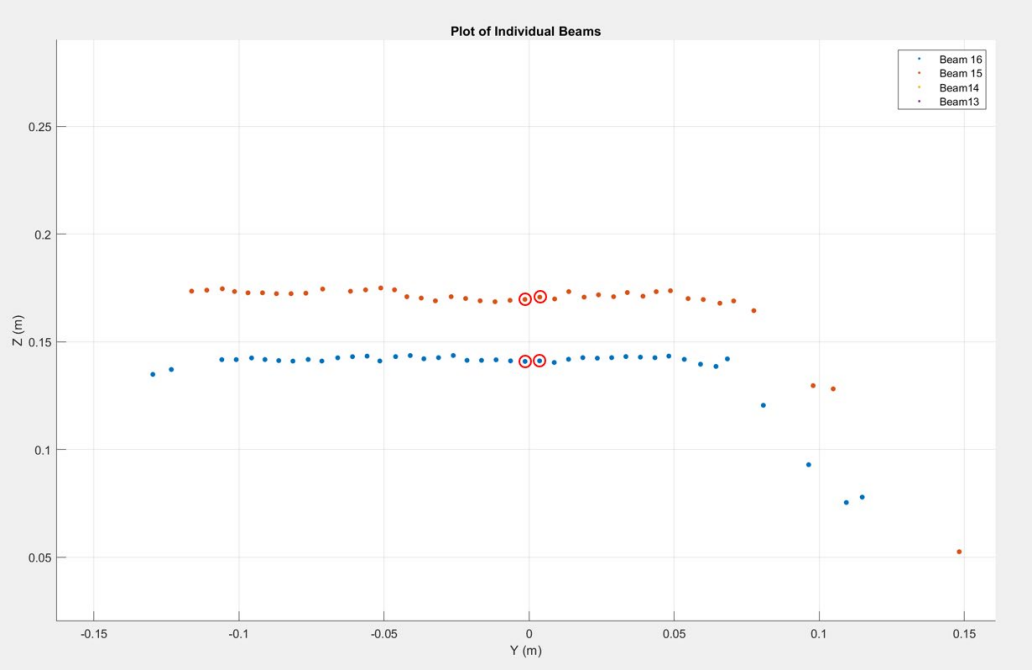


Figure 19. Test Points

1. Test Distance: 0.8 meters

In the case of the test distance of 0.8 m, the two beams closest to the ground plane that return points on the box are beams 15 and 16 as can be seen in Figure 18. From beams 15 and 16, the four points indicated on Figure 19 are chosen for analysis. These same four points are analyzed across one hundred different point clouds with the box at a distance of 0.8 m from the sensor. The PDFs of the dimensions between the four test points with corresponding mean and standard deviation are given in Figure 20.

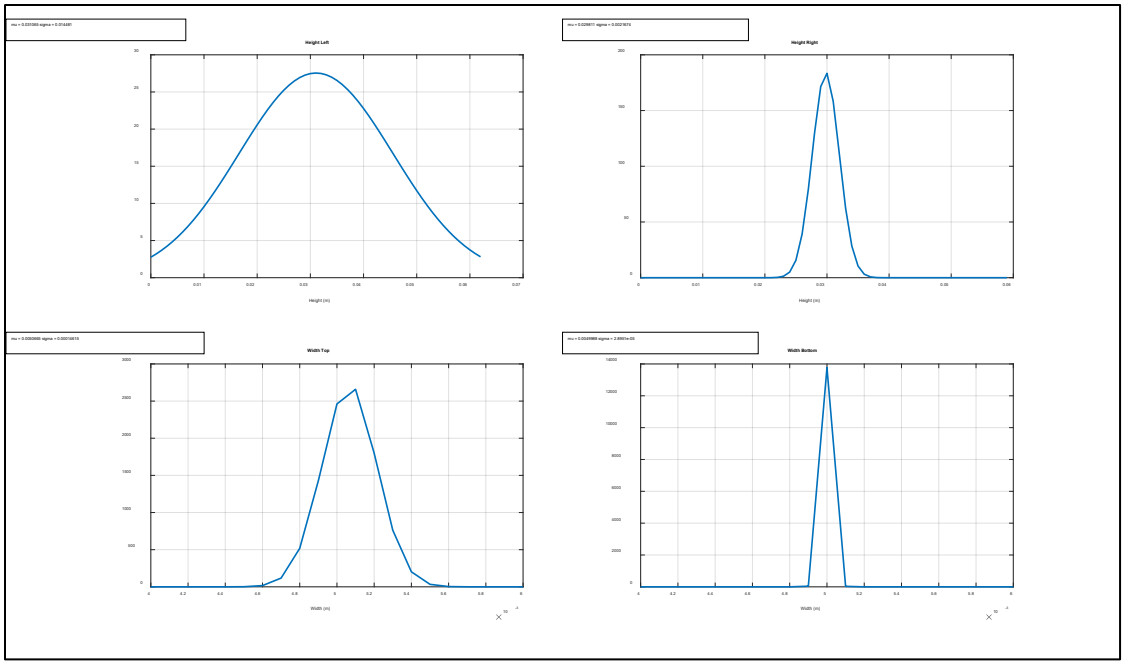


Figure 20. PDFs at Test Distance of 0.8 m

2. Test Distance: 1.5 meters

In the case of the 1.5 m test distance, there is an interesting phenomenon due to the scattering of beam 16 as the laser strikes the table. According to the data in Table 5, beam 16 will strike the ground approximately 0.28 m before the object. As the beam strikes the ground plane, it is reflected off the ground plane and goes on to strike the box. The sensor receives a greater energy return from the reflection off the box than from the table top and therefore records the reflection off the box as a point [33]. This is also why the Z coordinate for the points on the box are negative or below the table [34]. This same phenomenon is also seen at the 3.0 m test distance with beam 13. However, as can be seen in Figure 22, beam 16 still returns relatively accurate points at the base of the object for the test distance of 1.5m. The PDFs of the dimensions between the four test points with corresponding mean and standard deviation are shown in Figure 21.

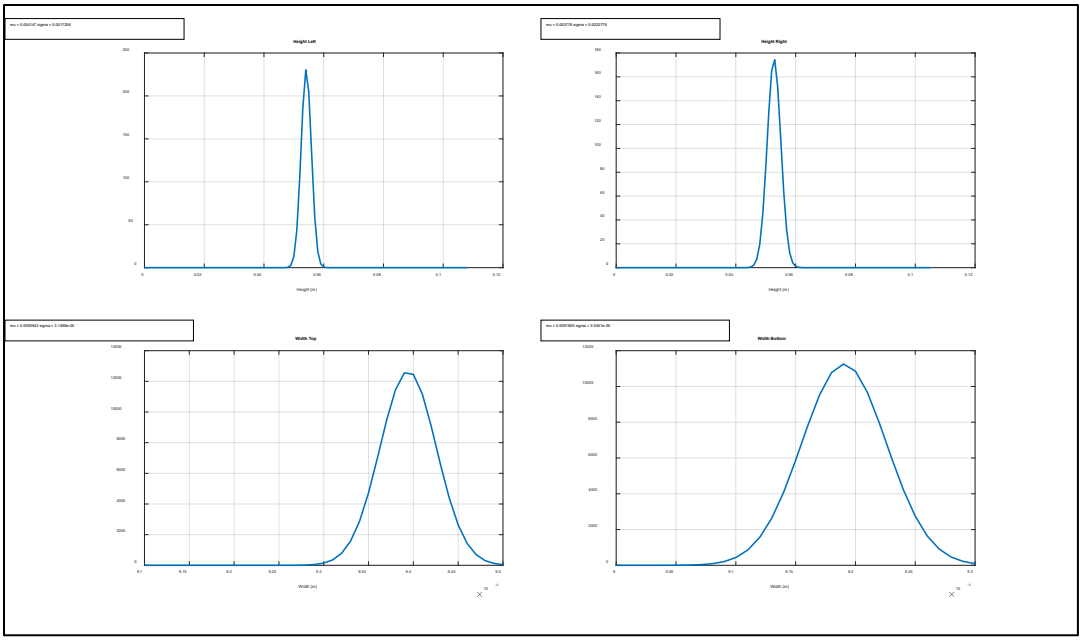


Figure 21. PDFs at Test Distance of 1.5 m

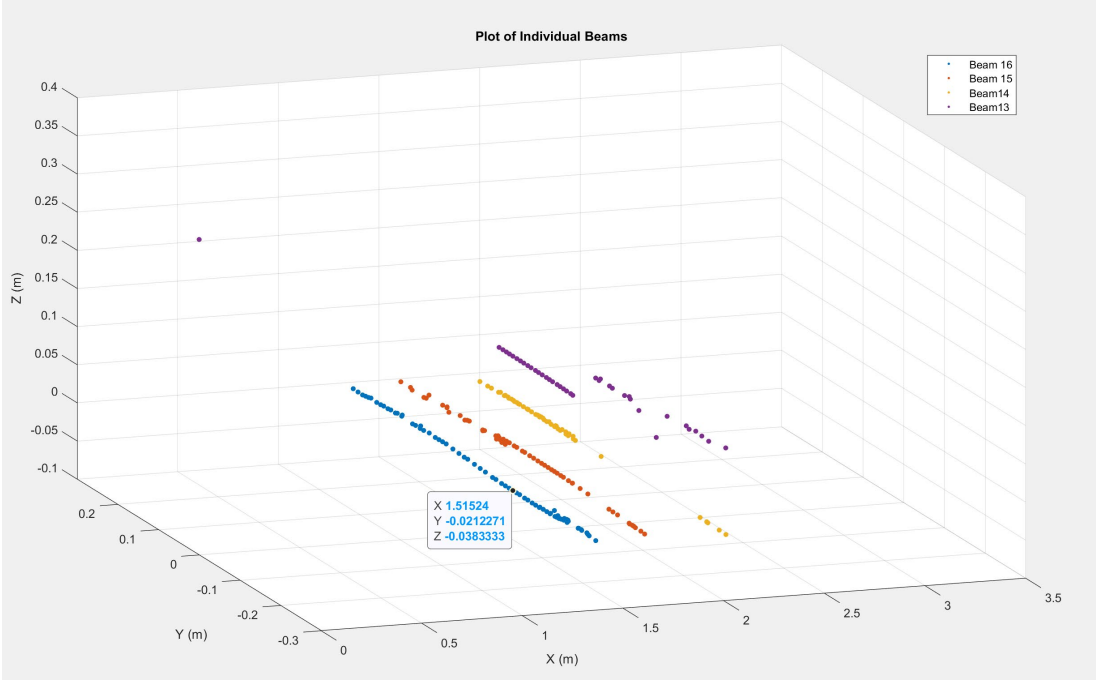


Figure 22. Point Cloud of Target Box at 1.5 m

3. Test Distance: 3.0 meters

As mentioned before, beam 13 shows scattered reflections at the base of the box. For the sake of analysis, these points were determined to be not sufficiently reliable and therefore the two closest beams to the ground plane were determined to be beams 12 and 11. The PDFs of the dimensions between the four test points with corresponding mean and standard deviation are given in Figure 23.

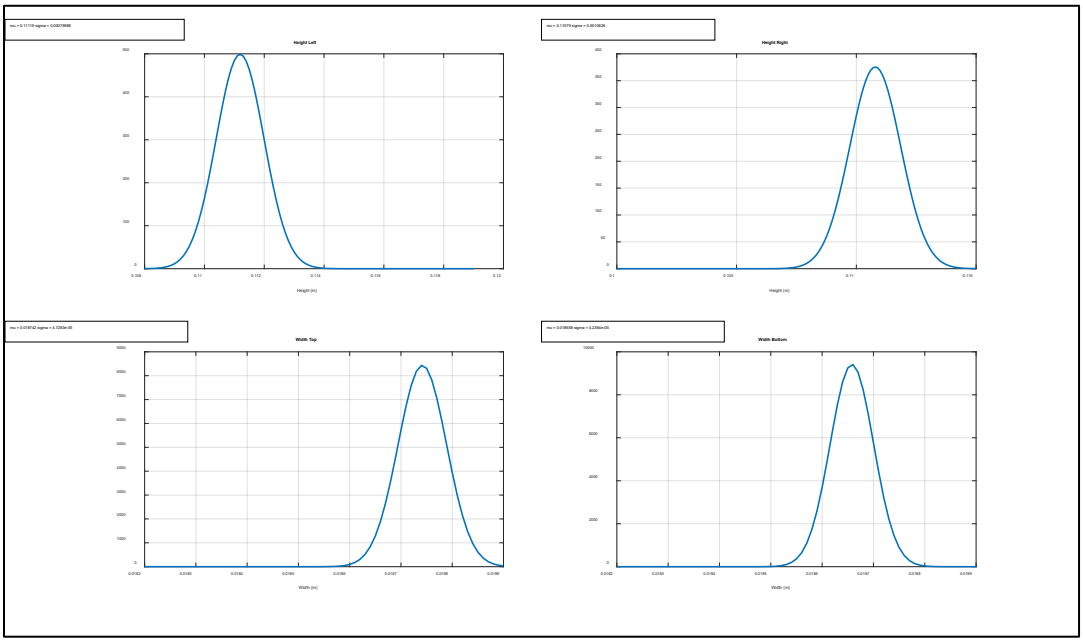


Figure 23. PDFs at Test Distance of 3.0 m

Table 7. Results of Minimum Object Dimensions Experiment

Test Distance: 0.8 m				
Dimension	Mean (μ) (m)	Confidence Interval (m)	Standard Deviation (m)	Confidence Interval (m)
H_L	0.0310651	0.0281916, 0.0339385	0.0144815	0.0127148, 0.0168227
H_R	0.0298111	0.029381, 0.0302412	0.00216737	0.00190297, 0.00251779
W_T	0.00506646	0.00503746, 0.00509546	0.000146154	0.000128324, 0.000169783
W_B	0.00499876	0.00499301, 0.0050045	2.89513e-05	2.54194e-05, 3.3632e-05

Test Distance: 0.8 m				
Dimension	Mean (μ) (m)	Confidence Interval (m)	Standard Deviation (m)	Confidence Interval (m)
Z_{min}	0.141624	0.141441, 0.141806	0.0013101	0.00119306, 0.0014528
Test Distance: 1.5 m				
Dimension	Mean (μ) (m)	Confidence Interval (m)	Standard Deviation (m)	Confidence Interval (m)
H_L	0.0541472	0.0538048, 0.0544896	0.00172561	0.0015151, 0.0020046
H_R	0.0527756	0.0523236, 0.0532276	0.00227794	0.00200005, 0.00264623
W_T	0.00939422	0.00938797, 0.00940047	3.14957e-05	2.76535e-05, 3.65878e-05
W_B	0.00919045	0.00918342, 0.00919748	3.54506e-05	3.11259e-05, 4.11821e-05
Z_{min}	-0.036528	-0.0367579, -0.0362981	0.00164874	0.00150144, 0.00182833
Test Distance: 3.0 m				
Dimension	Mean (μ) (m)	Confidence Interval (m)	Standard Deviation (m)	Confidence Interval (m)
H_L	0.111193	0.111035, 0.111352	0.000799883	0.000702303, 0.000929204
H_R	0.11079	0.110579, 0.111001	0.0010626	0.000932972, 0.0012344
W_T	0.0187417	0.0187323, 0.0187511	4.72829e-05	4.15147e-05, 5.49274e-05
W_B	0.018658	0.0186496, 0.0186664	4.23638e-05	3.71957e-05, 4.9213e-05
Z_{min}	0.0327893	0.0326841, 0.0328945	0.000754361	0.000686968, 0.00083653

By comparing the values in Table 7 with the corresponding values in Table 4, the measured results closely match the theoretical results. Furthermore, from the experimental data, we can draw conclusions about the dimensions and characteristics of the smallest detectable object to the sensor when mounted on the robot. The approximation for the smallest identifiable obstacle for each test distance is given in Table 8. The dimensions given are the height and width of the object as seen by the sensor in a 2D plane. The Z_{min} threshold

requirement from Table 6 was also considered. The Z_{min} threshold must first be met before the lowest beam of the sensor can be reasonably expected to strike the object.

Table 8. Smallest Detectable Object

Distance from Sensor (L)	Minimum dimensions of object (m) H x W
0.8 m	0.187 x 0.005
1.5 m	0.055 x 0.009
3.0 m	0.111 x 0.018

The lowest beam to the ground plane, beam 16, strikes the ground at approximately 1.2 m. The 1.2 m distance can be taken to be the optimum distance from which the sensor can detect a small object. Based on this assumption, the dimensions of an object at the test distance of 1.5 m most accurately predict the smallest object at the ground plane that the sensor will be able to identify.

This data is taken from a single point cloud snapshot. The data does not take into consideration that the sensor is mounted on the robot which is capable of both translation and rotation in the XY plane. As the robot moves and the beams are swept across the ground plane, any object in the path of the robot will at one time cross the 1.2 m threshold and thus be detected if the object is at least the dimensions given in Table 8. This concept is explored briefly and elaborated on in Chapters V and VI.

IV. NEGATIVE HEIGHT OBSTACLE DETECTION

A. PREVIOUS WORK

Multiple studies have been conducted concerning the identification and classification of what are commonly referred to as negative obstacles [14] or obstacles with negative height that are below the established ground plane. The study conducted in [35] makes use of cameras and stereo vision to detect negative obstacles in an indoor urban environment similar to the one used in this study, specifically the identification of drop-offs. In [36], multiple rules are presented for the establishment of terrain traverse ability including the analysis of terrain slope. The concept of slope is expanded with the use of a lidar sensor in [7], [14], and [37] to detect negative obstacles. The method used in this work is similar to the slope analysis method.

B. CONCEPT OF DETECTION

The method of detection for negative obstacles explored in this work is very similar to the method presented in [38]. Instead of the pixels used in [38], this work used the point returns from a single beam of the OS1-16 sensor. Similar to [38], a threshold value of the sensor was established and then monitored to detect any deviations. The threshold value used was the range values of a set of points from a single beam. The idea to use a single beam of the lidar sensor was adapted from the hardware setup used by Miyakawa in [7].

C. PROCESS

The first step involved the isolation of a single beam, in this case the beam closest to the ground plane, beam 16, as described in Chapter III. Beam 16 was chosen because the ground strike distance of beam 16 is the smallest of all the lidar beams. The smaller ground strike distance reduces the chance of other, non-negative obstacles from interfering with the calibration of the beam as discussed later. The drawback of using the beam with the smallest ground strike distance was that the robot has less time to react and avoid a detected negative obstacle. This is a significant issue for autonomous platforms traveling at higher speeds as indicated in [37], [38]. The translation speed of the robot for this work

was slow enough to allow for sufficient reaction time to any negative obstacles detected by beam 16. As discussed in Chapter V, the control algorithm was adjusted to accommodate for the smaller reaction time.

The use of different beams with longer ground strike distances, specifically beam 15, was explored. The use of beam 15 to identify negative obstacles would have allowed for greater reaction time but the chance of interference due to non-negative obstacles is greater. Therefore, in order to focus on the proof of concept for the identification of negative obstacles using the discussed method, only beam 16 was used.

The method of detection used in this work ultimately relies on detecting a change in the ground plane in the Z direction similar to the method used in [38]. The variable used to detect a negative obstacle is the distance from the lidar sensor to the ground plane R, as shown in Figure 24. The threshold value, R_{TH} is determined when the robot is initialized by computing the average R value of all 1,024 point returns of beam 16. This process only works if the robot is initialized on the intended ground plane of operation with the area around the robot out to a distance of 1.3 m clear of all obstacles negative and non-negative. This concept is depicted graphically in Figure 25.

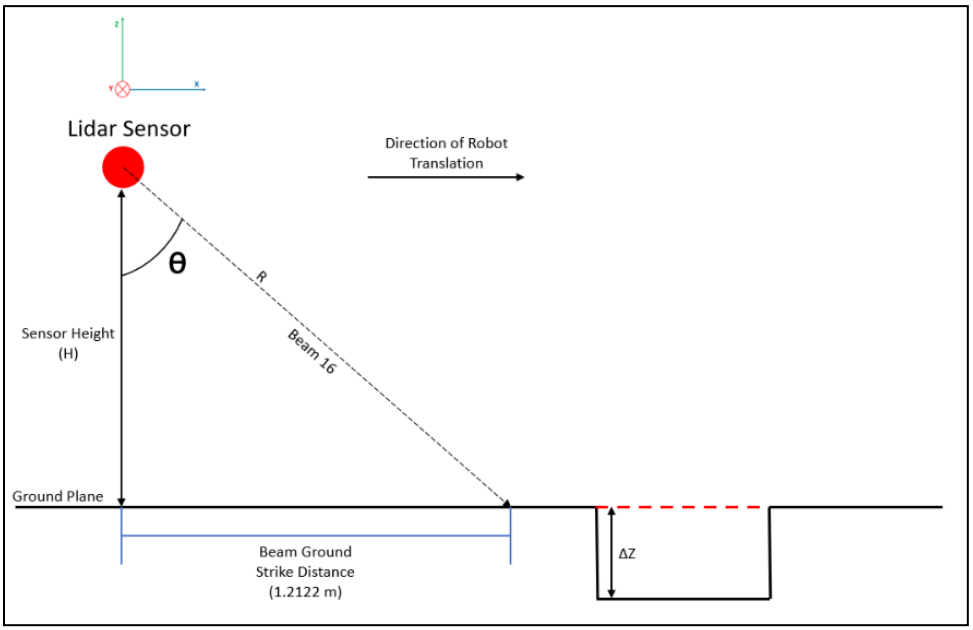


Figure 24. Beam 16 Threshold at Ground Plane

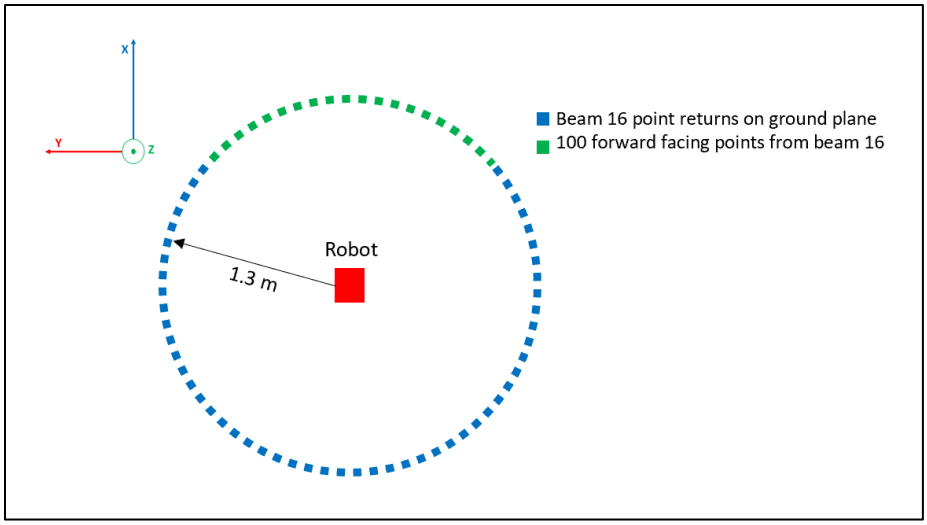


Figure 25. Robot Initialization Parameters

The presence of a negative obstacle will result in a value for R greater than R_{TH}, as shown in Figure 26. The greater value of R translates to a change in the elevation of the ground plane in the negative Z direction.

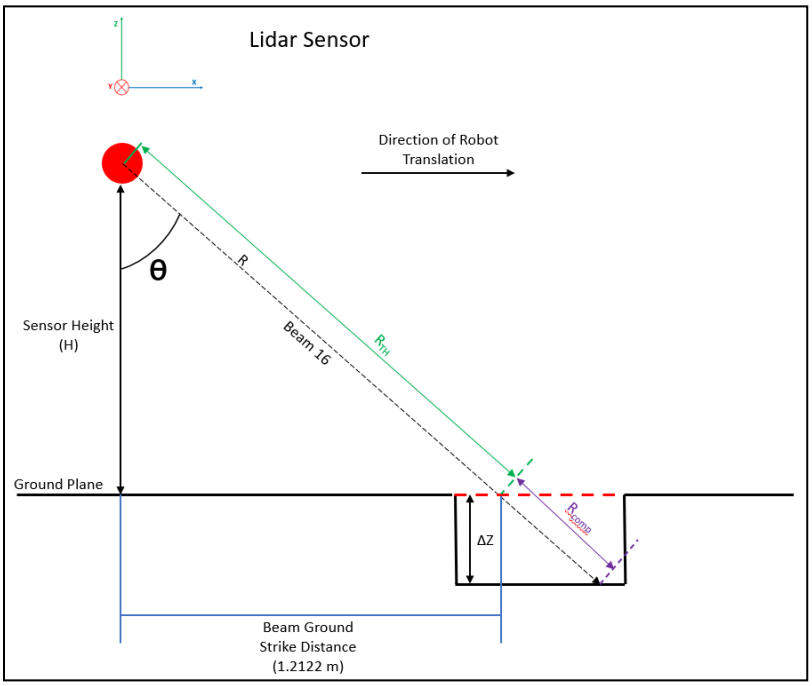


Figure 26. Detection of a Negative Obstacle

D. NEGATIVE OBSTACLE DETECTION ALGORITHM

The first step of the negative obstacle detection algorithm is to determine the value for R_{TH} as previously described. The second step is to isolate the 100 forward facing points from beam 16, as depicted in green in Figure 25. This number was determined to allow adequate detection of negative obstacles to the left and right of the robot. Increasing this number will increase the field of view of the robot for detecting negative obstacles but may constrict the movement of the robot in a tight, indoor environment. As the robot translates through the environment, the algorithm monitors the R value of each of the 100 forward facing points. Specifically, the algorithm compares the difference between the R value of each individual point and R_{TH} with a pre-designated comparison value $R_{comp,}$ as depicted in Figure 26. R_{comp} can be adjusted to designate the allowable ΔZ value traversable by the robot. For this work, the R_{comp} was conservatively set at 600 mm to both account for the limited capabilities of the robot to traverse vertical drops and ensure detection of negative obstacles for testing purposes.

As the robot translates in the forward X direction, each consecutive row of beam 16 point returns is analyzed and saved as a binary vector where a binary 1 represents a point on the ground plane and a binary 0 represents the presence of a negative obstacle. This concept is illustrated in a simplified fashion in Figure 27 to depict the presence of a hole. The algorithm then identifies the leading and trailing edges of the negative obstacle by identifying the transitions of 1 to 0 and 0 to 1 between the current scan and the previous scan or in the case of the example, within a column. In a similar fashion the algorithm determines the left and right edges of the obstacle by identifying the 10 and 01 transitions within each scan or row. These transition points, indicated by the red circles in Figure 27, are saved and used in the potential field model for navigation and obstacle avoidance as detailed in Chapter V.

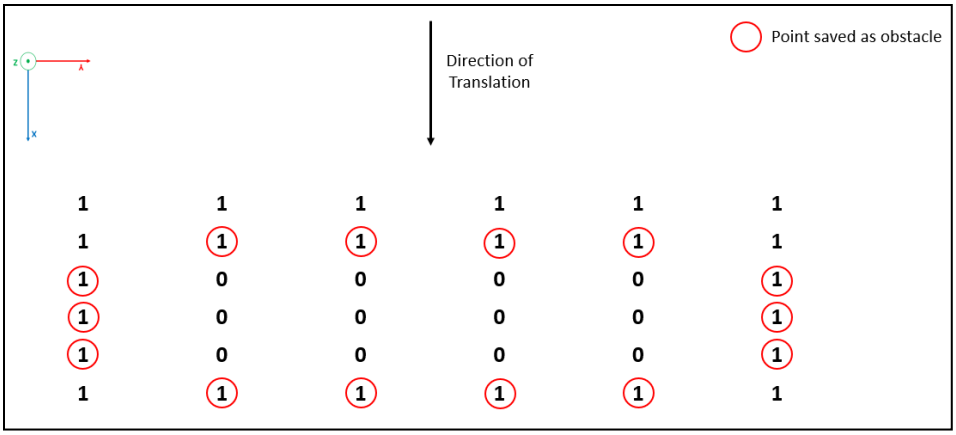


Figure 27. Example Detection of a Hole

E. EXPERIMENTAL SETUP AND RESULTS

To test the performance of the negative obstacle detection algorithm, an experiment was set up, as shown in Figure 28. The size of the circular hole was determined based on the diameter of the front wheels of the robot [37]. The kidney bean shaped hole was chosen to test the ability to detect a nontraditional geometric shape. The translation speed of the robot was limited to 20 mm per second and the robot moved along the table top towards the negative obstacles.

Figure 28. Negative Obstacle Detection Experimental Setup

The algorithm view of the negative obstacles is given in Figure 29. This image is the experimental equivalent of the example given in Figure 27 with the white space representing a logical 0 and the blue points representing a logical 1. The Y axis in Figure 29 is the scan number with the first scan representing the initial scan of beam 16 as the robot begins to move. As the robot moves the lidar continues to collect data at 10 Hz and each consecutive scan of beam 16 is taken from a new position as the robot translates towards the negative obstacles. As can be seen in Figure 29, the two holes can be clearly identified along with the edge of the cardboard.

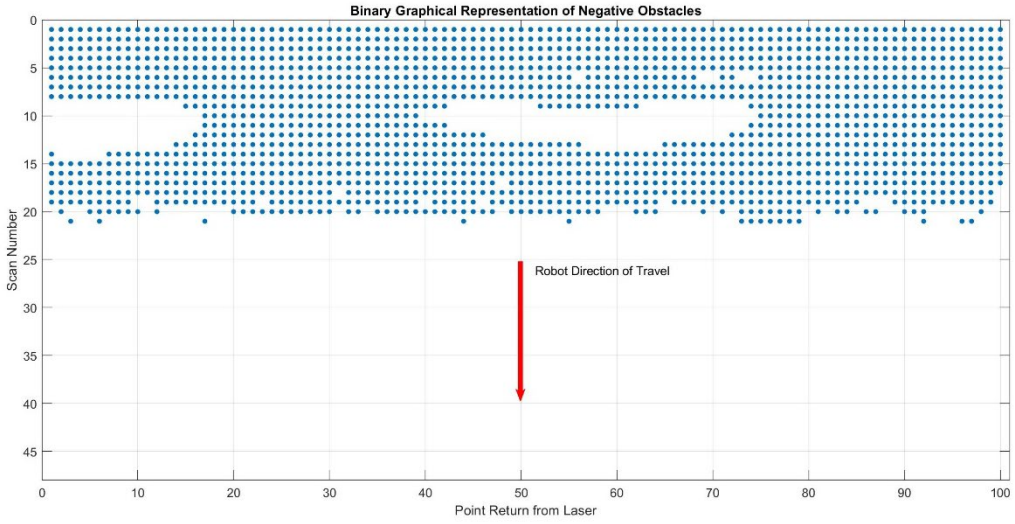


Figure 29. Binary Representation of Experimental Results

By using the binary interpretation, the algorithm then identifies the actual obstacle points, as indicated in Figure 30. In Figure 30, the table is defined as the ground plane and is located on the XY plane where Z is 0. The green point cloud in the right half of Figure 30 is the table the robot is moving on. As beam 16 passes the edge of the table, the beam begins to return points on the lab floor. The larger green point cloud in the left half of Figure 30 are the points representing the lab floor. The algorithm has identified the leading, trailing, left and right edges of both negative obstacles. The algorithm has also identified the edge of the cardboard as a drop off. The XYZ coordinates of the obstacle points can

then be fed into the potential field model for navigation and avoidance. This process is explored in detail in Chapter V.

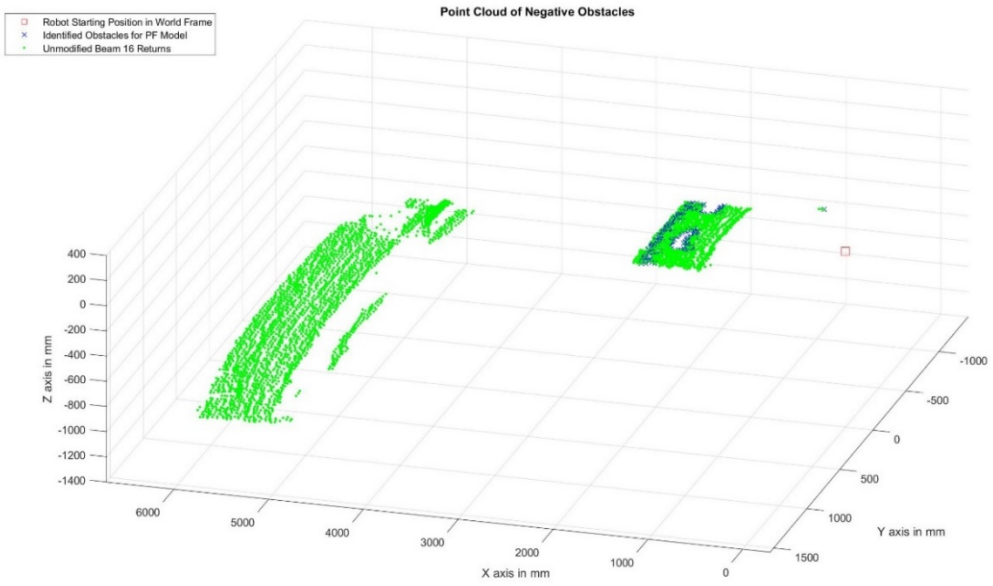


Figure 30. 3D Representation of Negative Obstacles

THIS PAGE INTENTIONALLY LEFT BLANK

V. SMALL OBJECT IDENTIFICATION

An algorithm that differentiates between small and large obstacles based on the definition for a small object detailed in Chapter III was developed. This algorithm was then integrated into a larger control algorithm that navigates the robot around both large and small obstacles. The basic premise of this method was to take the raw point cloud data from the OS1-16 sensor and filter the data to both reduce the number of points that will have to be computationally dealt with and categorize the various points as belonging to either a large or small obstacle.

A. POINT CLOUD FILTERING AND VOXELIZATION

Similar to the methods presented in [39] and [40], the conceptual end state of the point cloud filtering and voxelization algorithm was to reproduce the environment around the robot with occupied and unoccupied voxels. This environment is represented by a point cloud composed of the center points of occupied voxels with pre-defined dimensions as described by [40]. The objects throughout the environment are voxelized or represented by pre-defined three-dimensional cubes of equal size [41].

The process begins with the collection of the raw point cloud data from the OS1-16 lidar sensor. For the sake of reference and understanding, the picture presented in Figure 31 is the environment in which the data presented in this section was collected. The environment has both small and large objects with some of the key large objects highlighted in green and the small objects highlighted in red. From the environment presented in Figure 31, the lidar sensor returns the raw point cloud data resulting in the point cloud presented in Figure 32 with the red star representing the location of the sensor. The colors of the points in Figure 32 correspond to the intensity of the laser return at the particular point. The darker the color, the lower the intensity. This color map is used throughout this work when presenting point cloud data unless otherwise stated.

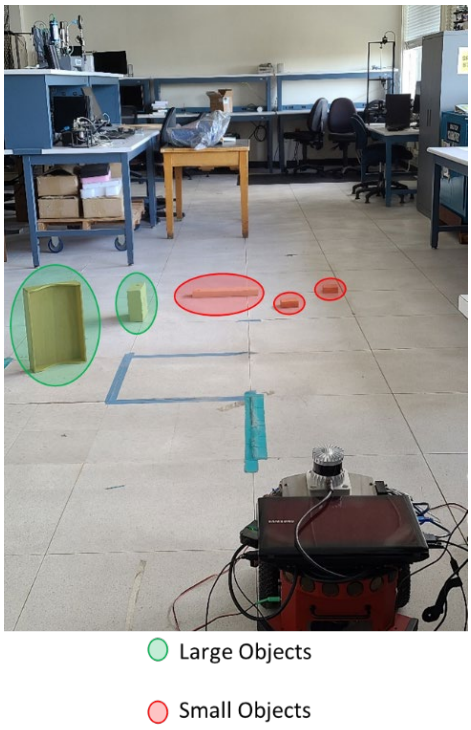


Figure 31. Point Cloud Filtering and Voxelization Sample Environment

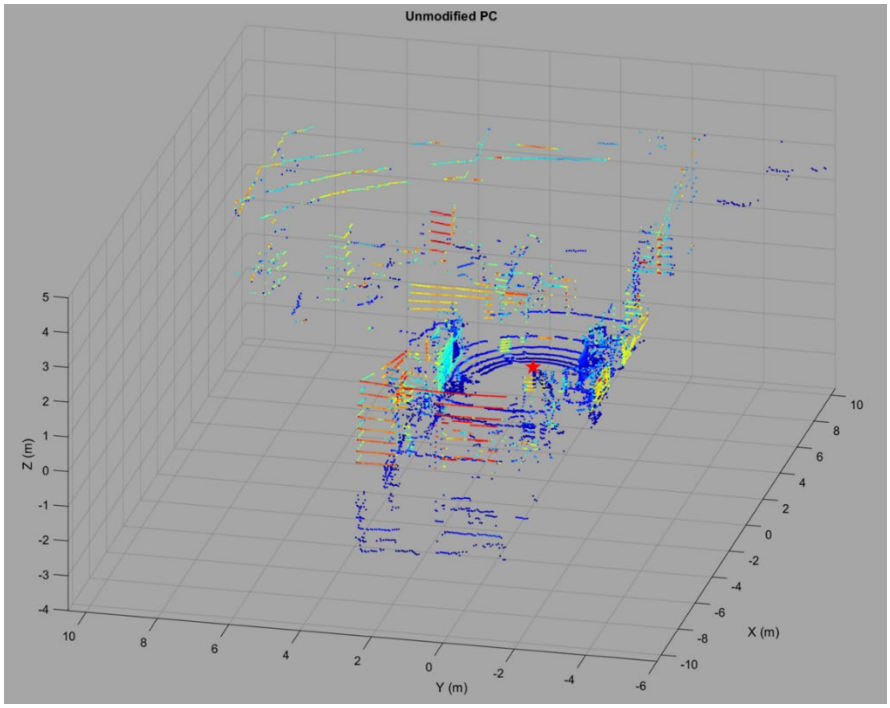


Figure 32. Raw OS1-16 Lidar Point Cloud Data

The first step in the filtering process is the removal of all invalid points that return near infinite distance data and points with coordinates at the origin. In the case of the OS1-16, when the sensor does not receive a return from a particular point, the sensor outputs the X, Y, and Z coordinates of (0, 0, 0). The next step in the filtering process is the removal of all points that lie outside the pre-defined region of interest (ROI). The ROI is a set of six coordinates that define the minimum and maximum limits for the X, Y, and Z values of any point. The exact values for the ROI used in this work can be found in the robot master control code in Appendix A and correspond to the field of view (FOV) of the robot as discussed later. The ROI of the point clouds presented in this section are much larger than the FOV of the robot to ensure clarity and understanding.

Aside from establishing the FOV of the robot, the primary purpose of the ROI is to segment and remove the ground plane from the rest of the point cloud data. The ground plane points can clearly be seen in Figure 32 as the concentric, circular rings expanding outwards from the origin. The ground plane points are also colored a dark blue, meaning the points have a low intensity value. This fact is exploited to ensure as much of the ground plane is extracted as possible. Due to the variation of the sensor, as discussed in Chapters II and III, defining the ROI alone is not enough to completely remove the ground plane from the point cloud data. The data is further filtered by removing all points with Z axis values and intensity values below a pre-selected elevation and intensity threshold. In the case of this work, the intensity threshold was determined by trial and error to be 220 and the Z value threshold was set to the vertical obstacle clearance capabilities of the robot, 60 mm. After removal of the ground plane, the resulting point cloud is presented in Figure 33 with the red star representing the approximate position of the sensor.

The concept of dead space discussed briefly in Chapter III is clearly illustrated when comparing the environment in Figure 31 with the corresponding point cloud in Figure 32. Close inspection of the point cloud presented in Figure 32 reveals that one of the smaller blocks seen in Figure 31 is missing. This is because the block falls between two of the lidar beams, or visually speaking, between two of the concentric rings that represent the ground plane in Figure 32.

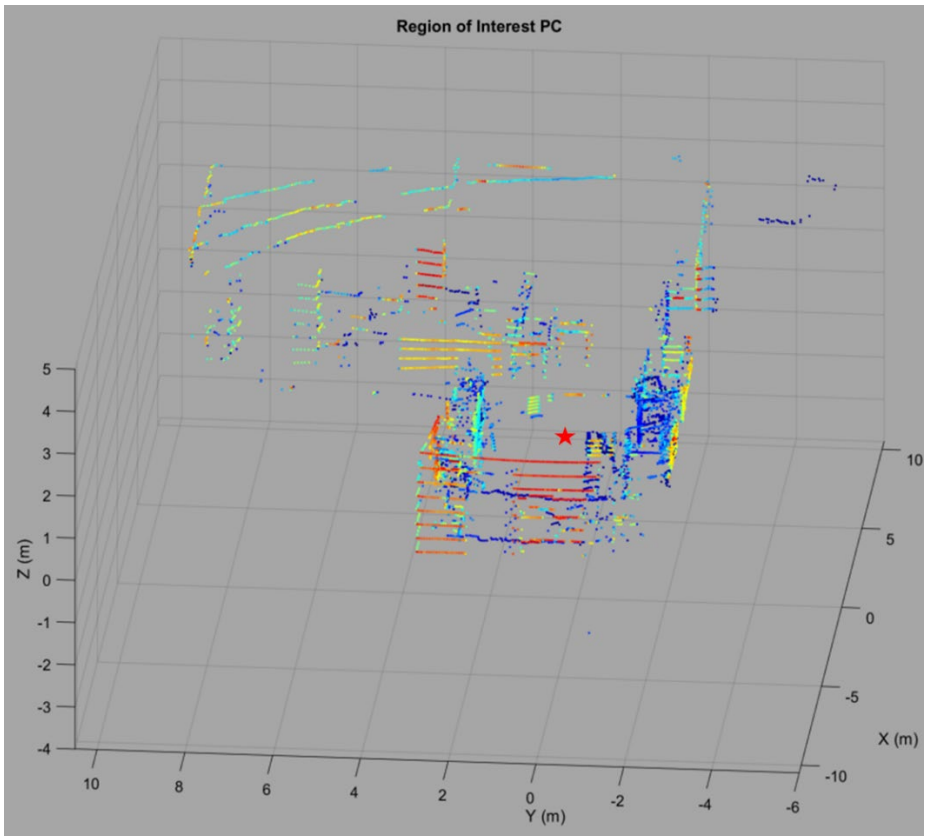


Figure 33. Point Cloud after Ground Plane Segmentation

The next step in the filtering process is the voxelization of the point cloud. This was achieved using the built-in MATLAB *pcdownsample* function described in detail in [42]. The *pcdownsample* function voxelizes the environment into cubic voxels of a predefined dimension and then returns a single X, Y, and Z coordinate for each voxel based on the average value of the coordinates of all the points that fall within each voxel [42]. The results of this function are presented in Figure 34 with the red star representing the position of the sensor.

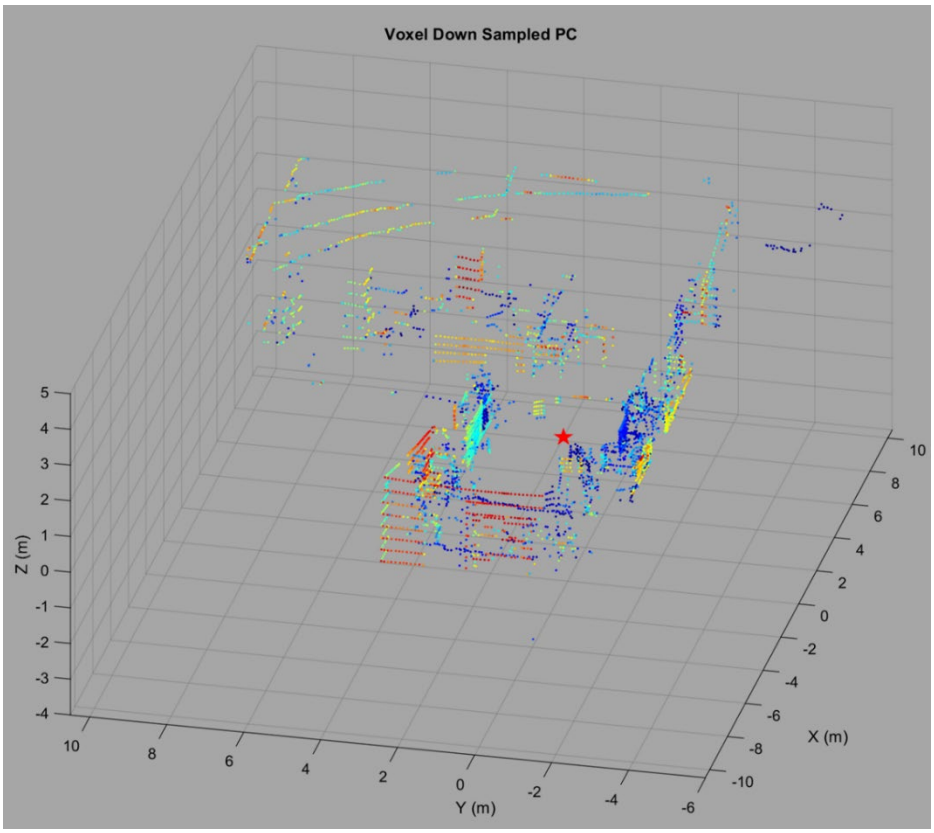


Figure 34. Output of MATLAB pcdownsample Function

The final step in the point cloud filtering and voxelization process is the replacement of the average points in each voxel with the center point of the occupied voxel. The code used to accomplish this method of point based voxelization [40] is given in Appendix A and the output point cloud is presented in Figure 35. The position of the sensor is presented by the red star in Figure 35.

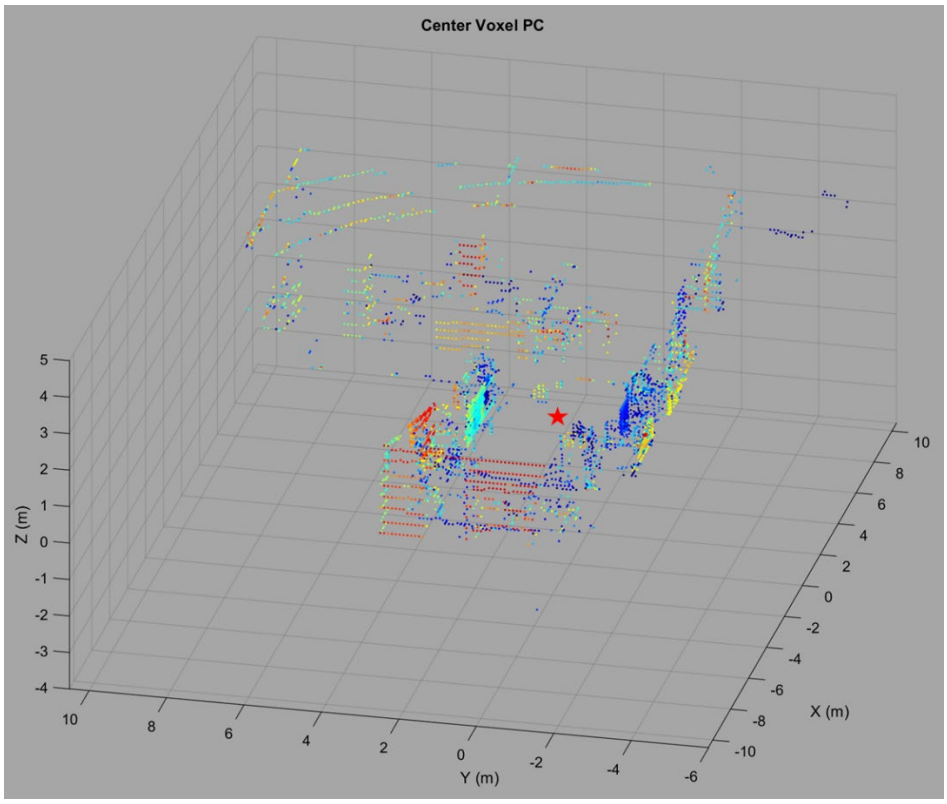


Figure 35. Result of Point Cloud Filtering and Voxelization

The filtering and voxelization reduces the number of points in the point cloud from 16,384 to 4,017. This number is reduced even further when the ROI is narrowed to reflect the FOV of the robot as discussed in the next section. A simplified example of the voxelization process is presented in Figure 36.

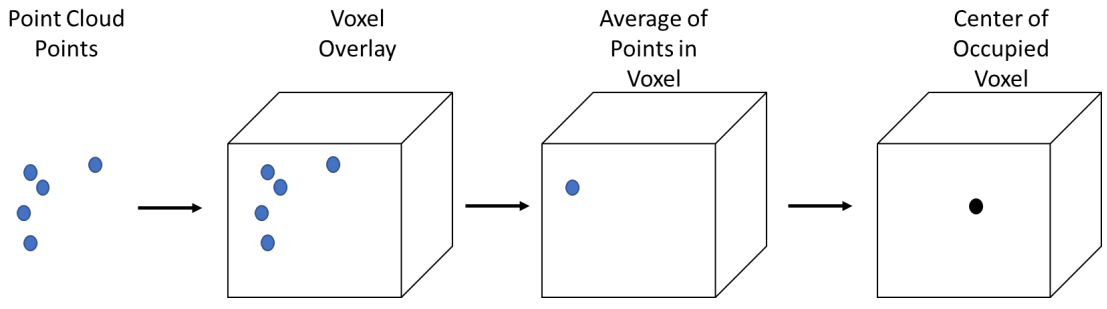


Figure 36. Voxelization Process Example

B. SMALL OBJECT CLASSIFICATION

With the point cloud filtered and voxelized, each point is then classified as either a small point or a large point. The first criteria to be considered is the height of the point in the world frame. If the point is greater than a pre-established threshold, the point is immediately classified as a large point. If the point falls below this threshold, there is the potential for the point to be a small point. The next step is to establish whether each point is part of a larger object. For example, a table leg will return a point cloud similar to the one shown in Figure 37. The bottom most point on the table leg will fall into the category of potential small objects because the point lies below the Z threshold. However, the point should be labeled as a large point because it is part of a larger object. The theory is that the lidar will be able to avoid this larger object using points on the object that are higher in elevation and thus more easily detected by the sensor as discussed in Chapter III. To classify the points in the potential small object category as either large or small, each point is analyzed in relation to the four nearest points to see if there is another point that lies directly above the point in question. This is possible because of the voxelization of the point cloud and the standardization of the voxel size or the distance L. This concept is illustrated with the lattice like depiction in Figure 38.

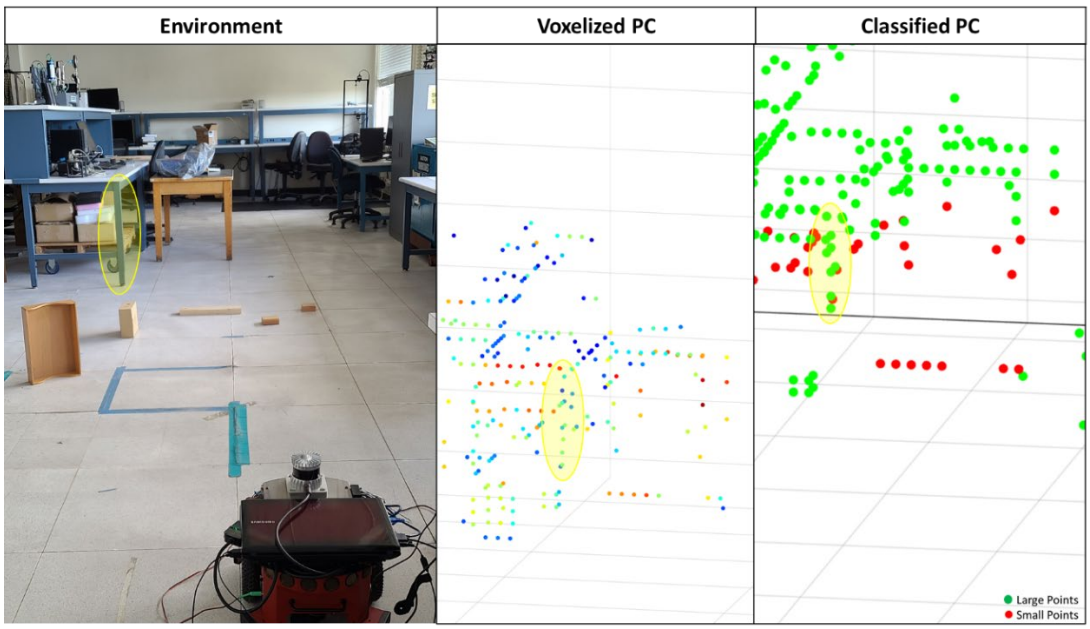


Figure 37. Table Leg Point Cloud Example

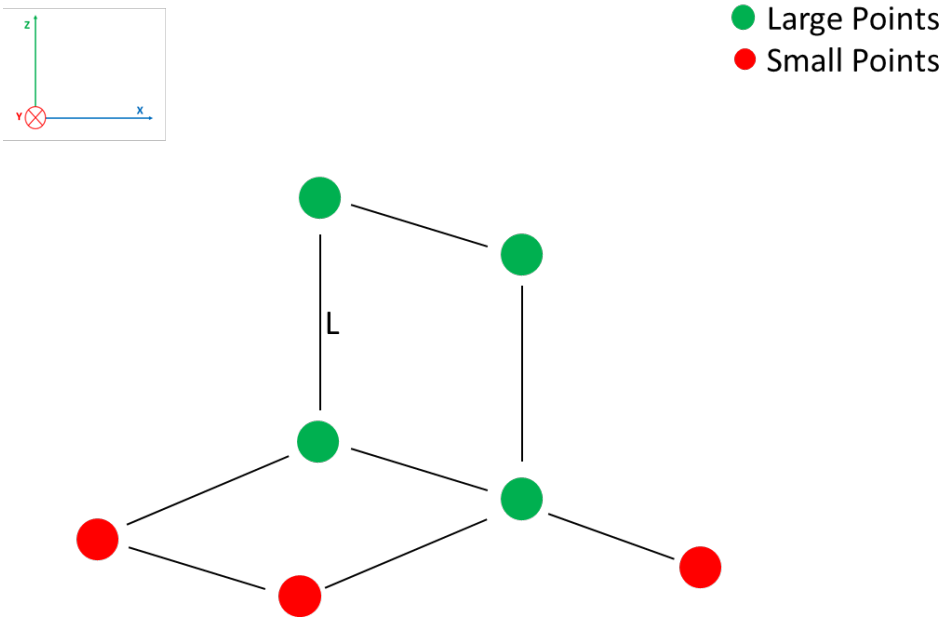


Figure 38. Concept of Small and Large Point Classification

The resulting point classification of the filtered and voxelized point cloud is given in Figure 39. As can be seen in Figure 39, the algorithm is not perfect and there are multiple misclassifications.

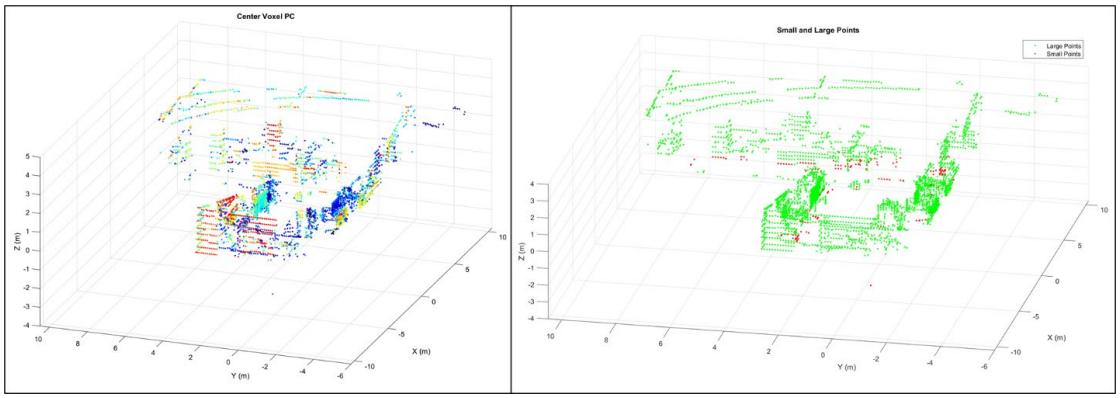


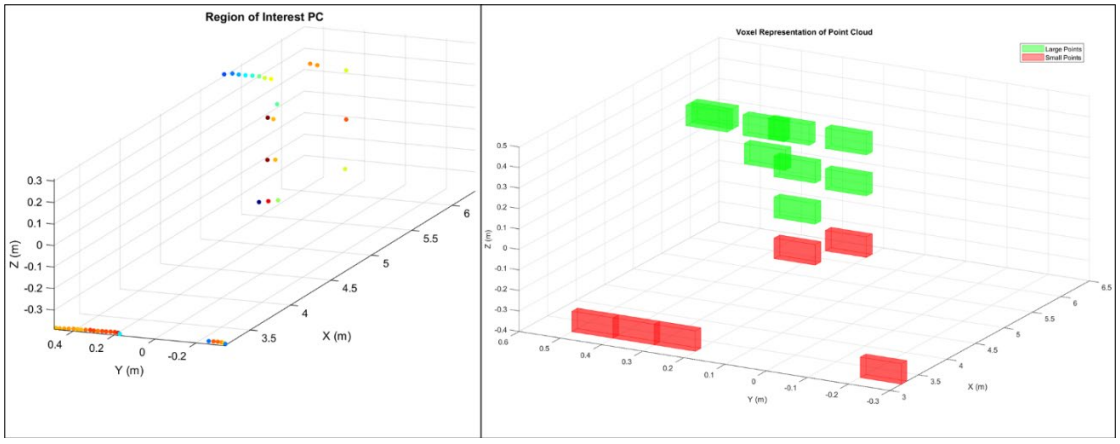
Figure 39. Voxelized and Classified Point Cloud Comparisons

Most of these misclassifications are at ranges greater than 4 m or due to irregularly shaped objects. The reason for the misclassification at range has to do with the size of the voxels used and the physical limitations of the sensor. For this work, the voxels used were cubic with length and width of 100 mm. The average distance between two beams is greater than 100 mm at distances greater than approximately 2.7 m as given in Chapter III, Table 4. Given the vertical error of the sensor, this means that at greater distances the likelihood of two vertically adjacent voxels containing a point decreases. This concept can be seen in Figure 40 where the bottom of a table leg is misclassified as a small point. This can be mitigated by adjusting the size of the voxels at the expense of a less detailed point cloud. Additionally, this issue is mitigated by the fact that as the robot translates towards an object, such as a table leg, there will be initial misclassifications that will eventually be corrected as the robot gets closer to the object. This phenomenon and the impact on obstacle avoidance are discussed in the experimental results section of Chapter VI.

C. FIELD OF VIEW

The point clouds presented in the previous section represented the complete data collection capability of the OS1-16 sensor. As stated, one of the objectives of this research was to minimize the computing requirements when dealing with point cloud data. For this reason, the FOV of the robot was limited to the forward facing area extending 500 mm to the left and right of the robot and approximately 100 mm above the height of the robot. The FOV of the robot when looking at the environment presented in Figure 31 is given in

Figure 40 along with the voxelized, small and large point FOV. The voxels in Figure 40 are for graphical representation only.



The voxels do not appear as cubes due to the spacing of the axis.

Figure 40. FOV of Robot Point Cloud and Voxelized Point Cloud

D. ROBOT CONTROL ALGORITHM

The robot control algorithm is a collection of algorithms, as shown in Figure 41. The control algorithm is composed of three primary sub algorithms, the point cloud filter and voxelization algorithm, the negative obstacle identification algorithm, and the potential field model. The previous sections detailed the point cloud filter and voxelization algorithm and the negative obstacle identification algorithm. These two algorithms identify points that represent obstacles to the robot. These points, along with the current pose of the robot, are then used as inputs to the potential field model which computes the relative attractive and repulsive forces acting on the robot as described in [3], [43], and [44]. The cumulative force acting on the robot is used to determine the translation and rotation commands the robot needs to execute to reach a pre-defined goal [44].

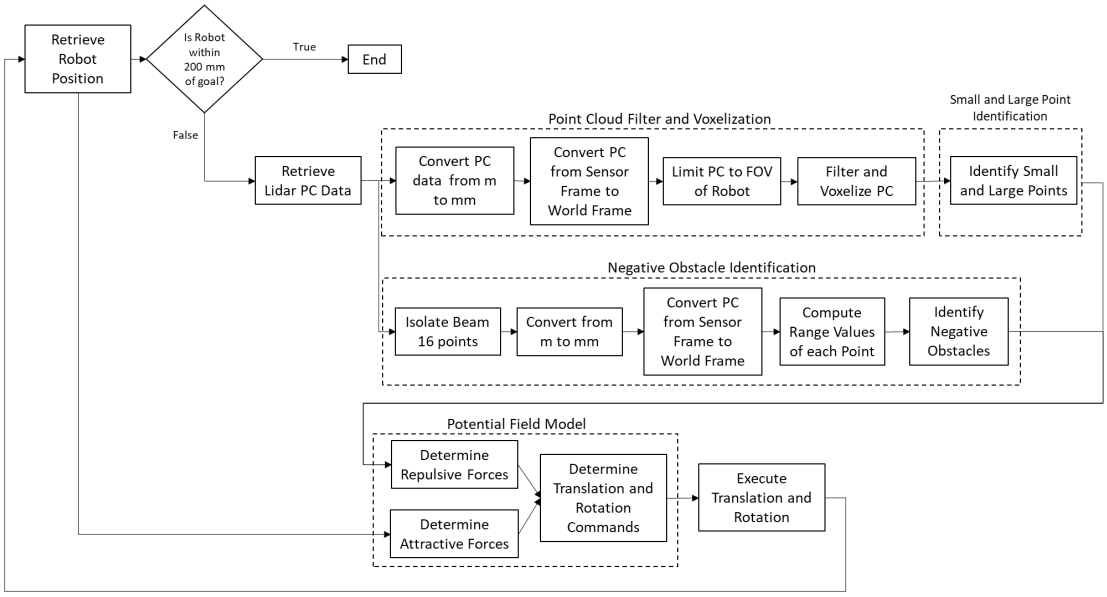


Figure 41. Block Diagram of the Control Algorithm

THIS PAGE INTENTIONALLY LEFT BLANK

VI. RESULTS

To evaluate the effectiveness of the algorithm and control method detailed in Chapter V, three experimental setups were used. For each experiment, the robot was given a goal in XYZ coordinates in the world frame and tasked with moving to within 200 mm of that goal. The environment was configured in such a way that the robot would have to navigate around a small obstacle, a large obstacle, or both to reach the goal. An additional fourth experiment was executed to highlight the ability of the lidar sensor to identify narrow objects and simulate the robot moving around a more natural obstacle. As previously stated in Chapter I, the wheel encoders built into the P3-DX were used to provide the robot with a pose estimate. The results of these experiments and the conclusions drawn from this work are presented here in Chapter VI.

A. LARGE OBSTACLE IDENTIFICATION AND AVOIDANCE

The first experiment involved the robot moving from an initial starting point to a point 6.0 m directly in front of the robot that served as the goal. Placed directly in the path of the robot was a large obstacle measuring 40.0 cm x 26.5 cm x 7.0 cm. The size of the obstacle was arbitrarily chosen to be significantly larger than the size of the small obstacle defined in Chapter III. The experimental setup is shown in Figure 42.

Figure 42. Experiment 1: Large Obstacle Detection and Avoidance

The robot successfully navigated around the obstacle to reach the goal. The path of the robot and the points detected as obstacles for the duration of the experiment are shown in Figure 43. The size of the blue circles represents the approximate size of the robot in relation to the obstacle and environment. The collection of green points on the right side of Figure 43 are the lidar returns off the wall directly behind the obstacle. The misclassification of points discussed in Chapter V can be seen in Figure 43 around the large obstacle. Additionally, the distribution of the points around the actual obstacle location should be noted. The nature of the point cloud filtering and voxelization process detailed in Chapter V accounts for some of this drift. Also contributing to the wide spread of obstacle points in relation to the actual obstacle location is the presence of erroneous points. These erroneous points are lidar returns from small irregularities in the ground plane such as the small gap between the floor tiles or the prominent edge of a piece of tape. Given that the obstacle identification and control algorithm only considers a single point cloud at a time, these erroneous points are only briefly detected, and the parameters of the potential field model are pre-set by trial and error to ensure that the erroneous points do not negatively impact the navigation of the robot. Finally, the time delay inherent in the

MATLAB script and the drift of the robot pose further add to the spread of the obstacle point locations. The elapsed time between determining the pose of the robot and the execution of steering commands due to the presence of an obstacle can be anywhere from 0.7 seconds to 1.2 seconds depending upon the environment. Within this time, the robot is still in motion and, depending on the translation rate of the robot, could move as far as 200 mm before receiving updated translation and rotation commands.

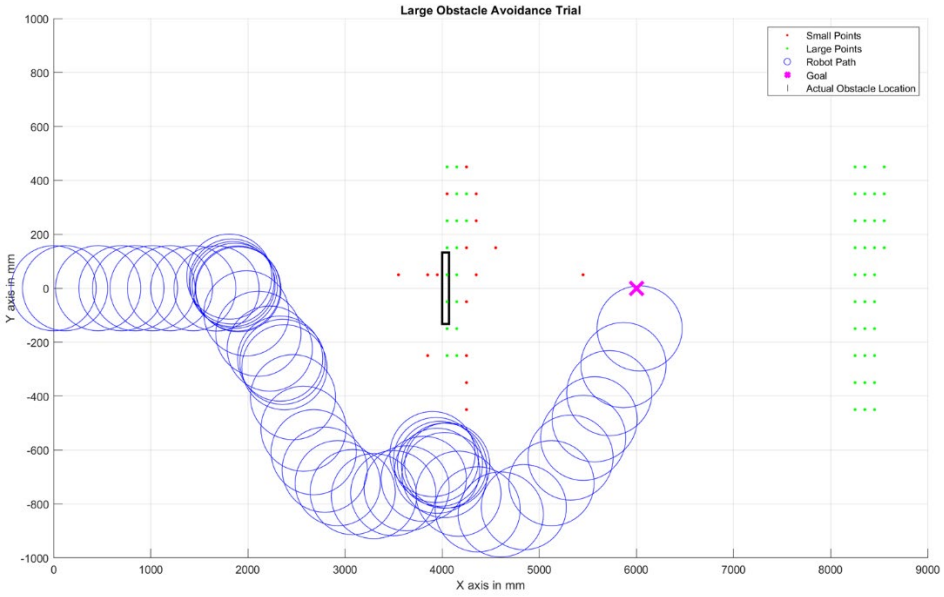


Figure 43. Robot Navigation Path for Experiment 1

B. SMALL OBSTACLE IDENTIFICATION AND AVOIDANCE

Similar to experiment one, experiment two involved the robot moving from an initial starting point to a goal 6.0 meters directly in front of the robot. A small obstacle with dimension 5.0 cm x 3.2 cm x 1.6 cm was placed directly in the path of the robot, as shown in Figure 44. The size of the small obstacle was determined based on the results found in Chapter III. The robot successfully navigated around the small obstacle to within 200 mm of the goal, as shown in Figure 45. The distribution of the small obstacle points shown in Figure 45 and the erroneous points are caused by the same factors discussed in the analysis of experiment one.

Figure 44. Experiment 2: Small Obstacle Detection and Avoidance

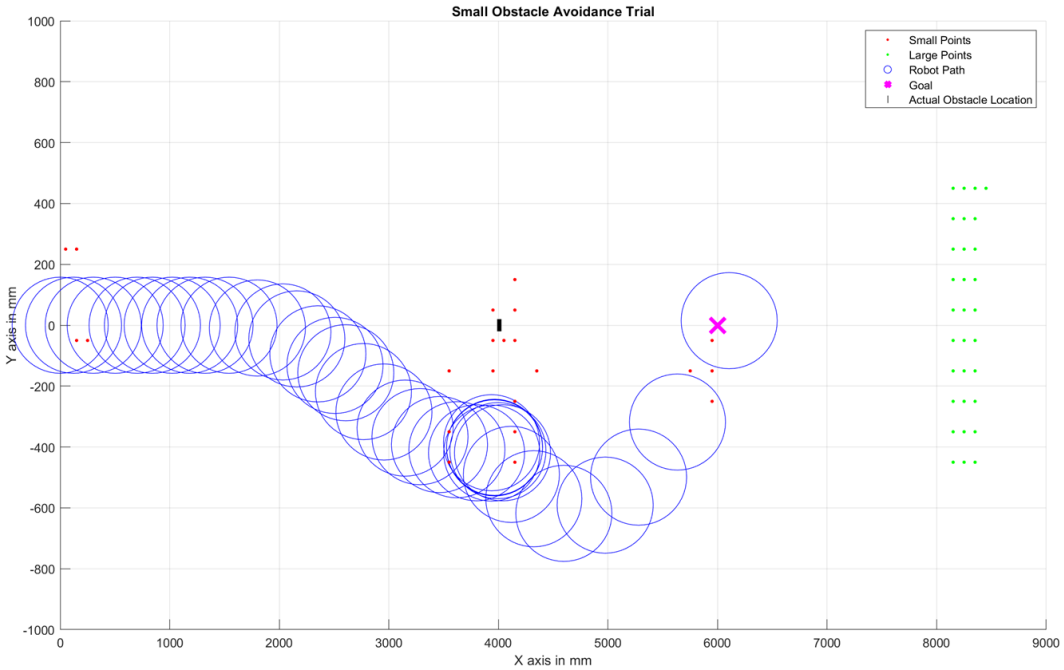


Figure 45. Robot Navigation Path for Experiment 2

C. COMBINED OBSTACLE IDENTIFICATION AND AVOIDANCE

Experiment three involved the robot again navigating to a goal 6.0 meters in front of the robot, this time with both a large and small obstacle placed in the path of the robot, as shown in Figure 46. The large obstacle used was identical to the obstacle used in experiment one. The size of the small obstacle used was 7.8 cm x 8.1 cm x 20.0 cm. A larger small obstacle than the obstacle used in experiment two was used to evaluate the versatility of the algorithm. The robot successfully navigated around both obstacles and reached the goal using the path shown in Figure 47. As discussed in the case of experiment one and two, erroneous points and a wide spread of obstacle points are shown in Figure 47. Additionally, from the path shown in Figure 47 the robot momentarily became stuck in front of the first obstacle as the robot looked for a clear path between the large obstacle and the metal cabinet shown in Figure 46.

Figure 46. Experiment 3: Combined Obstacle Detection and Avoidance

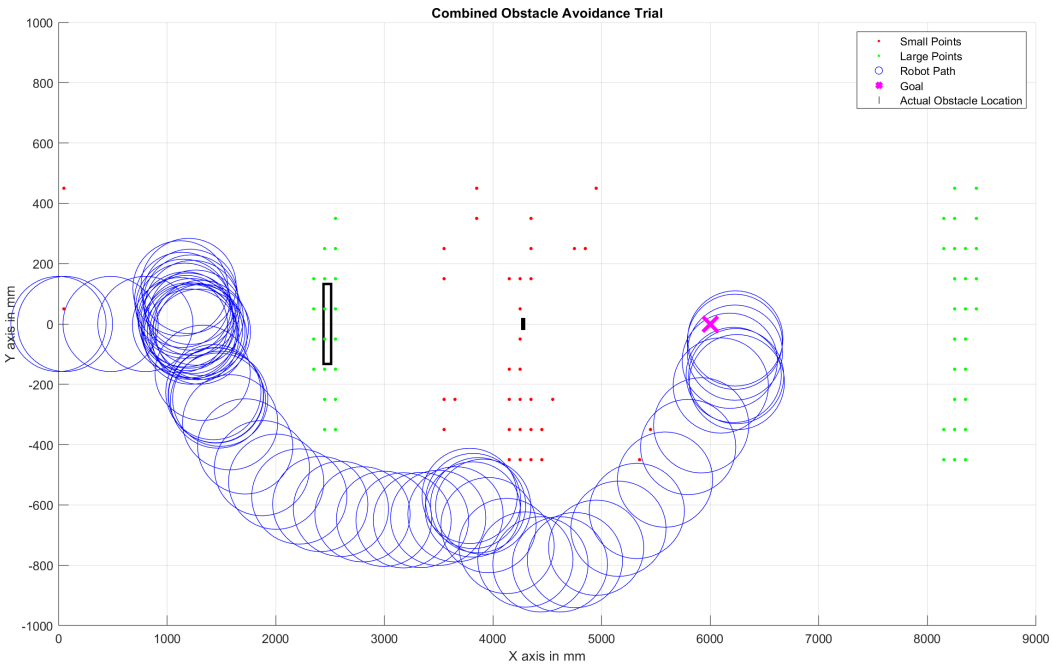


Figure 47. Robot Navigation Path for Experiment 3

D. TABLE OBSTACLE IDENTIFICATION AND AVOIDANCE

The fourth and final experiment required the robot to navigate around a table to reach a goal 7.0 meters directly in front of the robot. The table was positioned to ensure the robot needed to navigate around one of the table legs to reach the goal, as shown in Figure 48. The purpose of this experiment was to simulate the robot moving in a real- world indoor environment and to evaluate the ability of the OS1-16 sensor to detect a skinny obstacle such as a table leg. Furthermore, this experiment highlights the ability of the control algorithm to navigate the robot beneath obstacles and highlight this capability for future application on autonomous platforms with six degrees of freedom. The path taken by the robot to reach the goal is given in Figure 49. The same spread of obstacle points seen in experiments one, two, and three is present in experiment four due to the factors discussed in experiment one.

Figure 48. Experiment 4: Table Obstacle Detection and Avoidance

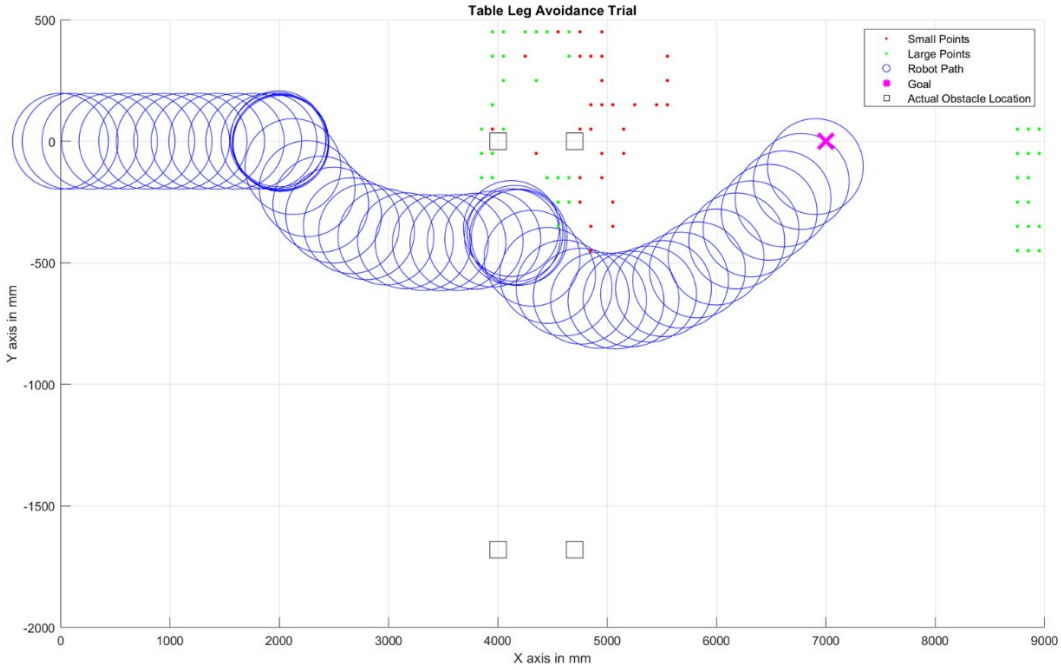


Figure 49. Robot Navigation Path for Experiment 4

The ability of the OS1-16 sensor to detect the legs of the table is clearly shown in Figure 50. The corresponding FOV of the robot is given in Figure 51 with the table leg clearly indicated.

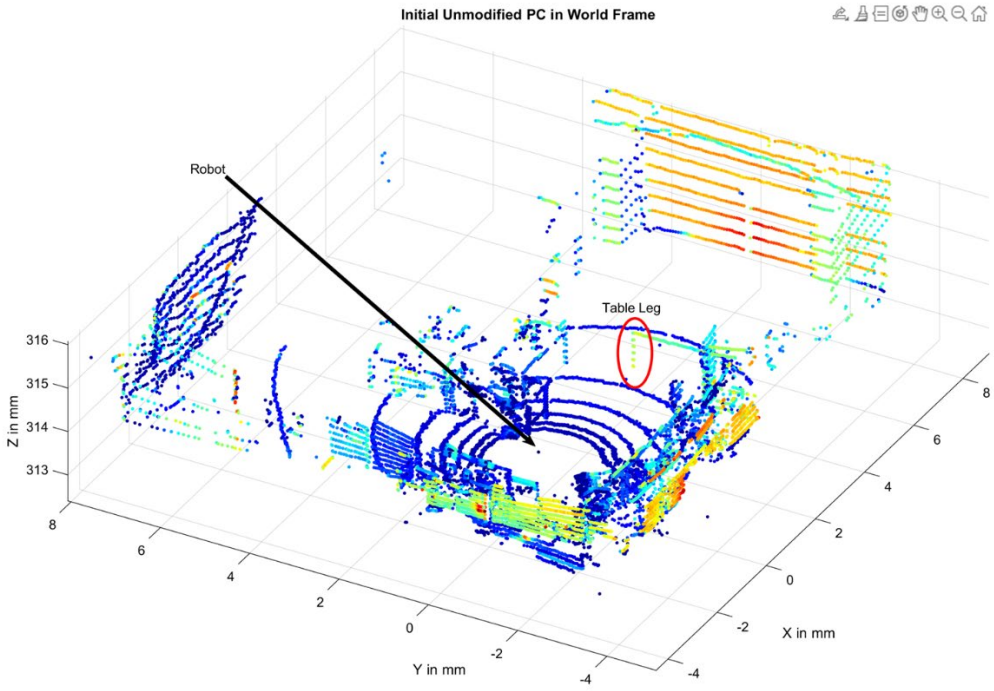


Figure 50. Unmodified Point Cloud of Experiment 4

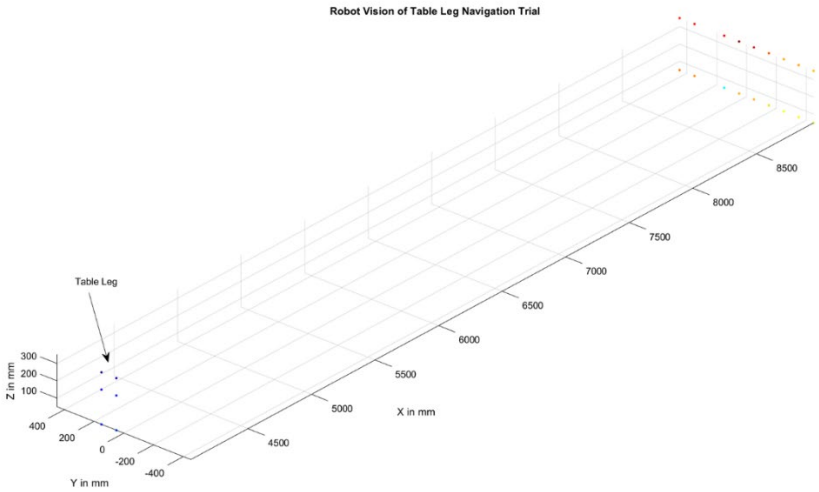


Figure 51. Robot Field of View for Experiment 4

VII. CONCLUSION

A. EVALUATION OF RESEARCH OBJECTIVES

The overall goal of this research, to evaluate the feasibility of conducting autonomous obstacle avoidance with input from a single lidar sensor, was accomplished. The experiments conducted in this research demonstrate the ability of the robot to identify and avoid both small and large obstacles. Furthermore, the capabilities of the OS1-16 lidar sensor were explored and the limitations regarding the size of detectable obstacles was determined.

B. LIMITATIONS

1. Negative Obstacle Testing

Given the limitations of the lab environment, a suitable negative obstacle was not available to test the negative obstacle detection algorithm developed in Chapter IV with the control algorithm from Chapter V. The work presented in Chapter IV demonstrates the ability of the sensor to detect negative obstacles but does not explore the feasibility of coupling this detection capability with the control method presented in Chapter V.

2. Obstacle Memory

The control algorithm developed in this work lacks any form of obstacle memory. As explained in Chapter V, the algorithm uses a single point cloud when detecting obstacles and responding with the necessary translation and rotation commands to the robot. When the next point cloud is received, the obstacles identified in the previous point cloud are forgotten. This means that if an obstacle is no longer in the FOV of the robot, the robot will not react to the obstacle. An obstacle memory feature was briefly implemented but further development and experimentation was abandoned due to time constraints. The obstacle memory algorithm developed is presented in Appendix A for potential use in future work.

3. Computing Power

The computing power of the R580 computer used as the control computer for this work was less than ideal. With the OS1-16 sensor running at 10 Hz, the computer can complete a single iteration of the while loop that receives, analyzes, and controls the robot in 0.8 seconds. As highlighted in Chapter VI, the delay caused by this time relative to the translation speed of the robot resulted in delayed steering commands to the robot and errors in obstacle location in the environment.

4. Algorithm Implementation

Like the limitations encountered in [4], the use of MATLAB to implement the algorithms developed in this work contributed to the time delay between receiving the input data from the sensor and executing the appropriate translation and rotation commands on the robot. The implementation of the algorithms developed in this work for a commercial system would require the use of a programming language that would allow for simultaneous execution of multiple algorithms or scripts [4].

5. Sensor Resolution

As detailed in Chapter II, the sensor used for this work was the 16 channel OS1-16. A higher resolution lidar sensor will allow for the detection of smaller obstacles and greater accuracy. A comparison between the OS1-16 and the 32 channel Ouster OS1-32 is given in Figure 52. The OS1-16 is capable of returning 16,384 points and the OS1-32 can return up to 32,768 points. The OS1-32 was not used in this work due to time constraints.

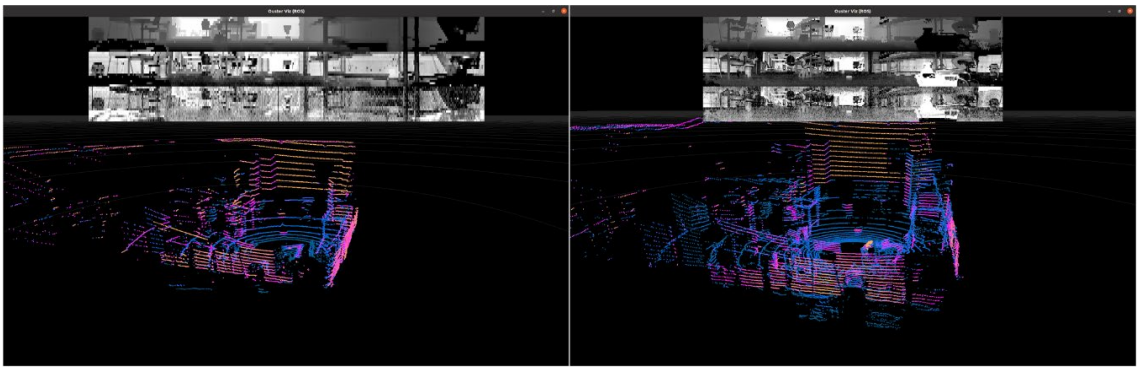


Figure 52. Comparison Between 16 Channel (Left) and 32 Channel (Right) Lidar Sensor

C. RECOMMENDATIONS FOR FUTURE WORK

1. Implementation on a Small Unmanned Aerial Vehicle

The algorithms developed in this work consider six degrees of freedom despite the P3-DX being limited to only three degrees of freedom. The reason for this design consideration is to allow for the use of the algorithms with a SUAV similar to the one used in [45]. The use of SUAV for indoor navigation would allow for a greater degree of mobility to navigate around identified obstacles. Furthermore, with adequate computing power, the speed of navigation could be increased.

2. Simultaneous Localization and Mapping

Combining the obstacle avoidance techniques developed in this work with the SLAM techniques developed in [4] would allow for a completely autonomous system that could quickly navigate through an interior space and provide real time map data to a user.

3. Obstacle Classification Based on Distance

To overcome the misclassification of obstacle points identified in Chapter V, a variable distance threshold could be implemented similar to the method utilized in [10]. In other words, the distance L between two points as shown in Figure 38 could vary based on the distance from the point to the sensor. This would decrease the possibility of misclassification, as discussed in Chapter V.

4. Implementation of Obstacle Memory

The lack of an effective obstacle memory necessitated the need to identify and react to small obstacles at a greater distance than normal to avoid missing the obstacles in the dead space around the robot. The implementation of an obstacle memory would allow the sensor to identify a small obstacle at an appreciable distance and remember the location of the obstacle in the world frame.

5. Route Planning

Given the initial snapshot by the lidar sensor from the starting position of the robot, a path could be planned through the environment. This path could be adhered to and only deviated from due to the presence of unanticipated obstacles. The detection of these obstacles could be solved using the algorithms developed in this work. The use of an octree filter for path planning in a voxel space might be a viable method to explore.

APPENDIX A. MATLAB SCRIPT

A. ROBOT_MASTER_CONTROL.M

```
%% Master Control Script for Robot Obstacle Avoidance
% Student: Bracci, Justin
% This code is adapted from the code exampleWander_DL.m written by Dr.
James Calusdian, NPS ECE dept.
% particularly the while loop structure and goal checking feature

clear all
close all
clc

%% Pre-Loop Section
% This section pre-defines constants and empty matrices before
initiating
% the while loop that actually controls the robot

% Define the Goal if applicable in the world frame in mm
goal = [7000 0 0]; %format: [x; y; z]

% Establish Range Clearance
range_Clearance = true; %initial value for while loop parameter

% Define Global Variables
global robot_height loopCounter Beam16_save p_c vs
vs = 100; % voxel size in mm
p_c = 3000; % this is the cutoff distance for small obstacles in mm
% it is defined here IOT factor into the ROI
robot_height = 500; % height of robot in world frame
% actual height is 387mm rounded to 400mm for use
% here

% Define ROI or Robot field of view
roi = [0, p_c+7000, -500, 500, -0.5, robot_height]; % Define region of
interest [xmin, xmax, ymin, ymax, zmin, zmax] in mm

% Counter to keep track of while loop
loopCounter = 0;
Max_Loop_Counter = 70; % Robot will stop when counter reaches this
number
runStatus = true; % initialize run status

% Turn memory on (true) or off (false)
Mem_OF = false;

% Initialize matrices
Obst_Memory_small = []; % Empty small object memory
Obst_Memory_Neg = []; % Empty large object memory
Robot_Coord_list = zeros(Max_Loop_Counter + 1, 4); % Pre-allocation of
robot path memory
```

```
Neg_Obst_Plot = []; % Empty negative obstacle recorder for post
plotting
large_points_plot = []; % Empty large obstacle recorder for post
plotting
small_points_plot = []; % Empty small obstacle recorder for post
plotting

% Subscribe to LiDAR Topic (See How to Subscribe to ROS Topics for more
% info)
rosinit; % Only necessary for remote activation, comment out when
running directly
pause(5) % pause to allow rosinit to activate
laser = rossubscriber('/os_cloud_node/points');

% Establish Beam 16 threshold for negative object detection
% IMPORTANT NOTE: In order for the robot to establish an accurate
% threshold, the robot must be initiated on the intended navigation
ground
% plane with the ROI clear of all positive and negative obstacles
PCloud = receive(laser, 10); % Recieve LiDAR data in ros PointCloud2
msg format
PCloud =
pointCloud(readXYZ(PCloud),'Intensity',readField(PCloud,'intensity'));%
Convert from ROS message PointCloud2 to Matlab pointCloud Object
[Th_16, Beam16] = Negative_Obstacle_TH(PCloud); % Determine threshold
values for negative obstacles
Beam16_save(:,:,1) = Beam16.Location; % Save the initial Beam 16 values
Beam16_Index_Save(1,:) = true(1,length(Beam16_save)); % Save and
initialize the first Beam 16 binary index

%% Conect to the Robot
% The connector script p3_connector_Payne.m used here was written by
Jameson
% Payne, adapted from one written by Dr. James Calusdian, NPS ECE dept.
% J. S. Payne, "Autonomous interior mapping robot utilizing lidar
% localization and mapping," M.S. thesis, Dept. Elect. And Comp. Eng.,
NPS,
% Monterey, CA, USA, 2020. [Online]. Available: http://hdl.handle.net/
10945/66121

p3_connector_Payne('/dev/ttyUSB0');
pause(5) % pause to allow matlab to catch up

%% Start While Loop and Robot Control

while (p3_getBumpersClear && range_Clearance && runStatus)
tic
% Establish Pose of robot
[x,y,theta] = p3_getXYHeading; % get pose of robot using wheel encoders
in mm and degrees
z = 0; % hard code robot z coordinate as 0, this would change for robot
with higher degrees of freedom
pose = [x, y, z, deg2rad(theta)]; % Robot pose in [x, y, z, heading]
format; change heading to radians
```

```
% Save Robot Pose for plotting
Robot_Coord_list(loopCounter+1,:) = pose;

% Pre-Process LiDAR data
PCloud = receive(laser, 10); % Recieve LiDAR data in ros PointCloud2
msg
PCloud =
pointCloud(readXYZ(PCloud),'Intensity',readField(PCloud,'intensity'));%
Convert from ROS message PointCloud2 to Matlab pointCloud Object
[PC_U1, PC] = OS1_FOV_4(PCloud, roi, pose); % Refine and down sample PC

% Identify Negative Obstacles
% NOTE: the input PC for negative obstacle detection is the
% non-downsampled PC (PC_U1)
[Beam16_save, Neg_Obstacle, Beam16_Index_Current] =
Negative_Obstacle_Detection(PC_U1, Th_16,
Beam16_Index_Save(loopCounter+1,:), pose);
Beam16_Index_Save(loopCounter+2,:) = Beam16_Index_Current;

% Identify 'small' and 'large' points
[small_points, large_points, points_index] = Obstacle_Class(PC);

% Determine potential field forces
[fwdVel, rotVel, Frep_r, Fatt, Ftotal, Frep_small_r, Frep_large_r,
small_points, large_points] = potentialField_2(pose, goal,
small_points, large_points, Obst_Memory_Neg, Obst_Memory_small,
Obst_Memory_Neg); %Determine Repulsive force Frep in [x, y, z] in robot
frame

% Set Robot Motion
p3_setTransVel(fwdVel);
p3_setRotVel(rotVel);

% For Testing and Presentation only, comment out for faster run times
% Create matrices to save for plotting: all points in world frame
large_points = Robot_to_World(large_points, pose);
small_points = Robot_to_World(small_points, pose);
Neg_Obst_Plot = Robot_to_World(Neg_Obst_Plot, pose);
large_points_plot = [large_points_plot; large_points];
small_points_plot = [small_points_plot; small_points];
Neg_Obst_Plot = [Neg_Obst_Plot; Neg_Obstacle];

% Manage Memory Bank for small and negative obstacles
[Obst_Memory_small, Obst_Memory_Neg] =
Obst_Memory_Manager(small_points, Neg_Obstacle, pose,
Obst_Memory_small, Obst_Memory_Neg, Mem_OF);

% Display formatted data for monitoring/troubleshooting
% Uncomment as needed
clc
% mystring1 = sprintf('x y heading %5.2f m %5.2f m %5.2f deg', pose(1),
pose(2), pose(4));
% disp(mystring1)
% mystring3 = sprintf('Attractive Force %7.2f %7.2f', Fatt(1),
Fatt(2));
```

```
% disp(mystring3)
% mystring2 = sprintf('Repulsive Force %7.2f %7.2f', Frep_r(1),
Frep_r(2));
% disp(mystring2)
% mystring5 = sprintf('Total Force %7.2f %7.2f', Ftotal(1), Ftotal(2));
% disp(mystring5)
% mystring8 = sprintf('Small Object Repulsive Force %7.2f %7.2f %7.2f',
Frep_small_r(1,1), Frep_small_r(1,2), Frep_small_r(1,3));
% disp(mystring8)
% mystring9 = sprintf('Large Object Repulsive Force %7.2f %7.2f %7.2f',
Frep_large_r(1,1), Frep_large_r(1,2), Frep_large_r(1,3));
% disp(mystring9)
% mystring6 = sprintf('Translation Velocity %7.2f', fwdVel);
% disp(mystring6)
% mystring7 = sprintf('Rotation Velocity %7.2f',rotVel);
% disp(mystring7)
% mystring10 = sprintf('Number of Negative Objects Detected %7.2f',
length(Neg_Obstacle));
% disp(mystring10)
% mystring11 = sprintf('Loop Count: %7.2f', loopCounter);
% disp(mystring11)
% mystring12 = sprintf('Distance to Goal: %7.2f', (norm(q-goal')));
% disp(mystring12)

% Stop Robot if within 200mm of goal
if (norm((pose(1,1:3))-goal)) < 200
p3_setTransVel(0);
p3_setRotVel(0);
range_Clearance = false;
disp('Goal has been reached. Ending Program.')
p3_disconnector
end

% Exit the while loop when finished
if(loopCounter < Max_Loop_Counter)
loopCounter = loopCounter + 1;
else
runStatus = false;
myString = 'Ending program...';
disp(myString);
p3_disconnector;
end
toc
end

%% Save Data
save('~/Desktop/sharefile2/Current_Run_Folder/
Robot_Path.mat','Robot_Coord_list');
save('~/Desktop/sharefile2/Current_Run_Folder/
Beam16_Index_Save.mat','Beam16_Index_Save');
save('~/Desktop/sharefile2/Current_Run_Folder/
Beam16_save.mat','Beam16_save');
save('~/Desktop/sharefile2/Current_Run_Folder/
Neg_Obst_Plot.mat','Neg_Obst_Plot');
```

```
save('~/Desktop/sharefile2/Current_Run_Folder/
Large_points_save.mat','large_points_plot');
save('~/Desktop/sharefile2/Current_Run_Folder/
Small_points_save.mat','small_points_plot');
```

B. NEGATIVE_OBSTACLE_TH.M

```
%% Negative Obstacle Thresholds

% This script will calculate the threshold values for beam 16 in mm.
This
% threshold value is then used in the function Negative_Obstacle_ID to
% determine if a negative obstacle is in the path of the beam.

% Inputs: PCloud => MATLAB pointcloud object
% Outputs: Th_16 => a scalar value in mm indicating the distance from
the
% sensor to the ground plane
% Beam16 => MATLAB pointcloud object containing 100 points of Beam 16
in mm
% in the World Frame.

function [Th_16, Beam16] = Negative_Obstacle_TH(PCloud)

%% Isolate Beams 16
Beam16 =
pointCloud(PCloud.Location(((1024*15)+1):((1024*16)),:),'Intensity',
PCloud.Intensity(((1024*15)+1):((1024*16)),:));

%% Identify ROI Points and convert to mm
% For this application, 100 points are used. This can be changed by
% adjusting the ranges (462:561) below but might require changes
elsewhere
% in other programs.
% IMPORTANT NOTE: The Negative_Obstacle_Detection function requires an
even
% number of points
Beam16 = pointCloud((Beam16.Location(462:561,:).*(10^3)),'Intensity',
PCloud.Intensity(462:561,:));

%% Remove Invalid Points from PC (NaN or Inf)
Beam16 = removeInvalidPoints(Beam16);

%% Transform the PC from Sensor Frame to World Frame
% NOTE: the pose of the robot is hard coded as 0 0 0 0 for the
% establishment of the threshold. If this is not the case, replace [0 0
0
% 0] with robot pose in world frame.
Beam16 = Sensor_to_World(Beam16, [0,0,0,0]); % convert PC to world
frame with robot pose assumed to be at origin

%% Identify Ranges in mm
Range16 = PC_Range(Beam16);

%% Identify Threshold in mm
Th_16 = mean(Range16);
end
```

C. OS1_FOV_4.M

```
%% OS1_FOV_4

% Input: PC_U => MATLAB pointCloud Object, unmodified straight from
sensor in meters
% roi => 1x6 region of interest matrix of the form [xmin, xmax, ymin,
ymax, zmin, zmax]
% r_pose => pose of robot in world frame in mm and radians [x, y, z,
% theta]
% Outputs: PC_U1 => original unmodified PC from sensor for testing and
% evaluation purposes
% PC_U => Down sampled and processed MATLAB pointCloud object in mm

% Pre-processing procedures of this function:
% Conversion of PC from meters to mm
% Conversion of PC from sensor to world
% narrowing of PC to include only points in ROI
% removal of all points below a defined intensity threshhold
% Down sampling of point cloud into 3D voxels of height, length, and
width
% vs (defined below)

function [PC_U1, PC_U] = OS1_FOV_4(PC_U, roi, r_pose)
global vs % establish the voxel size as a global variable
PC_U1 = PC_U; % save the PC in the world frame for use in Negative
Obstacle identification

%% Convert PC from meters to mm
PC_U = pointCloud((PC_U.Location.*(10^3)),'Intensity', PC_U.Intensity);

%% Remove Invalid Points from PC (NaN or Inf)
PC_U = removeInvalidPoints(PC_U);

%% Remove Points with coordinates [0 0 0]
index = bsxfun(@eq,PC_U.Location,(zeros(1,3))); % binary comparison of
matrix rows with [0, 0, 0]
index_3 = sum(index,2); % sums the rows which results in values of 3
indicating a point [0 0 0]
index_nz = (index_3 ~= 3); % non-zero index
PC_U = select(PC_U, index_nz); % PC with all points of [0 0 0] removed

%% Transform PC from sensor to world
PC_U = Sensor_to_World(PC_U, r_pose);

%% Identify ROI Points
indices = findPointsInROI(PC_U, roi); % Identify points in region of
interest
PC_ROI = select(PC_U, indices); % Create PC with point of interest

%% Remove points with Intensity < 200 & z < 0
int_threshold = 220; % Intensity threshold (most ground points are ~100
on lab floor surface)
robot_clearance = 60; % Clearance of robot in mm
```

```
% for the lab setting, by increasing the intensity threshold to 200,
this
% eliminates almost all of the ground returns while keeping the returns
for
% the 4cm high obstacles. This value was determined by manual
inspection of
% PCs and will likely differ depending on the composition of the ground
% plane
index = (PC_ROI.Intensity < int_threshold); %& (PC_ROI.Location(:,3) <
robot_clearance);
PC_ROI = select(PC_ROI, ~index);

%% Downsample PC using cubic voxels of size vs in mm

PC_DS = pcdownsample(PC_ROI, 'gridAverage', vs);

%% Replace point cloud data with center point of occupied voxels
sz = size(PC_DS.Location);
xyzVoxel = zeros(sz(1),3); % Pre-allocate matrix

for i = 1:sz(1)
% Coordinates of average location of points in voxel
x = PC_DS.Location(i,1);
y = PC_DS.Location(i,2);
z = PC_DS.Location(i,3);

% Center coordinate of occupied voxel
x_v = (floor(x/vs)*vs)+(vs/2);
y_v = (floor(y/vs)*vs)+(vs/2);
z_v = (floor(z/vs)*vs)+(vs/2);

% save voxel center points
xyzVoxel(i,:) = [x_v, y_v, z_v];
end

% Replace average points with voxel centers and re-define output PC_U
PC_U = pointCloud(xyzVoxel,'Intensity', PC_DS.Intensity); % Convert
from ROS message PointCloud2 to Matlab pointCloud Object

end
```

D. NEGATIVE_OBSTACLE_DETECTION.M

```
%% Negative Obstacle Detection

% This Function identifies negative obstacles in the forward field of
view
% of the robot.

% Input: PCloud => unmodified MATLAB pointCloud object from the sensor
% Th_16 => threshold value for beam 16 obtained from
% Negative_Obstacle_TH function
% Beam16_index => index of previous Beam 16 scan where 1 indicates
% clear area and 0 indicates a negative obstacle
```

```
% r_pose => pose of the robot in the world frame [x y z theta] in mm
and radians

% Output: Beam16_save => 3 dimensional matrix of the form n x m x L
where n
% is the number of points (in this case 100), m is 3
% for x, y, z in mm and L is the Beam 16 scan number in
% terms of the loop counter
% Neg_Obstacle => n x 3 matrix of the form [x, y, z] in mm of
% identified negative obstacles in the world frame
% where n is the number of identified obstacles
% Beam16_Index_Current => The current beam16 scan logical index
% where 1 indicates clear area and 0 indicates a negative obstacle

function [Beam16_save, Neg_Obstacle, Beam16_Index_Current]=
Negative_Obstacle_Detection(PCloud, Th_16, Beam16_Index, r_pose)
global loopCounter Beam16_save

%% Check for Negative Obstacles
[Beam16, Neg_Obstacle_index_16] = Negative_Obstacle_ID(PCloud, Th_16,
r_pose);
Beam16_save(:,:,loopCounter+2) = Beam16.Location; % Save t-1 Beam 16
scan
Beam16_last = Beam16_save(:,:,loopCounter+1); % xyz coordinates of Beam
16 scan at t-1

%% Binary Comparison
Neg_Obstacle_index_16 = Neg_Obstacle_index_16';
% Pre-Define Matrices
Index_P1 = false(1,100); % index for [1;0] comparison or 'ground' to
'hole'
Index_P2 = false(1,100); % index for [0;1] comparison or 'hole' to
'ground'
% Identify leading and trailing edges of negative obstacle by comparing
% previous scan to current scan and looking for binary transitions
for k = 1:100 % the 100 comes from the number of points from Beam 16
being analyzed
if Neg_Obstacle_index_16(1,k) == false && Beam16_Index(1,k) == true
Index_P1(1,k) = true;
else if Neg_Obstacle_index_16(1,k) == true && Beam16_Index(1,k) ==
false
Index_P2(1,k) = true;
end
end
end

% Identify transition points in current index: [1,0] or [0,1]
% essentiall identify left and right edges of negative obstacle from
% current scan
B = strfind(Neg_Obstacle_index_16,[1,0]);
C = strfind(Neg_Obstacle_index_16,[0,1]);
Index_C = false(1,length(Neg_Obstacle_index_16)); % Pre-define output
Index_C(B) = 1;
Index_C(C+1) = 1;
```

```
% Identify Negative Obstacles
Obstacles_Prev = Beam16_last(Index_P1',:);
Obstacles_Cur = Beam16.Location(Index_C',:);
Obstacles_Cur2 = Beam16.Location(Index_P2',:);
Neg_Obstacle = [Obstacles_Prev; Obstacles_Cur; Obstacles_Cur2];

% Save Beam Index
Beam16_Index_Current = Neg_Obstacle_index_16;

end
```

E. NEGATIVE_OBSTACLE_ID.M

```
%% Negative Obstacle ID

% This script identifies negative obstacles in the robots path by
analyzing
% 100 point returns from Beam 16 in the forward facing direction of the
% robot.

% Inputs: PCloud => Unmodified MATLAB pointcloud object
% Th_16 => a scalar value in mm indicating the distance from the
% sensor to the ground plane
% r_pose => pose of the robot in the world fram in mm and radians [x,
y, z, theta]
% Outputs: Beam16 => MATLAB pointcloud object containing 100 points of
Beam 16 in mm
% in the World Frame.
% Neg_Obstacle_index_16 => 1x 100 logical array where 1 indicates
% clear area and 0 indicates a negative obstacle

function [Beam16, Neg_Obstacle_index_16] = Negative_Obstacle_ID(PCloud,
Th_16, r_pose)

%% Isolate Beams 16
% NOTE: for a complete explanation see the section on OUSTER data PC
format
% in report appendix
Beam16 =
pointCloud(PCloud.Location(((1024*15)+1):((1024*16)),:),'Intensity',
PCloud.Intensity(((1024*15)+1):((1024*16)),:));

%% Identify ROI Points and convert to mm
% For this application, 100 points are used. This can be changed by
% adjusting the ranges (462:561) below but might require changes
elsewhere
% in other programs.
% IMPORTANT NOTE: The Negative_Obstacle_Detection function requires an
even
% number of points
Beam16 = pointCloud((Beam16.Location(462:561,:).*(10^3)),'Intensity',
PCloud.Intensity(462:561,:));

%% Remove Invalid Points from PC (NaN or Inf)
Beam16 = removeInvalidPoints(Beam16);
```

```
%% Transform the PC from Sensor Frame to World Frame
Beam16 = Sensor_to_World(Beam16, r_pose); % convert PC to world frame
with robot pose assumed to be at origin

%% Identify Ranges
Range16 = PC_Range(Beam16); % in mm

%% Account for range returns of 0
% NOTE: Currently, a range of 0 by the OS1-16 indicates that there was
no return for the point in question
% and it is assumed that the return is clear area, trusting points to
% the left and right to dtect the pressence of a negative obstacle
% large enough to impede the robot.
index = Range16 == 0;
Range16(index) = Th_16;

%% Identify Negative Obstacles
% The fudge factor is the +- distance from the threshold value (Th_16)
that
% the function will allow for. Using the angular orientation of the
Beam
% being used (in this case Beam 16) this value can be translated to
account
% for the acceptable drop the robot can navigate.
fudge_factor = 600; % in mm

% Negative Obstacles
Neg_Obstacle_index_16 = (Range16 - Th_16) > fudge_factor;
Neg_Obstacle_index_16 = ~Neg_Obstacle_index_16; % 1 is all clear; 0 is
obstacle

end
```

F. OBSTACLE_CLASS.M

```
%% Obstacle_Class calssifies points as either belonging to a small or
large object

% This function classifies each point in a PC as either a 'large' point
or
% a 'small point'. The function initially assumes that all points above
a
% certain height threshold are large. The Function then assumes that
all
% the points below the height threshold are 'small' and looks for
reasons
% to classify them as large (see explanation below).

% Input: PC => modified, voxelized point cloud in the world frame in mm

% Output: small_points => [x, y, z] coordinates of small points in mm
in
% world frame
% large_points => [x, y, z] coordinates of large points in mm in
```

```
% world frame
% points => a logical index of small (0) and large (1) points
% corresponding to the input PC

function [small_points, large_points, points] = Obstacle_Class(PC)
global vs

% set classification height for small points in world frame in mm, all
points with z component < small_height have the potential to be
% labeled as small points
small_height = 107; % in mm
sz = size(PC.Location);
tol = 0.01; % A very small value tolerance value experimentally
determined.
points = true(sz(1),1);

for i = 1 : sz(1)
if PC.Location(i,3) <= small_height % if this is true, there is the
potential for classification as 'small'
% find the 4 nearest neighbor points or neighbors
index = findNearestNeighbors(PC, (PC.Location(i,:)), 4);
neighbors = PC.Location(index,:);
szn = size(neighbors);
if szn(1) ~= 1 % This is the case if there is only 1 point in the FOV
and that point is small
% Identify if any of the nearest enighbors are directly above the
% point in question
P1 = PC.Location(i,:).*ones(size(neighbors));
dif = P1 - neighbors;
dif = dif(2:length(dif),:); % this is a 4x3 matrix of the x, y , and z
difference between the point in question and the 3 nearest neighbors
for k = 1 : length(dif)
index_temp = ismembertol(abs(dif(k,3)),vs,tol) &&
ismembertol(floor(abs(dif(k,1))),0,tol) &&
ismembertol(floor(abs(dif(k,2))),0,tol);
% 3 criteria are checked to identify if there is another point
% directly above the point in question
% 1.) does the neighbor occupy a voxel in the row above the
% current point
% 2.) is the neighbor in line with the point in question
% along the y-axis
% 3.) is the neighbor in line with the point in question in the
% x-direction
index_2(k) = index_temp;
end
if any(index_2)
% if the answer is 'true' to any of the above, the point is
% part of a larger object and can thus be classified as a
% 'large point'
points(i,1) = true;
else % the point is indeed a small point
points(i,1) = false;
end
else
points(i,1) = true;
```

```
end
else % classify as 'large' (logical 1)
points(i,1) = true;
end
% points is a sz(1) x 1 matrics of logical 1 and 0 which denote large
(1)
% and small (0) points
end

% Small Points
small_points = PC.Location(~points,:);

% Small Points
large_points = PC.Location(points,:);

end
```

G. POTENTIALFIELD_2.M

```
%% Function to determine the potential field forces
% This function outputs the attractive and repulsive force experienced
by
% the robot in the robot frame and is adapted from the code
POTENTIALFIELD.M written by
% J.S. Payne and the code exampleWander_DL.m written by Dr. James
Calusdian, NPS ECE dept. NPS/Calusdian image_wander_ROS

% This script employs the same method used in [43].
% Inputs:
% r_pose => pose of the robot in the world frame in mm and radians [x,
y,
% z, theta]
% goal => navigation goal in the world frame in mm [x, y, z];
% range => nx1 matrix of the ranges for each point in the processed
% voxelized point cloud
% small_points => [x, y, z] coordinates of small points in mm in world
frame
% large_points => [x, y, z] coordinates of small points in mm in world
frame
% Neg_Obstacle => n x 3 matrix of the form [x, y, z] in mm of
identified negative obstacles in the world frame
% where n is the number of identified obstacles
% points_index => a logical index of small (0) and large (1) points
% corresponding to the input PC
% Obst_Memory_small => small obstacles stored in the memory in the
world
% frame in [x, y, z] format
% Obst_Memory_Neg => negative obstacles stored in the memory in the
world
% frame in [x, y, z] format

% Outputs:
% fwdVel => translation velocity in mm/sec
% rotVel => rotational velocity in radians/sec
```

```
% Frep_r => Total repulsive force acting on the robot in the robot
frame
% Fatt_r => Total attractive force acting on the robot in the robot
frame
% Ftotal => Total force acting on the robot in the robot frame
% Frep_small_r => Repulsive force resulting from small obstacles in the
% robot frame
% Frep_large_r => Repulsive force resulting from large obstacles in the
% robot frame
% small_points => [x, y, z] coordinates in mm in the world frame of the
% small points influencing the robot
% large_points => [x, y, z] coordinates in mm in the world frame of the
% large points influencing the robot

%% Compute the Repulsive Force Frep_r
function [fwdVel, rotVel, Frep_r, Fatt_r, Ftotal, Frep_small_r,
Frep_large_r, small_points, large_points] = potentialField_2(r_pose,
goal, small_points, large_points, Neg_Obstacle, Obst_Memory_small,
Obst_Memory_Neg)
global p_c

% Potential Field Algorithm Parameters used to compute fwd_vel and
rot_vel
k1 = 0.7; % sensitivity for transVel
k2 = 12; % sensitivity for rotVel
dA = 1000; % distance in mm for attractive force to reduce robot speed
when close to goal
ZETA = 0.43; % gain for attractive force

%% Determine repulsive forces from 'small' points
% Potential Field Constants and Gains
eta = 1e11; % repulsive gain for small obstacles

small_points = cat(1,small_points,Obst_Memory_small); % add small
obstacles from memory
[small_points, range_s] = World_to_Robot(small_points,r_pose); %
Convert from world frame to robot frame and calculate range

index = range_s <= p_c; % Remove any points that are too far from the
robot
small_points = small_points(index,:); % pre-define matrix
if ~isempty(small_points) % if there are small points withing p_c
then...
sz = size(small_points);
Frep_small_r = zeros(3,sz(1)); % Pre-allocate repulsive force matrix
% Calculate Small Obstacle Repulsive Forces
for k = 1:sz(1)
Frep = eta*((1/range_s(k,:))-(1/p_c))*(1/((range_s(k,:))^3))*( -
(small_points(k,:)'));
Frep_small_r(:,k) = Frep';
end

% Filter out NaN Frep values resulting from ranges of 0
index = isnan(Frep_small_r);
Frep_small_r(index) = 0;
```

```
% Calculate Cumulative Small Repulsive Forces in the x, y and z
directions
Frep_small_r = [sum(Frep_small_r(1,:)); sum(Frep_small_r(2,:));
sum(Frep_small_r(3,:))]; %total repulsive forces in xyz coordinates in
robot frame
Frep_small_r = Frep_small_r'; % form of [Fx, Fy, Fz]

else
Frep_small_r = [0 0 0];
end

%% Determine Repulsive Force from 'Negative' points
% Potential Field Constants and Gains
eta_N = 5e10; % repulsive gain for negative obstacles
p_c_N = 1800; % cut off distance in mm for negative obstacles

Neg_Obstacle = cat(1,Neg_Obstacle,Obst_Memory_Neg); % add negative
obstacles from memory
[Neg_Obstacle, range_N] = World_to_Robot(Neg_Obstacle, r_pose); %
Convert from world frame to robot frame and calculate range

index = range_N <= p_c_N; % Remove any points that are too far from the
robot
Neg_Obstacle = Neg_Obstacle(index,:); % Pre-define matrix
if ~isempty(Neg_Obstacle) % if there are Negative Obstacles withing
p_c_N then...
sz = size(Neg_Obstacle);
Frep_neg_r = zeros(3,sz(1)); % Pre-allocate repulsive force matrix
% Calculate Negative Obstacle Repulsive Forces
for k = 1:sz(1)
Frep = eta_N*((1/range_N(k,:))-(1/p_c_N))*(1/((range_N(k,:))^3))*( -
(Neg_Obstacle(k,:)'));
Frep_neg_r(:,k) = Frep';
end

% Filter out NaN Frep values resulting from ranges of 0
index = isnan(Frep_neg_r);
Frep_neg_r(index) = 0;

% Calculate Cumulative Negative Obstacle Repulsive Forces in the x, y
and z directions
Frep_neg_r = [sum(Frep_neg_r(1,:)); sum(Frep_neg_r(2,:));
sum(Frep_neg_r(3,:))]; % total repulsive forces in xyz coordinates in
robot frame
Frep_neg_r = Frep_neg_r'; % form of [Fx, Fy, Fz]

else
Frep_neg_r = [0 0 0];
end

%% Determine repulsive forces from 'large' points
% Potential Field Constants and Gains
eta_L = 1e11; % Large Obstacle repulsive gain
p_c_L = 1800; % large obstacle cut off distance in mm
```

```
[large_points, range_L] = World_to_Robot(large_points,r_pose); %
Convert from world frame to robot frame and calculate range

index = range_L <= p_c_L; % Remove any points that are too far from the
robot
large_points = large_points(index,:); % Pre-define the matrix
if ~isempty(large_points) % if there are large points withing p_c_L
then...
sz = size(large_points);
Frep_large_r = zeros(3,sz(1)); % Pre-allocate repulsive force matrix
% Calculate Large Obstacle Repulsive Forces
for k = 1:sz(1)
Frep = eta_L*((1/range_L(k,:))-(1/p_c_L))*(1/((range_L(k,:))^3))*(-
(large_points(k,:)'));
Frep_large_r(:,k) = Frep';
end

% Filter out NaN Frep values resulting from ranges of 0
index = isnan(Frep_large_r);
Frep_large_r(index) = 0;

% Calculate Cumulative Large Obstacle Repulsive Forces in the x, y and
z directions
Frep_large_r = [sum(Frep_large_r(1,:)); sum(Frep_large_r(2,:));
sum(Frep_large_r(3,:))]; %total repulsive forces in xyz coordinates in
robot frame
Frep_large_r = Frep_large_r'; %form of [Fx, Fy, Fz]

else
Frep_large_r = [0 0 0];
end

%% Determine cumulative repulsive force
% this is where everything is put into the format [Fx; Fy; Fz]
Frep_r = (Frep_large_r') + (Frep_small_r') + (Frep_neg_r');

%% Determine attractive force
% Compute the Attractive Force: Fatt_r
dist = norm([r_pose(1); r_pose(2); r_pose(3)] - goal'); %distance from
robot to goal in mm

if dist <= dA
Fatt_w = -(ZETA)*([r_pose(1); r_pose(2); r_pose(3)] - goal');
%Attractive force in world coordinate frame
else
Fatt_w = -(ZETA)*dA*(([r_pose(1); r_pose(2); r_pose(3)] - goal')/dist);
%Attractive force in world coordinate frame
end

% Convert attractive force from world frame to robot frame
Fatt_r = [cos(r_pose(4)), sin(r_pose(4)), 0; (-sin(r_pose(4))),
cos(r_pose(4)), 0; 0, 0, 1]*Fatt_w;

%% Determine Fwd velocity and rotational velocity
```

```
Ftotal = Frep_r' + Fatt_r; % Total force acting on robot [Fx; Fy; Fz]
fwdVel = k1 * Ftotal(1); % translation velocity in mm/sec
rotVel = k2 * atan2(Ftotal(2), Ftotal(1)); % rotation velocity in rad/
sec

%% Set constraints on velocities (translation and rotation)
% The code in this section was taken from the script PotentialField.M
% written by Jameson Payne.
% J. S. Payne, "Autonomous interior mapping robot utilizing lidar
% localization and mapping," M.S. thesis, Dept. Elect. And Comp. Eng.,
% NPS, Monterey, CA, USA, 2020. [Online]. Available: %
http://hdl.handle.net/10945/66121

dir = atan2(Ftotal(2), Ftotal(1));

if dir > pi/2 || dir < -pi/2
if fwdVel > 0
fwdVel = 40; % This is minimum translation velocity (+-40)
elseif fwdVel < 0
fwdVel = -40;
end

if rotVel > 0
rotVel = 10; % This is minimum rotational velocity (+-10)
elseif rotVel < 0
rotVel = -10;
end
else
if fwdVel > 200 % This is maximum translation velocity (+-200)
fwdVel = 200;
end

if rotVel > 40 % This is maximum rotational velocity (+-40)
rotVel = 40;
end
end

end
```

H. ROBOT_TO_WORLD.M

```
%% Convert from Robot frame to World frame

% This function transforms x y z coordinates from the Robot frame to
the
% World frame using the techniques presented in [46] Inputs must be in
mm and outpute will be in mm
%
% input: PC_R => can either be a MATLAB pointCloud Object or a nx3
matrix
% of xyz points. Both data types must be in mm
% r_pose => pose of robot in world frame in mm and radians [x, y, z,
theta]
% Output: P_w => Same as input, either a pointCloud object or a matrix
```

```
function [P_W] = Robot_to_World(P_R, r_pose)
% Define the homogenous transform in mm
T_WR = [cos(r_pose(4)), (-sin(r_pose(4))), 0, r_pose(1);
(sin(r_pose(4))), cos(r_pose(4)), 0, r_pose(2);
0, 0, 1, (r_pose(3) + 313.5);
0, 0, 0, 1];

% Convert from robot frame to world frame
if isa(P_R,'pointCloud') % If the input is a MATLAB pointCloud object
sz = size(P_R.Location);
P_W = zeros(sz(1),3);
for i = 1:sz(1)
P_r = cat(1, P_R.Location(i,:)', [1]);
P_w = T_WR*P_r;
P_W(i,:) = P_w(1:3,1)';
end
P_W = pointCloud(P_W, 'Intensity', P_R.Intensity);

else % Otherwise assume the input is a nx3 matrix of xyz points
sz = size(P_R);
P_W = zeros(sz(1),3);
for i = 1:sz(1)
P_r = cat(1, P_R(i,:)', [1]);
P_w = T_WR*P_r;
P_W(i,:) = P_w(1:3,1)';
end
end
end
```

I. SENSOR_TO_ROBOT.M

```
%% Sensor to World Frame Translation

% This function transforms x y z coordinates from the sensor frame to
the
% world frame using the techniques presented in [46]. Inputs must be in
mm and outpute will be in mm
%
% input: PC_S => can either be a MATLAB pointCloud Object or a nx3
matrix
% of xyz points. Both data types must be in mm
% r_pose => pose of robot in world frame in mm and radians [x, y, z,
theta]
% Output: Same as input: either a pointCloud object or a matrix

function [PC_W] = Sensor_to_World(PC_S, r_pose)

% Define the homogenous transform
phi = deg2rad(1.54); % angular rotation about the y axis (1.54)

T_RS = [cos(phi), 0, (-sin(phi)), 0; % Homogenous transform for sensor
to robot
0, 1, 0, 0;
sin(phi), 0 cos(phi), 0;
```

```
0, 0, 0, 1];

T_WR = [cos(r_pose(4)), (-sin(r_pose(4))), 0, r_pose(1); % Homogenous
transform for robot to world
(sin(r_pose(4))), cos(r_pose(4)), 0, r_pose(2);
0, 0, 1, (r_pose(3) + 0313.5);
0, 0, 0, 1];

T_WS = T_WR*T_RS; % Homogenous transform for sensor to world

if isa(PC_S,'pointCloud') % If the input is a MATLAB pointCloud object
sz = size(PC_S.Location);
pc_w = zeros(sz(1),3); % define empty x y z coordinates for world PC

for i = 1:sz(1)
P_s = cat(1,PC_S.Location(i,:)',[1]);
P_w = T_WS*P_s;
pc_w(i,:) = P_w(1:3,1)';
end
PC_W = pointCloud(pc_w, 'Intensity', PC_S.Intensity);

else % Otherwise assume the input is a nx3 matrix of xyz points
sz = size(PC_S);
pc_w = zeros(sz(1),3); % define empty x y z coordinates for world PC

for i = 1:sz(1)
P_s = cat(1,PC_S(i,:)',[1]);
P_w = T_WS*P_s;
pc_w(i,:) = P_w(1:3,1)';
end
PC_W = pc_w;
end
```

J. ROBOT_PATH_PLOTTING.M

```
%% Robot path plotting
close all
clear all
clc

load('Large_points_save.mat')
load('Small_points_save.mat')
load('Robot_Path.mat')

% Define Robot z as ground plane
Robot_Z = zeros(1,length(Robot_Coord_list));

% Remove excessive small obstacles
index = small_points_plot(:,1) < 6000;
small_points_plot = small_points_plot(index,:);

figure(1)
hold on
scatter3(large_points_plot(:,1), large_points_plot(:,2),
zeros(1,length(large_points_plot)), 60, '.g')
```

```
scatter3(small_points_plot(:,1), small_points_plot(:,2),
zeros(1,length(small_points_plot)), 60, '.r')
scatter3(Robot_Coord_list(:,1), Robot_Coord_list(:,2), Robot_Z, 500,
'ob')
scatter3(6000, 0, 0, 100, 'xc')
scatter3(4000, 0, 0, 100, 'dk')
title('Robot Path and Small Obstacle Plot')
xlabel('X axis in mm')
ylabel('Y axis in mm')
zlabel('Z axis in mm')
ylim([-1000,1000])
legend('Large Points', 'Small Points', 'Robot Path', 'Goal', 'Actual
Obstacle Location')
grid on
```

K. OBST_MEMORY_MANAGER.M

```
%% Obstacle Memory Manager

% This function allows the robot to maintain a memory of obstacles in
the
% world frame that will influence the motion of the robot without being
in
% the current field of view. The memory disregards the 'oldest'
obstacles and saves the most 'recent' obstacles
%
% NOTE: Large obstacles are not stored in memory because it is assumed
that they are large enough to be seen by the sensor in real time. This
function was not used during testing as noted in Chapter VII.
%
% Inputs:
% small_points => small points in mm in [x y z] format in robot frame
from
% most recent scan that influenced the robot motion
% Neg_Obstacle => Negative Obstacles in mm in [x y z] format in robot
frame from
% most recent scan that influenced the robot motion
% r_pose => pose of the robot in the world frame
%
% Outputs:
% Obst_Memory_small => list of small obstacles stored in memory in [x,
y,
% z] format in mm in world frame
% Obst_Memory_Neg => list of negative obstacles stored in memory in [x,
y,
% z] format in mm in world frame

function [Obst_Memory_small, Obst_Memory_Neg] =
Obst_Memory_Manager(small_points, Neg_Obstacle, r_pose,
Obst_Memory_small, Obst_Memory_Neg, Mem_OF)
if Mem_OF % If false, outputs blank matrices for small and negative
obstacle memory

memory_length_small = 5; % Memory length for small obstacles
```

```
memory_length_neg = 10; % Memory length for negative obstacles

%% Convert from robot frame to world frame
small_points = Robot_to_World(small_points,r_pose);
Neg_Obstacle = Robot_to_World(Neg_Obstacle,r_pose);

%% Save Obstacles in memory
Obst_Memory_small = cat(1,small_points,Obst_Memory_small); % add most
recent obstacles to top of obstacles memory 'stack'
sz1 = size(Obst_Memory_small);
if sz1(1) > memory_length_small
Obst_Memory_small = Obst_Memory_small(1:memory_length_small,:); %
maintain memory length
else
end

Obst_Memory_Neg = cat(1,Neg_Obstacle, Obst_Memory_Neg); % add most
recent obstacles to top of obstacles memory 'stack'
sz2 = size(Obst_Memory_Neg);
if sz2(1) > memory_length_neg
Obst_Memory_Neg = Obst_Memory_Neg(1:memory_length_neg,:); % maintain
memory length
else
end

% %% Convert from robot frame to world frame
% % Define the homogenous transform
% T_WR = [cos(r_pose(4)), (sin(r_pose(4))), 0, r_pose(1);
% (-sin(r_pose(4))), cos(r_pose(4)), 0, r_pose(2);
% 0, 0, 1, (r_pose(3) + 313.5); % Recall that we are working in mm here
% 0, 0, 0, 1];
%
% % Convert from robot frame to world frame
% sz1 = size(Obst_Memory_small);
% sz2 = size(Obst_Memory_Neg);
% for i = 1:sz1(1)
% P_r = cat(1, Obst_Memory_small(i,:)', [1]);
% P_w = T_WR*P_r;
% Obst_Memory_small(i,:) = P_w(1:3,1)';
% end
%
% for i = 1:sz2(1)
% P_r = cat(1, Obst_Memory_Neg(i,:)', [1]);
% P_w = T_WR*P_r;
% Obst_Memory_Neg(i,:) = P_w(1:3,1)';
% end

else
Obst_Memory_small = [];
Obst_Memory_Neg = [];
end
end
```

L. P3_CONNECTOR_PAYNE.M

The following script is taken directly from [4].

```
function p3_connector(comString)
% p3_connector initializes the connection to the robot. This script was
% adapted from one written by Dr. James Calusdian, NPS ECE dept.
% p3_connector(comString) opens the communication with either the real
% robot or MobileSim. To connect to the actual robot the input
parameter
% comString must be set equal to 'Com1' or appropriate com port.
% To connect to MobileSim, comString must be set to 'MobileSim'.
% Also see p3_disconnector for additional information.

% in case we have some ports open from previous failed connections
if ~isempty(instrfindall)
delete(instrfindall);
end

if ~isempty(timerfindall)
delete(timerfindall)
end

global robotConnector;
global SIP_HANDLER;
global PULSE;

% first define the sync bytes that we need to use
SYNC0 = uint8([250 251 3 0 0 0]);
SYNC1 = uint8([250 251 3 1 0 1]);
SYNC2 = uint8([250 251 3 2 0 2]);
START_SERVER = uint8([250 251 3 1 0 1]);
ENABLE_MOTORS = uint8([250 251 6 4 59 1 0 5 59]);


% also define the constants and variables we need
syncState = 0; % switch parameter
sync0Lock = 0; % case parameter
sync1Lock = 1; % case parameter
sync2Lock = 2; % case parameter
syncLock012 = false; % overall sync status
tryCounter = 0; % number of attempts to communicate
MAX_TRIES = 3; % number of times to try synching up with robot



% determine what type of input we have
if nargin==0
s1 = sprintf('p3_connector FAIL! Must provide an input parameter');
s2 = sprintf('Exiting connector function.\n');
disp(s1);
disp(s2);
return;
else
if strcmp(comString, 'MobileSim')
```

```
s = sprintf('Connecting to MobileSim...');
disp(s);
% define our tcip connection
robotConnector = tcpip('localhost',8101); % connecto to MobileSim
%set(robotConnector,'Terminator','');
fopen(robotConnector); % open the connection

elseif strcmp(comString, comString)
s = sprintf('Connecting to real robot on Com1...');
disp(s);
% establish serial connection to the real robot...
robotConnector = serial(comString,'BaudRate',9600);
fopen(robotConnector);

else
s1 = sprintf('Input parameter not recognized');
s2 = sprintf('Exiting p3_connector function.\n');
disp(s1);
disp(s2);
return;

end
end


% send and verify our syncronization packets
while( ~syncLock012 )

switch syncState

case sync0Lock
s = sprintf('Sending Sync0');
disp(s);
fwrite(robotConnector, SYNC0);
[response, counts] = fread(robotConnector, [1 6],'uint8');
if isequaln(response, SYNC0)
syncState = sync1Lock;
s = sprintf('Sync0 acknowledged\n');
disp(s);

else
if tryCounter < MAX_TRIES
syncState = sync0Lock;
syncLock012 = false;
tryCounter = tryCounter + 1;
s = sprintf('Sync0 fail. Sending Sync0 again\n');
disp(s);
else
syncLock012 = true; % set to TRUE to get out of while-loop
s = sprintf('Sync0 fail. Max tries exceeded\nClosing local port\n');
disp(s);
fclose(robotConnector);
end

end
```

```
case sync1Lock
s = sprintf('Sending Sync1');
disp(s);
fwrite(robotConnector, SYNC1);
[response, counts] = fread(robotConnector, [1 6], 'uint8');
if isequaln(response, SYNC1)
syncState = sync2Lock;
s = sprintf('Sync1 acknowledged\n');
disp(s);
syncLock012 = false;

else
if tryCounter < MAX_TRIES
syncState = sync1Lock;
syncLock012 = false;
tryCounter = tryCounter + 1;
s = sprintf('Sync1 fail. Sending Sync1 again\n');
disp(s);
else
syncLock012 = true; % set to TRUE to get out of while-loop
s = sprintf('Sync1 fail. Max tries exceeded\nClosing local port\n');
disp(s);
fclose(robotConnector);
end
end

case sync2Lock
s = sprintf('Sending Sync2');
disp(s);
fwrite(robotConnector, SYNC2);pause(0.8);
bytesAvail = robotConnector.BytesAvailable;
[response, counts] = fread(robotConnector, [1 bytesAvail],'uint8');
s = sprintf('Connected to %s\n',response(3:end-3));
disp(s);

% send the OPEN command to start up server
fwrite(robotConnector, START_SERVER);

% start up heartbeat timer
p3_heartbeatTimer;
answer = PULSE.Running;
s = sprintf('Heartbeat timer is %s\n',answer);
disp(s);

% start the SIP handler (timer) to read packets
p3_SIP_Timer;
answer = SIP_HANDLER.Running;
s = sprintf('SIP Handler is %s\n',answer);
disp(s);

% send command 4 to enable the motors
fwrite(robotConnector, ENABLE_MOTORS); pause(0.8)

% break out of this loop
```

```
syncLock012 = true;


end % switch-case

end % while

pause(5); % wait a few seconds for everything to sync up
```

APPENDIX B. P3-DX MATLAB CODES

All code in this section was written by Dr. James Calusdian, NPS ECE Department [16].

A. P3_GETBUMPERSCLEAR.M

```
function [bumpersClear] = p3_getBumpersClear
%P3_GETBUMPERSCLEAR returns true if ALL bumpers clear, false otherwise.
%
% Copyright Naval Postgraduate School, 2015

global SIPdata;
HEADER_BYTE0 = uint8(250);
HEADER_BYTE1 = uint8(251);

% if we have the SIPdata available, we can pull out the battery
voltage.
% First, let's double check that we have the right data
if SIPdata(1) == HEADER_BYTE0
if SIPdata(2) == HEADER_BYTE1
bumperStatus = make16(SIPdata(17),SIPdata(16));

if bumperStatus
bumpersClear = false;
%disp('FALSE');
else
bumpersClear = true;
%disp('TRUE');
end


end
end
```

B. P3_GETXYHEADING.M

```
function [xPos,yPos,theta] = p3_getXYHeading
%P3_GETXYHEADING returns the robot's coordinates and heading.
% [xPos,yPos,theta] = p3_getXYHeading returns the x and y coordinates
of
% the robot in millimeters, and theta is the robot heading in degrees.
%
% Copyright Naval Postgraduate School, 2015

global SIPdata;
HEADER_BYTE0 = uint8(250);
HEADER_BYTE1 = uint8(251);
angleConvFactor = (2*pi/4096)*360/(2*pi); % degs per count
```

```
% if we have the SIPdata available, then we can pull out the xPos,
yPos,
% and heading information.

% first, let's double check that we have the right data by checking the
% sync bytes again.
if SIPdata(1) == HEADER_BYTE0
if SIPdata(2) == HEADER_BYTE1
tempX = make16(SIPdata(6),SIPdata(5));
tempY = make16(SIPdata(8),SIPdata(7));
tempTheta = make16(SIPdata(10),SIPdata(9));
xPos = double(convertToSignedInt(tempX));
yPos = double(convertToSignedInt(tempY));
theta = double(convertToSignedInt(tempTheta)) * angleConvFactor;
theta = wrapTo180(theta);
end
else
xPos = NaN;
yPos = NaN;
theta = NaN;
end
```

C. P3_SETTRANSVEL.M

```
function p3_seTransVel(transVel)
%P3_SETTRANSVEL sets the translational velocity for the robot.
% p3_setTransVel(transVel) causes the robot to move forward (+) or
% backward (-) at the speed of "transVel" mm/sec.
% See ARCOS command 11 in Operations Manual, pp 31.
%
% Copyright Naval Postgraduate School, 2015

global robotConnector;
HEADER_BYTE0 = uint8(250);
HEADER_BYTE1 = uint8(251);
commandNumber = uint8(11);

% construct the command to rotate
if transVel < 0
argType = uint8(27); % negative
else
argType = uint8(59); % positive
end

% convert "rotVel" into two bytes of uint8
temp = uint16(abs(transVel));
[MSB, LSB]=split16(temp);

% next construct the command packet
command = uint8([commandNumber argType LSB MSB ]);
byteCount = uint8(length(command) + 2); % include 2 checkSum bytes
[chkMSB, chkLSB] = checksum4p3(command);
translateCommand = uint8([HEADER_BYTE0 HEADER_BYTE1 byteCount command
chkMSB chkLSB]);
```

```
% send everything to the robot
fwrite(robotConnector, translateCommand);
```

D. P3_SETROTVEL.M

```
function p3_setRotVel(rotVel)
%P3_SETROTVEL causes the robot to rotate ccw(+) or cw(-) at specified
deg/sec.
% p3_setRotVel(rotVel) causes the robot to rotate with the angular
velocity
% specified in rotVel (deg/second). A (+) rotVel produces a CCW
rotation,
% and a (-) rotVel produces a CW rotation, as viewed from the top of
the
% robot.
% See ARCOS command 9 in Operations Manual, pp 31.
%
% Copyright Naval Postgraduate School, 2015

global robotConnector;
HEADER_BYTE0 = uint8(250);
HEADER_BYTE1 = uint8(251);
commandNumber = uint8(9);

% construct the command to rotate
if rotVel < 0
argType = uint8(27); % negative
else
argType = uint8(59); % positive
end

% convert "rotVel" into two bytes of uint8
temp = uint16(abs(rotVel));
[MSB, LSB]=split16(temp);

% next construct the command packet
command = uint8([commandNumber argType LSB MSB ]);
byteCount = uint8(length(command) + 2); % include 2 checkSum bytes
[chkMSB, chkLSB] = checksum4p3(command);
setRotCommand = uint8([HEADER_BYTE0 HEADER_BYTE1 byteCount command
chkMSB chkLSB]);

% send everything to the robot
fwrite(robotConnector, setRotCommand);
```

E. P3_DISCONNECTOR.M

```
function p3_disconnector
% P3_DISCONNECTOR disconnects from the robot or MobileSim.
% This function disconnects from the robot by stopping the SIP handler,
% stopping the PULSE timer, and closing the robotConnector object. No
input
% or output parameters are required for this function. Also see
p3_connector.
%
```

```
% Copyright Naval Postgraduate School, 2015

global robotConnector;
global SIP_HANDLER;
global PULSE;
global myTestData;

% first define the sync bytes that we need to use
CLOSE_CONNECTION = uint8([250 251 3 2 0 2]);


% stop processing the SIP packets
s = sprintf('Stopping the SIP handler...\n');
disp(s);
stop(SIP_HANDLER);
answer = SIP_HANDLER.Running;
s = sprintf('SIP Handler is %s',answer);
disp(s);
answer = SIP_HANDLER.AveragePeriod;
s = sprintf('SIP_HANDLER timer period actual is %7.2f seconds',
answer);
disp(s);
delete(SIP_HANDLER);
%save('testData.mat','myTestData');

% stop the heartbeat, which was started with p3_heartbeatTimer.
s = sprintf('Stopping PULSE heartbeat...\n');
disp(s);
answer = PULSE.AveragePeriod;
s = sprintf('PULSE timer period actual is %7.2f seconds', answer);
disp(s);
stop(PULSE);
answer = PULSE.Running;
s = sprintf('Pulse heartbeat is %s',answer);
disp(s);
delete(PULSE);


% close the robot connection
s = sprintf('Closing robot connection');
disp(s);
fwrite(robotConnector,CLOSE_CONNECTION); pause(0.8);
fclose(robotConnector);
delete(robotConnector);
```

LIST OF REFERENCES

[1] Velodyne Lidar, "Autonomous vehicles," Accessed Jun. 11, 2021. [Online]. Available: https://velodynelidar.com/industries/autonomous/

[2] Ouster, "High-performance digital lidar solutions," Accessed Jun. 11, 2021. [Online]. Available: https://ouster.com/

[3] R. Siegwart, I. Nourbakhsh, and D. Scaramuzza, *Introduction to Autonomous Mobile Robots*, 2nd ed. Cambridge, MA, USA: The MIT Press, 2011.

[4] J. S. Payne, "Autonomous interior mapping robot utilizing lidar localization and mapping," M.S. thesis, Dept. Elect. and Comp. Eng., NPS, Monterey, CA, USA, 2020. [Online]. Available: http://hdl.handle.net/10945/66121

[5] *2018 U.S. Marine Corps S&T Strategic Plan*. Marine Corps Warfighting Laboratory/Futures Directorate, Quantico, VA, USA, 2017. [Online]. Available: https://www.onr.navy.mil/-/media/Files/About-ONR/2018-USMC-S-and-T-Strategic-Plan.ashx?la=en&hash=73B2574A13A8EC6AAE60CF4670E05C6F97309B8F

[6] A. Discant, A. Rogozan, C. Rusu, and A. Bensrhair, "Sensors for obstacle detection - a survey," in *2007 30th International Spring Seminar on Electronics Technology (ISSE)*, Cluj-Napoca, Romania, May 2007, pp. 100–105. doi: 10.1109/ISSE.2007.4432828.

[7] A. S. Miyakawa, "Autonomous ground vehicle low-profile obstacle avoidance using 2D LIDAR," M.S. thesis, Dept. Elect. and Comp. Eng., NPS, Monterey, CA, USA, 2019. [Online]. Available: http://hdl.handle.net/10945/63486

[8] N. Bernini, M. Bertozzi, L. Castangia, M. Patander, and M. Sabbatelli, "Real-time obstacle detection using stereo vision for autonomous ground vehicles: A survey," in *17th International IEEE Conference on Intelligent Transportation Systems (ITSC)*, Qingdao, China, Oct. 2014, pp. 873–878. doi: 10.1109/ITSC.2014.6957799.

[9] A. Asvadi, C. Premebida, P. Peixoto, and U. Nunes, "3D Lidar-based static and moving obstacle detection in driving environments: An approach based on voxels and multi-region ground planes," *Robotics and Autonomous Systems*, vol. 83, pp. 299–311, Sep. 2016, doi: 10.1016/j.robot.2016.06.007.

[10] N. Baras, G. Nantzios, D. Ziouzios, and M. Dasygenis, "Autonomous obstacle avoidance vehicle using LIDAR and an embedded system," in *2019 8th International Conference on Modern Circuits and Systems Technologies (MOCAST)*, 2019, pp. 1–4. doi: 10.1109/MOCAST.2019.8742065.

[11] S. Hening, C. A. Ippolito, K. S. Krishnakumar, V. Stepanyan, and M. Teodorescu, "3D LiDAR SLAM integration with GPS/INS for UAVs in urban GPS-degraded environments," presented at the AIAA Information Systems-AIAA Infotech @ Aerospace, Grapevine, Texas, USA, Jan. 2017. doi: 10.2514/6.2017-0448.

[12] M. Tulldahl, H. Larsson, G. Tolt, F. Bissmarck, C. Grönwall, and J. Nordlöf, "Application and capabilities of lidar from small UAV," Baltimore, MD, USA, May 2016, p. 98320V. doi: 10.1117/12.2224258.

[13] D. Droeschel and S. Behnke, "Efficient continuous-time SLAM for 3D lidar-based online mapping," in *2018 IEEE International Conference on Robotics and Automation (ICRA)*, Brisbane, QLD, Australia, May 2018, pp. 5000–5007. doi: 10.1109/ICRA.2018.8461000.

[14] E. Shang, X. An, J. Li, and H. He, "A novel setup method of 3D LIDAR for negative obstacle detection in field environment," in *17th International IEEE Conference on Intelligent Transportation Systems (ITSC)*, Qingdao, China, Oct. 2014, pp. 1436–1441. doi: 10.1109/ITSC.2014.6957888.

[15] Adept Technology Inc., *Pioneer 3-DX*. Pioneer 3-DX Data Sheet, 2011.

[16] J. Calusdian, "Help documentation for MATLAB functions used with the P3 mobile robot," unpublished.

[17] C. S. Hargadine, "Mobile robot navigation and obstacle avoidance in unstructured outdoor environments," M.S. thesis, Dept. Elect. and Comp. Eng., NPS, Monterey, CA, USA, 2017. [Online]. Available: http://hdl.handle.net/10945/56937

[18] X. Guoan, "3 ways to use SSH on windows to log into Linux server," *LinuxBabe*, Oct. 24, 2019. Accessed Jul. 20, 2021. [Online]. Available: https://www.linuxbabe.com/linux-server/ssh-windows

[19] Ouster, "OS1 gen 1 (serial numbers starting with "os1-") mid-range high-resolution imaging lidar," 2.0. 2020. [Online]. Available: https://data.ouster.io/downloads/datasheets/datasheet-gen1-v2p0-os1.pdf

[20] T. Gray, "Ouster range and precision webinar 06.2020," San Francisco, CA, USA, Jun. 11, 2020. [Online]. Available: https://go.ouster.io/webinar/how-to-understand-lidar-performance-range-precision-accuracy/thank-you/?submissionGuid=389fdcea-3039-471e-9bce-8b9228074695

[21] J. Tatum, "1.13 Lambertian surface," in *Stellar Atmospheres*, 2017, pp. 12–13. Accessed: Jul. 20, 2021. [Online]. Available: http://orca.phys.uvic.ca/~tatum/stellatm.html

[22] M. Young, *Optics and Lasers: Including Fibers and Optical Waveguides*, 5th ed. Berlin: Springer, 2000.

[23] M. Pharr, W. Jakob, and G. Humphreys, "09 - Materials," in *Physically Based Rendering*, 3rd ed., M. Pharr, W. Jakob, and G. Humphreys, Eds. Boston: Morgan Kaufmann, 2017, pp. 571–594. doi: 10.1016/B978-0-12-800645-0.50009-9.

[24] "Range resolution," Radar Tutorial, 1998. Accessed Jul. 20, 2021. [Online]. Available: https://www.radartutorial.eu/01.basics/Range%20Resolution.en.html

[25] P. Angus, "Lidar as a camera - digital lidar's implications for computer vision," *Ouster*, Aug. 31, 2018. Accessed Jul. 20, 2021. [Online]. Available: https://ouster.com/blog/the-camera-is-in-the-lidar

[26] MathWorks, "What Is MATLAB?" Accessed Jul. 20, 2021. [Online], Available: https://www.mathworks.com/discovery/what-is-matlab.html

[27] MathWorks, "Lidar Toolbox," Accessed Jul. 20, 2021. [Online], Available: https://www.mathworks.com/products/lidar.html

[28] MathWorks, "ROS Toolbox," Accessed Jul. 20, 2021. [Online], Available: https://www.mathworks.com/help/ros/index.html?s_tid=CRUX_lfnav

[29] ROS, "About ROS," Accessed Jul. 20, 2021. [Online], Available: https://www.ros.org/about-ros/

[30] J. Lambert *et al.*, "Performance Analysis of 10 Models of 3D LiDARs for Automated Driving," *IEEE Access*, vol. 8, pp. 131699–131722, 2020, doi: 10.1109/ACCESS.2020.3009680.

[31] T. Grey, "Effective range and the high resolution advantage," *Ouster*, Aug. 07, 2020. Accessed Jul. 20, 2021. [Online], Available: https://ouster.com/blog/effective-range-and-resolution

[32] V. Lindberg, "Propagation of errors, basic rules," in *Uncertainties and Error Propagation*, 2000. Accessed: Jul. 20, 2021. [Online]. Available: https://www.geol.lsu.edu/jlorenzo/geophysics/uncertainties/Uncertaintiespart2.html

[33] M. Raffi, "Lidar vs. camera: driving in the rain," *Ouster*, Feb. 04, 2020. Accessed Jul. 20, 2021. [Online], Available: https://ouster.com/blog/lidar-vs-camera-comparison-in-the-rain

[34] J. S. Evans and A. T. Hudak, "A multiscale curvature algorithm for classifying discrete return LiDAR in forested environments," *IEEE Trans. Geosci. Remote Sensing*, vol. 45, no. 4, pp. 1029–1038, Apr. 2007, doi: 10.1109/TGRS.2006.890412.

[35] A. Murarka, M. Sridharan, and B. Kuipers, "Detecting obstacles and drop-offs using stereo and motion cues for safe local motion," in *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Nice, France, Sep. 2008, pp. 702–708. doi: 10.1109/IROS.2008.4651106.

[36] H. Seraji, "Rule-based traversability indices for multi-scale terrain assessment," *IFAC Proceedings Volumes*, vol. 37, no. 7, pp. 159–164, Jun. 2004, doi: 10.1016/S1474-6670(17)32141-9.

[37] J. Larson and M. Trivedi, "Lidar based off-road negative obstacle detection and analysis," in *2011 14th International IEEE Conference on Intelligent Transportation Systems (ITSC)*, Oct. 2011, pp. 192–197. doi: 10.1109/ITSC.2011.6083105.

[38] T. Hong, S. Legowik, and M. Nashman, "Obstacle detection and mapping system," in *NISTIR 6213,* Aug. 1998, pp. 1–22.

[39] Y. Roth-Tabak and R. Jain, "Building an environment model using depth information," *Computer*, vol. 22, no. 6, pp. 85–90, Jun. 1989, doi: 10.1109/2.30724.

[40] T. Hinks, H. Carr, L. Truong-Hong, and D. F. Laefer, "Point cloud data conversion into solid models via point-based voxelization," *J. Surv. Eng.*, vol. 139, no. 2, pp. 72–83, May 2013, doi: 10.1061/(ASCE)SU.1943-5428.0000097.

[41] D. Cohen, A. Kaufman, and Y. Wang, "Generating a smooth voxel-based model from an irregular polygon mesh," *The Visual Computer*, vol. 10, no. 6, pp. 295–305, Jun. 1994, doi: 10.1007/BF01900824.

[42] MathWorks, "pcdownsample," Accessed Jul. 20, 2021. [Online], Available: https://www.mathworks.com/help/vision/ref/pcdownsample.html

[43] X. Yun and K. Tan, "A wall-following method for escaping local minima in potential field based motion planning," in *1997 8th International Conference on Advanced Robotics. Proceedings. ICAR'97*, Monterey, CA, USA, 1997, pp. 421–426. doi: 10.1109/ICAR.1997.620216.

[44] X. Yun, "Fundamentals of Robotics Part 5: Motion Planning," class notes for EC4310 Fundamentals of Robotics, Dept. Elect. and Comp. Eng., NPS, Monterey, CA, USA, winter 2020.

[45] L. Hardesty, "Flight of fancy," *MIT News | Massachusetts Institute of Technology*, Dec. 03, 2009. Accessed Jul. 20, 2021. [Online], Available: https://news.mit.edu/2009/helicopters-1203

[46] X. Yun, “Fundamentals of Robotics Part 1: Description of Position and Orientation,” class notes for EC4310 Fundamentals of Robotics, Dept. Elect. and Comp. Eng., NPS, Monterey, CA, USA, winter 2020.

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California