A NOVEL LOW-COST AUTONOMOUS 3D LIDAR SYSTEM

By

Ryker L. Dial, B.S.

A Thesis Submitted in Partial Fulfillment of the Requirements

for the Degree of

Master of Science

in

Electrical Engineering

University of Alaska Fairbanks

May 2018

APPROVED:

Seta Bogosyan, Committee Co-Chair
Michael Hatfield, Committee Co-Chair
Orion Lawlor, Committee Member
Charlie Mayer, Chair
Department of Electrical and Computer Engineering
Doug Goering, Dean
College of Engineering and Mines
Michael Castellini,
Dean of the Graduate School

Abstract

To aid in humanity's efforts to colonize alien worlds, NASA's Robotic Mining Competition pits universities against one another to design autonomous mining robots that can extract the materials necessary for producing oxygen, water, fuel, and infrastructure. To mine autonomously on the uneven terrain, the robot must be able to produce a 3D map of its surroundings and navigate around obstacles. However, sensors that can be used for 3D mapping are typically expensive, have high computational requirements, and/or are designed primarily for indoor use. This thesis describes the creation of a novel low-cost 3D mapping system utilizing a pair of rotating LIDAR sensors, attached to a mobile testing platform. Also, the use of this system for 3D obstacle detection and navigation is shown. Finally, the use of deep learning to improve the scanning efficiency of the sensors is investigated.

Table of Contents

List of Figures

Page

List of Tables

Acknowledgments

I would like to thank my friends and family for their support, my thesis committee members Dr. Bogosyan, Dr. Lawlor, and Dr. Hatfield, as well as other professors at UAF for their help and guidance, the NSF-Funded CyberAlaska grant for providing funding for this thesis, and NASA and Caterpillar for hosting the Robotic Mining Competition, through which I have gained valuable robotics and engineering experience.

Chapter 1 Introduction

1.1 Purpose

Since classical times, people have dreamt of creating automata to perform tasks that are too menial or dangerous for humans. The best known creator of such devices in antiquity is Hero of Alexandria, a Greek engineer who invented mechanical wonders such as animated statues, automatic doors, and vending machines [1]. Unsurprisingly, as time has marched onward so has the complexity of these automata. More recently, the use of autonomous robotic systems in factories has allowed mass production of goods on a grand scale, which has dramatically transformed the human condition. Typically, factory robots operate under a strict set of constraints in a highly controlled environment, only performing a few defined functions. As the field of robotics expands to automate more and more menial and dangerous tasks, however, many robotic systems need to be able to operate in and maneuver through real-world environments, which are much more dynamic and unpredictable. This necessitates the development of robotic systems with ever-increasing intelligence and capabilities.

To assist in the human colonization of alien worlds, NASA, in cooperation with companies like Caterpillar, hosts the NASA Robotic Mining Competition every summer at the Kennedy Space Center in Florida. The goal of this competition is to develop robots that can autonomously mine resources such as dust and ice, as these are prevalent on planets such as Mars, and can be used to build structures and generate water, oxygen, and fuel to support human colonization. As such, these mining robots must be able to operate in outdoor environments with uneven terrain, with obstacles such as craters and rocks necessitating the use of 3D obstacle detection. The purpose of this thesis was to develop a low-cost 3D mapping system to be used on UAF's mining robot for this competition; this system shall be henceforth referred to as the VS-LIDAR system, for variable scanning LIDAR.

1.2 3D Sensors

1.2.1 Cameras

Most autonomous mobile robots rely on some sort of vision to navigate through the world and complete their assigned tasks, and there are many different methods by which they obtain this vision. Perhaps the most familiar would be through the use of cameras. Equipped with a pair of cameras set up in a stereo configuration, a robot can determine the distances to objects in its environment, working on much the same principle as human eyes do [2]. By comparing differences in camera frames while the robot is moving, a single camera can also be used to estimate the motion of the robot and the 3D structure of its environment [3]. The relatively low cost of cameras makes them a popular choice for vision, but they suffer from several drawbacks. They are sensitive to changes in lighting, require careful calibration, and the 3D data they produce is done at quite the computational expense, while being noisy and error-prone [4].

1.2.2 RGB-D Cameras

Because of these limitations for cameras, many low-cost indoor robots utilize devices known as RGB-D cameras, which provide the depth (distance) to points in the scene, in addition to a color image. These began to see widespread use in November 2010 after Microsoft released the Kinect, a peripheral designed for their Xbox 360 gaming console. The Kinect, in addition to being equipped with a standard RGB camera, contains an infrared projector and an infrared camera. By projecting an infrared pattern onto the environment and detecting distortions in the pattern with the camera, the Kinect is able to determine distances to points in its field of view; this method of measuring depth is referred to as structured-light. The fact that it was mass-produced for a gaming console means that it is cheaply and widely available. It quickly became popular for use on low-cost indoor robots, such as the popular prototyping platforms Turtlebot 1 and Turtlebot 2 [5]. Nowadays, the Kinect can be obtained from second-hand stores for as low as \$5. Many researchers have used the Kinect and similar structured-light cameras for low-cost indoor mapping and autonomy, such as [6], [7], and [8]. Other RGB-D sensors, such as the second version of the Kinect (Kinect v2), use time-of-flight measurements instead of projecting a pattern, meaning that they measure depth by emitting pulses of light and measuring the time they take to bounce off of objects and return, which is then converted into distance using the speed of light. The precise, high resolution timing circuitry required makes these types of RGB-D cameras more expensive than projection based ones, with base models typically costing between \$100 and \$200.

RGB-D sensors are fantastic for indoor robots, as they can quickly obtain dense point clouds (3D scans) with a high update rate ($\geq$ 10 Hz). However, in places with high ambient light, like outdoor areas, their measurements become noisier, degrading performance. This is not as pronounced for time-of-flight cameras (e.g. [9]), but structured-light sensors can fail completely since their projected pattern is sensitive to interference [10].

1.2.3 LIDAR

The most accurate distance sensors utilize light detection and ranging, or LIDAR, which uses pulses of laser light to illuminate a target, then uses the reflected pulse to measure distance. LIDAR sensors can be broadly divided into two categories: scanning LIDARs and scannerless LIDARs [11]. For scanning LIDARs, the light from a laser is used to measure the distance to a single target point, and the laser light is steered and/or the laser is moved to capture an entire scene, with multiple lasers often being used to increase the capture speed. Scanning LIDARs can be configured to capture a scene with the viewpoint of a regular camera, a 2D planar slice around the LIDAR, or a 3D cloud around the LIDAR. For scannerless LIDARs, the light from a single laser is diverged so that it illuminates the entire scene at once, with the reflections being detected by a 2D photodetector array. Time-of-flight cameras fall into the scannerless LIDAR category. Because the laser is diverged, considerably reducing the energy in each individual reflection, scannerless LIDARs are typically more sensitive to ambient light than scanning LIDARs. However, scannerless LIDARs have the advantage of having no moving parts and generate much more data than scanning LIDARs: for example, the Kinect v2 has a data rate of 6.5 million points per second while the Velodyne Puck, a popular LIDAR from Velodyne, used on everything from indoor robots to self-driving cars, only produces 300,000 points per second. But this high data rate can be a double-edged sword, as processing the data requires significant bandwidth and computational resources, and most of this data will likely be filtered out anyway because of redundancy. For example, the original Kinect, with a comparatively smaller data rate of 2.3 million points per second, practically monopolizes the USB 2.0 bus on our mining robot's control laptop.

Most scanning LIDARs used for ground robots use one of three methods of determining distance. The first method measures the time-of-flight directly using timing circuitry, whereas the second method measures the time-of-flight indirectly by measuring the phase difference between the emitted laser pulse and the reflected pulse. The third method is triangulation. In this method, the LIDAR has a laser emitter and an imaging sensor separated by a known distance. When the emitter emits a pulse of laser light, it is reflected off the target and hits the imaging sensor at a measurable angle. This angle and the separation between the two devices is used calculate the distance to the target, with the maximum range of the LIDAR being determined by the separation. In general, LIDARs which measure the time-of-flight directly are more expensive than those that measure it indirectly with the phase difference, with both being more expensive than triangulation LIDARs. Time-of-flight LIDARs also have much further ranges than triangulation LIDARs, on the order of 100 meters for ground-based systems and several kilometers for aerial surveyors, compared to just a few meters for triangulation. However, triangulation based systems are typically more accurate than time-of-flight systems at close range due to the extremely short light travel times involved, but their accuracy does fall off fast as distance increases [12]. It is also worth noting that all LIDARs, because they use reflected laser light, have trouble with targets that are transparent, letting laser light pass through, or are black, absorbing the laser light.

1.3 Overview of Work and Contributions

This thesis describes the creation of a low-cost 3D mapping system that can be used by a mobile robot for autonomous navigation and obstacle avoidance. It utilizes a pair of rotating 2D LIDAR sensors, which work together to increase visual coverage and to compensate for a reduction in the system's update rate caused by this rotation. The rotation parameters of the LIDARs can be reconfigured during runtime to provide denser or sparser scans, or to target a specific area.

Scanning LIDARs are used over other sensors because of the low computational requirements of processing their data, the accuracy of their data, and their ability to operate in a wider range of environments due to being able to work in the dark as well as in the light. Custom hardware and software was made to interface with and control the LIDARs, and this LIDAR driver was made fully compatible with Robot Operating System (ROS) so that the VS-LIDAR is essentially "plug-and-play" with a large variety of existing robotic systems. The closest equivalent low-cost 3D LIDAR system commercially available was the Scanse Sweep 3D scanner kit [13], which also obtains 3D scans by rotating a 2D LIDAR. This scanner has a 40 meter range, captures 1000 data points per second, and costs $700. In comparison, while the VS-LIDAR only has a 6 meter range, it captures almost quadruple the amount of data at 3700 points per second, while costing half as much at $350. Additionally, this 3D scanner kit is no longer available for purchase.

The VS-LIDAR was attached to a mobile robotic base and its use for obstacle detection and navigation was demonstrated using existing navigation packages within ROS. To improve navigation, the robot's wheel odometry, which tracks position, orientation, and velocity, was improved by fusing it with data from an inertial measurement unit using an Extended Kalman Filter. This improvement also allows data collected from the LIDARs to be more consistent while the robot is moving around. However, without correction, error in this improved odometry would still grow without bound. Thus, the system also includes a camera, which it uses to detect ArUco markers, which are a computationally efficient means to determine the absolute location and orientation of the robot and to identify landmarks.

To summarize the specific contributions of this work:

- The creation of a novel 3D LIDAR system with higher obstacle avoidance capabilities and lower cost that other LIDARs in the same price bracket. This was achieved by:
  - The use of a two rotating 2D LIDAR system to obtain 3D scans and compensate for the scanning deficiencies of each LIDAR individually.
  - The design and creation of a frame and 3D printed components which place the LIDARs in an optimal scanning configuration.
  - The design and creation of custom interface boards, custom microcontroller firmware, and a custom set of software packages which allow the VS-LIDAR system to be plug-and-play with the ROS development environment.
- Evaluation experiments performed to demonstrate the improved capability of the two LIDAR system to navigate in environments with unknown static and dynamic obstacles, illustrating the practicality of the system:

  - Experiment results illustrating the novel LIDAR system to be more capable for close range obstacle detection and for operation in sunlit environments than other low-cost 3D sensors.
  - An example implementation of autonomous navigation utilizing pre-existing, standard, open-source ROS software packages, which illustrates the VS-LIDAR system being used for 3D obstacle avoidance and path planning in a real-world environment, handling both static and dynamic obstacles.

- The investigation of a novel deep reinforcement learning based method of increasing the scanning efficiency of rotating 2D LIDARs, referred to as "adaptive scan dithering" in this thesis.

The software, firmware, and hardware developed in support of this thesis have been made open-source and are publicly available at the following URLs:

- Software packages, board schematics, CAD files: *https://github.com/rykerDial/vslidar*
- Microcontroller firmware: *https://os.mbed.com/users/rldial/code/xv11_lidar_controller/*

1.4 Multi-LIDAR and Rotating LIDAR Systems

Many autonomous robotic systems utilize multiple LIDAR sensors to map their surroundings, with perhaps the most famous of which being self-driving cars. These multi-LIDAR systems made their big debut in 2005 in the DARPA Grand Challenge, and in 2007 in the DARPA Urban Challenge. One multi-LIDAR self-driving car entered in the Urban Challenge consisted of six 2D LIDAR sensors, three of which were rotated to obtain 3D data [14]. Another vehicle with multiple LIDARs, which actually won the Urban Challenge, was equipped with a newly released 3D LIDAR, the Velodyne HDL-64 [15]. In fact, five of the six vehicles that finished the challenge were equipped with this LIDAR, which led Velodyne down the path to becoming the leading LIDAR manufacturer.

But with LIDAR sensors being so expensive, a big push is being made by researchers and manufacturers alike to develop cheaper sensors and find more efficient means of sensing. In [16], citing the prohibitively high cost of 3D LIDARs, the authors test out high-speed 3D obstacle detection for a self-driving car using an array of five 2D LIDARs. These LIDARS are kept at a static angle to provide planar scans of the road at short, medium, and long ranges. The combined cost of the five LIDARs is about $40,000, certainly an improvement over the typical cost of LIDAR systems for self-driving cars in use at the time, but still quite expensive.

Of course, the use of LIDAR sensors is spreading not only in self-driving cars, but in all manner of mobile robots, both grounded and flying. Described in [17] is an indoor 3D mapping system using an orthogonal pair of 2D LIDARs. One LIDAR is mounted horizontally, and is used to estimate the motion and determine the location of the robot, while the other is mounted vertically, with the motion estimate being used to stitch the vertical scans together into a 3D map as the robot drives around. The system was designed to be driven by a human operator, but if it was made autonomous this LIDAR setup would not allow it to perform 3D obstacle detection; this, coupled with the fact that it relies on features like walls for motion estimation and localization, would restrict its autonomous operation to more mundane indoor environments. In terms of price, the two LIDAR sensors alone cost over \$18,000. A similar method of mapping buildings with a pair of orthogonal 2D LIDARs is used by Google's Cartographer backpack [18].

In [19], researchers describe the creation of a robot capable of fetching beer from a refrigerator. This system performs 3D obstacle detection by tilting a 2D LIDAR, but since the update rate is slow the data is supplemented with point clouds obtained from a stereo camera setup. The tilting LIDAR has a slow update rate and a large field-of-view, while the stereo camera has a high update rate but a slow field-of-view, so the tilting LIDAR is used to identify "macro structures" in the environment such as people and the refrigerator, which is then used to turn the camera towards points of interest for denser, faster scans. Of course, this system is only meant to be operated in indoor, illuminated environments, with the kinds of obstacles you would typically encounter in an office setting.

[20] describes a system similar to the one in this thesis, which uses a rotating 2D LIDAR to obtain 3D maps. It is capable of mapping both indoors and outdoors. However, this system only performs 3D mapping when the robot is stationary, keeping the LIDAR horizontal while driving for 2D obstacle detection. This work did not investigate using the system autonomously at all, instead having it remotely operated with the help of cameras, and no follow-up work could be found, so it is unclear whether or not they would have stuck to this configuration.

1.5 Thesis Organization

The rest of this work is organized as follows. Chapter 2 details the hardware used in the VS-LIDAR, discussing the individual components used as well as the full system assembly. Chapter 3 goes over the software used to control the LIDARs and allow the robot to navigate through its environment. Afterwards, Chapter 4 describes several experiments performed to evaluate the performance of the system. Chapter 5 describes an attempt to improve the scanning efficiency of the system using deep learning. Finally, Chapter 6 discusses the conclusions of this work, opportunities for future work, and lessons learned.

Chapter 2 Hardware

2.1 Overview

The hardware layout for the VS-LIDAR system is fairly straightforward; a simple system diagram is shown in Figure 2.1. 3D scans of the environment are obtained by panning the 2D Revo Laser Distance Sensor with a Dynamixel AX12A servo. This distance sensor and servo form a 3D LIDAR module, and the system uses two of these modules. These 3D LIDAR modules are controlled by an STM32F767ZI microcontroller unit (MCU) via a simple plug-and-play interface provided by custom-built LIDAR interface boards. Also attached to the microcontroller is the BNO055 inertial measurement unit (IMU), which provides useful data on the motion of the system. The microcontroller sends data from the servos, LIDARs, and IMU to an Acer C720 Chromebook computer, which processes this data and uses it to control the behavior of the system. Also connected to the laptop is the Genius webcam, which is used to detect ArUco markers for absolute position and orientation detection. Additionally, the VS-LIDAR system is mounted to the Create 2 mobile robotic platform, which allows the system to explore its environment. In the coming sections, each of these components is described in greater detail, and the assembly of these components into the system as whole is illustrated.

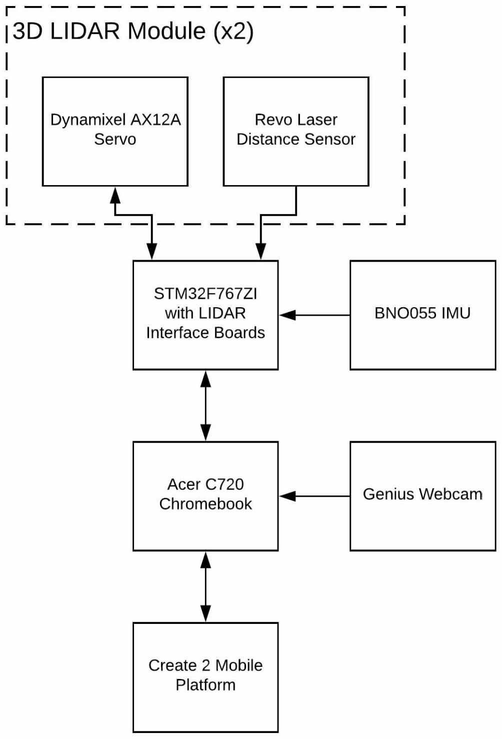


Figure 2.1: VS-LIDAR System Hardware Layout

2.2 Components

2.2.1 Revo Laser Distance Sensor

The Revo Laser Distance Sensor (LDS) is a low cost LIDAR described in the 2008 paper of Konolige et al. [21], and is used by the robot to perceive the structure of its environment. A labeled picture of this device is shown in Figure 2.2. This sensor is more commonly known as the Neato XV-11 LIDAR, as it made its debut on the mass consumer market in 2010 on the XV-11 robotic vacuum cleaner from Neato Robotics, where it was used by the robot to map out its environment for more efficient cleaning. This LIDAR module can be purchased separately from third party sellers typically for about $100, making it perhaps the cheapest 2D scanning LIDAR available, and the relatively low cost and decent capabilities of this device has made it a popular choice amongst robotics hobbyists and researchers, which sparked a large community effort to reverse engineer the device. The main site describing the control protocol of the device, along with other important properties, is the XV11 Hacking Wiki [22]. Also, for those interested, the significant properties of the laser itself are listed in [23, pp. 51].

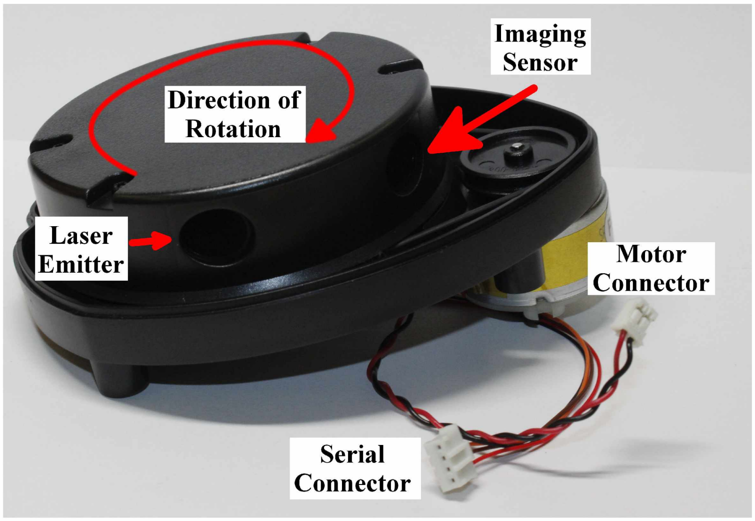


Figure 2.2: Revo LIDAR

This LIDAR consists of a laser emitter and an imaging device, which are spun around by a belt connected to a motor, and distances to objects are determined using triangulation. As mentioned previously, to determine the distance the emitter first sends out a pulse of laser light, which reflects off an object and then hits the imaging sensor at a certain measured angle. The angle, along with the known positions of the emitter and imaging device, are then used to calculate the distance to the object. This process is repeated in one-degree increments along the rotation of the LIDAR, so one full rotation produces 360 distance measurements. Note that the device only rotates in one plane, so the sensor only obtains a planar slice of its environment. The sensor supports a maximum rotation rate of 320 RPM, so the device can make a maximum of 1920 distance measurements per second. Using triangulation limits the range of this sensor to be between 0.2 m and 6 m.

Sensor input and output is accomplished with six wires. Two of these are to power the motor, and the other four are used to power and communicate with an internal microcontroller. Communication with the microcontroller is done over a full-duplex serial connection operating at 115,200 baud, full duplex meaning that it supports both serial transmission (TX) and reception (RX) and has one line dedicated for each. The serial configuration is 8N1, meaning for every byte of data their are 8 data bits, no parity bits, and one stop bit. The microcontroller controls the laser emitter and imaging sensor, measuring distances as well as the intensity of the returned signals. This data is compiled into packets which are then transmitted over the serial line. Each packet contains four pairs of distance/intensity measurements, the angle of rotation corresponding to the first measurement pair, the motor's RPM, and two cyclic redundancy check (CRC) bytes which allow the packet to be checked for errors. A number of commands can also be transmitted to the microcontroller, such as commands to report the firmware version or calibrate the device.

2.2.2 Dynamixel AX-12A Smart Serial Servo

The Dynamixel AX12A is a "smart" digital servo sold by Robotis, and they are used to pan the LIDARs so that the robot can perceive its environment in three dimensions. These servos are designed for and commonly used by robotic systems that require precision positional control. Typical hobby-grade analog servos, which are controlled via simple modulated electrical signals, usually have a rated range of motion of 0-180°. However, in practice these analog servos rarely achieve their full theoretical range of motion, and for most the feedback detailing the actual position of the servo is not exposed to the user, meaning that one cannot know for sure what the actual angle is. For panning the LIDARs, this uncertainty in the actual angle is unacceptable because it would result in a significant error in fusing the laser scans together into point clouds, and this inaccuracy would have a negative impact on the robot's vision-based navigation.

What makes Dynamixel servos smart is that they have sophisticated on-board circuitry that controls the servo. This circuitry can set the position of the servo to anywhere from 0-300° with a resolution of 0.29°, allowing for much finer control over traditional servos. The digital interface, a half-duplex 8N1 serial bus operating at 1,000,000 baud, allows users to interact with the servo using a simple API [24], and also allows multiple servos to be controlled using a single wired connection. In addition to setting the position of the servo, this interface provides feedback on not only the actual position, but also on the temperature, the supplied voltage, and the current load on the servo. Further, the user can set limits on torque and the acceptable error between actual and ideal position, change the clockwise and counterclockwise limits of rotation, and control the rotation speed of the servo with a max speed of 114 RPM and a resolution of 0.111 RPM. The fine control on rotation limits and speed is particularly useful for ensuring a precise panning profile for the LIDARs. These feature-packed servos are ideal for applications that require fine positional and operational control, and this combined with their relatively low price of $45 makes them an easily justifiable replacement for analog servos.

2.2.3 Bosch BNO055 Inertial Measurement Unit

The BNO055 is a low cost inertial measurement unit (IMU) containing an accelerometer, a gyroscope, and a magnetometer, which is described in [25]. The accelerometer provides linear accelerations along the x, y, and z axes of the device, the gyroscope provides angular velocities about these axes, and the magnetometer provides a 3D compass bearing relative to the magnetic north pole. It also contains a dedicated microcontroller, which not only calibrates the three sensors during runtime, but also combines the sensor data into orientation estimates using proprietary algorithms. These orientation estimates can be reported as either Euler angles (roll, pitch, and yaw) or as quaternions; the quaternion output is more accurate, so it is used in the VS-LIDAR system. Additionally, the board produces two additional readings with the accelerometer data: one that is the linear accelerations with acceleration due to gravity removed, and one that is just the gravitational accelerations. The IMU is a very useful source of information when trying to estimate the robot's motion and orientation, so it lends itself well to robot localization.

Communication with the board is accomplished with an I2C interface, and the device is configurable for data fusion modes that both include and exclude the magnetometer. The magnetometer requires much more calibration to be accurate than the accelerometer and the gyroscope, and just as the readings of a traditional compass can be skewed by ferrous metals and sources of electricity, so too can the readings of the magnetometer. For these reasons, in the VS-LIDAR system the magnetometer is disabled, and the IMU performs its data fusion and processing using only the gyroscope and accelerometer data.

2.2.4 STM32F767ZI Microcontroller and LIDAR Interface Boards

The STM32F767ZI is a high performance ARM Cortex-M7 based microcontroller from STMicroelectronics [26]. This board interfaces with the LIDARs, servos, and the IMU, and facilitates the compilation and transfer of sensor data to the robot's primary computer. It sports a 216 MHz CPU and a host of other features that make it powerful enough to run the VS-LIDAR system. The LIDAR boards shown in Figure 2.3 were designed to allow for an easy, plug-and-play interface between the STM32F767ZI and the scanning LIDAR system. The larger of the two boards is the LIDAR Main Board, which plugs into the microcontroller as shown. A circuit schematic for the main board is shown in Figure 2.4. On the left side of the board is power circuitry. 9-12 V is fed into the circuit via a barrel jack, which is used to power the servos and is also stepped down to 5

V to power the LIDARs. A switching regulator is used to step down the input voltage to 5 V, both reducing the power consumption of the board as well as the heat generated (in a prototype using a simple linear regulator, the regulator was getting so hot that it required a heatsink). In the middle of the board are connectors for the servos, as well as an octal line buffer to arbitrate the half-duplex servo serial bus. On the right side of the board are two RJ-45 ports which are used to connect to the LIDAR Interface Boards.

The smaller of the two boards shown in Figure 2.3 is the LIDAR Interface Board, and there is one of these boards per LIDAR. A circuit schematic for these boards is shown in Figure 2.5. Each board has ports for the LIDAR's serial and motor connectors, along with a simple circuit allowing the speed of the LIDAR's motor to be controlled by a pulse-width modulated (PWM) signal. This circuit consists of a field-effect transistor (FET) which is switched on and off by the PWM signal, allowing enough current to flow through the motor to achieve the desired speed. There is also a diode placed across the motor terminals to protect the main board circuitry from any electricity generated by the motor, as well as a resistor pulling the gate of the FET to ground to ensure that the FET is off when the PWM signal is low. This circuitry is used by a proportional-integral (PI) controller to keep the motor's RPM near a target level; more on this controller in Section 3.4.

To interface with each LIDAR, three dedicated wires are needed: two for serial TX and RX, and one for the PWM signal. Additionally, the LIDARs need connections to 5 V and ground for power; these lines can be shared between LIDARs. Thus, the eight lines of the RJ-45 bus are exactly enough to support two LIDARs. Jumper pads on the LIDAR interface boards are used to select which lines are used for TX/RX and the PWM signal, so only one type of interface board needs to be manufactured.

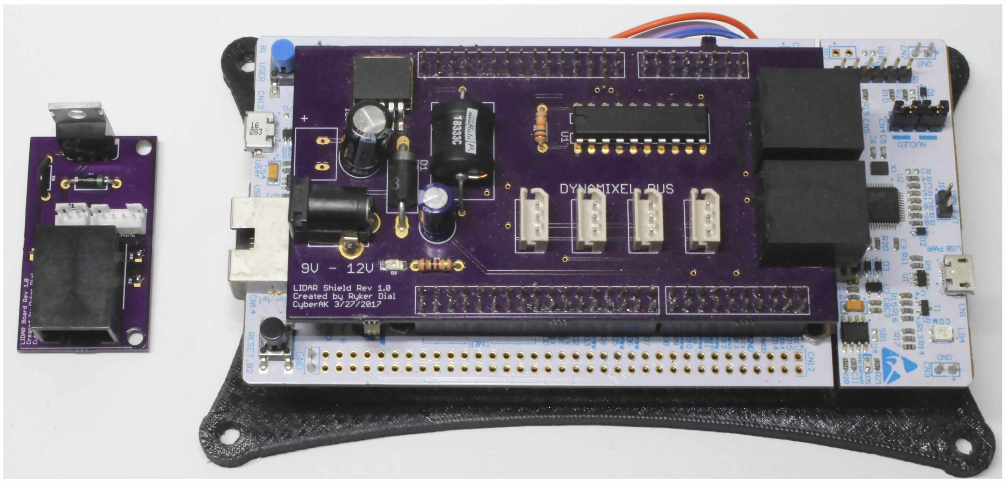


Figure 2.3: LIDAR Interface Boards

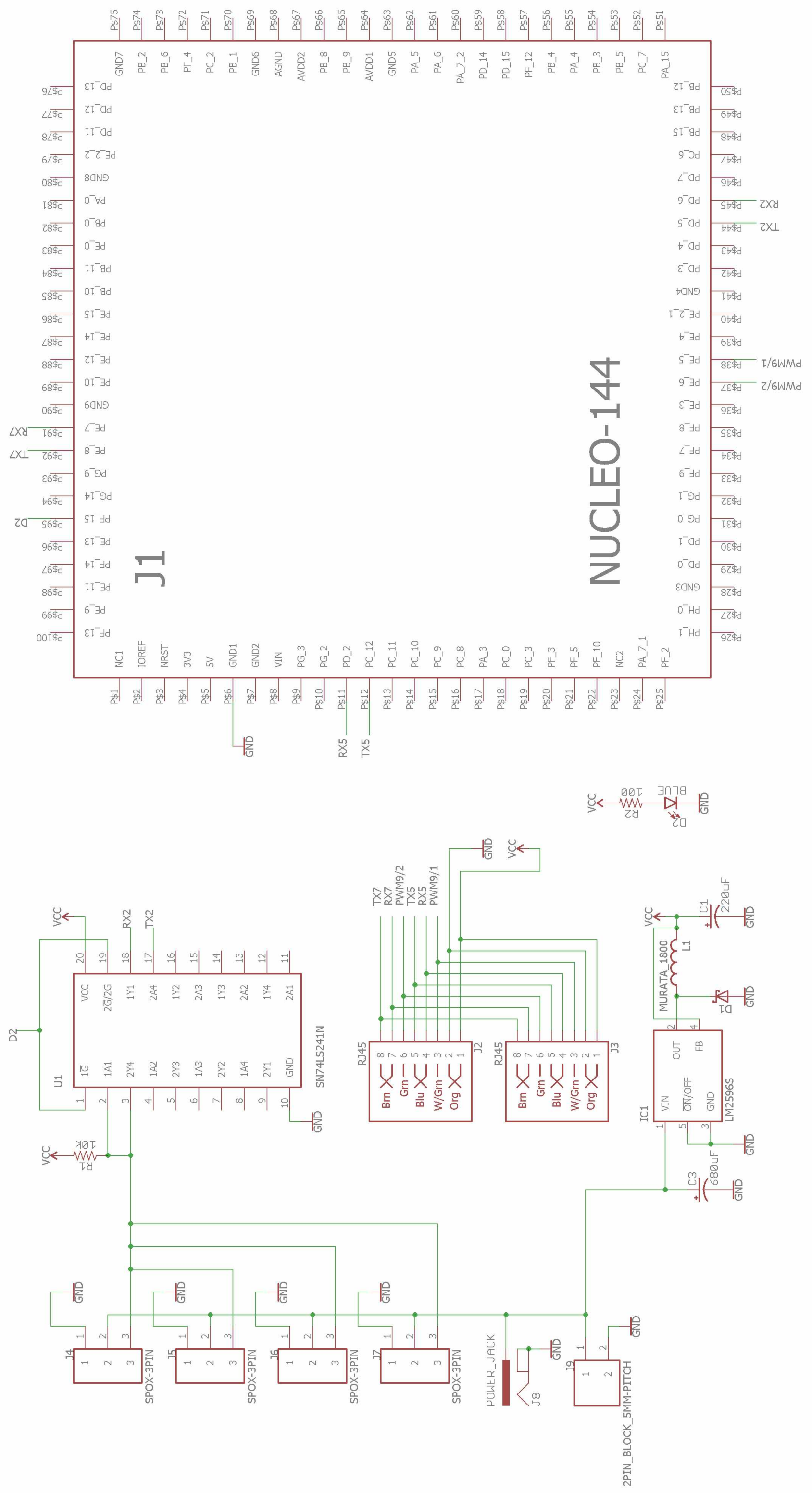


Figure 2.4: Circuit Schematic for LIDAR Main Board

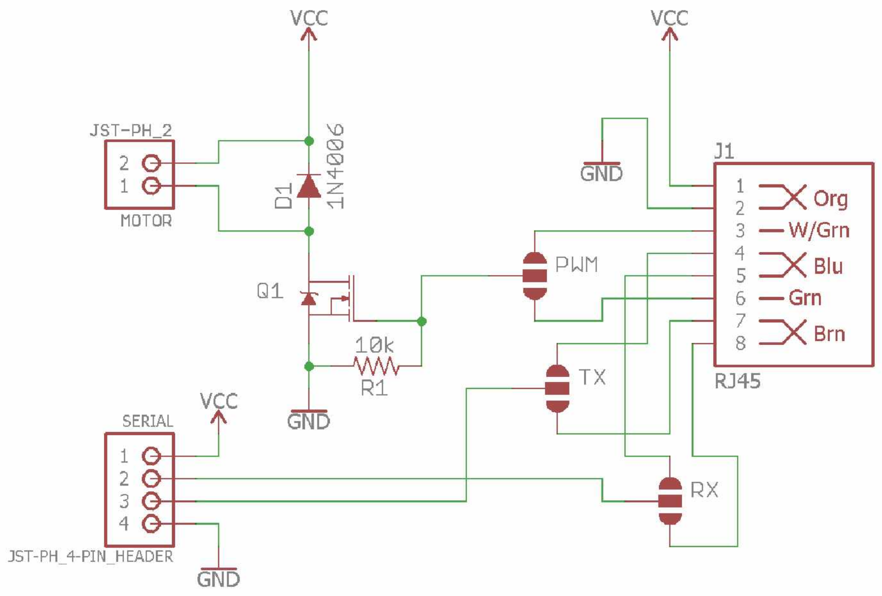


Figure 2.5: Circuit Schematic for LIDAR Interface Board

2.2.5 Create 2 Programmable Mobile Robotic Platform

The Create 2 is a low cost mobile robotic platform sold by iRobot for use by STEM educators and by robotics hobbyists and researchers. Interfacing with the platform is done with the Create 2 Open Interface Specification [27], and the system has a host of sensors that are useful for robot autonomy. The platform has two independently driven wheels allowing it to move using differential drive, a common drive style for mobile robots where the velocities of the two wheels are controlled to allow the robot to either move forward, backward, or turn with a certain radius. This turning radius can be zero, allowing the robot to turn in place about the center of its two driven wheels. Encoders track the movement of these wheels, and this motion is integrated into odometry that estimates the robot's trajectory. This tracking is very useful for robot localization, but cannot be relied on as the sole source of localization information, as the error in the odometry will drift without bound over time unless corrected by some absolute reference (e.g., with regards to a map). Sources of error include wheel slippage and limited resolution of motion integration [28, pp. 269-270]. While the Create 2 is a very useful platform for testing the VS-LIDAR system's surveying and exploration capabilities, it should be noted that the system can be used on any mobile ground robot which provides wheel odometry and is big enough to support its weight and inertial forces.

2.2.6 Acer C720 Chromebook and Genius Webcam

The computer chosen to run the robot's control software is the Acer C720-3404 Chromebook. It is equipped with an Intel Core i3-4005 CPU clocked at 1.7 GHz, 4 GB of RAM, and a 32 GB solid-state storage drive. Instead of the stock Chrome OS operating system it is running Ubuntu 14.04, a popular Linux distribution with a host of available tools that make software development easier. These were chosen because they are fairly powerful relative to their cheap $279.99 price tag, and they have the additional benefit of having a small form factor, making them easier to incorporate in robotics projects. The webcam used for ArUco marker detection is a Genius webcam, which sports a wide-angle field-of-view and a resolution of 1080p, and was chosen because of its decent specs and the fact there were a lot left over from a previous robotics project. This particular camera costs $49.99, but it could easily be replaced by a cheaper model.

2.3 System Assembly

2.3.1 3D LIDAR Module

Figure 2.6 shows how the Revo LDS, Dynamixel servo, and LIDAR interface board are assembled into a 3D LIDAR module using 3D-printed brackets. The Revo LDS is placed at a 90° angle relative to the servo such that the axis of rotation for the servo is orthogonal to the axis of rotation for the laser scans. This configuration enables and maximizes the 3D coverage of the LIDAR sensor.

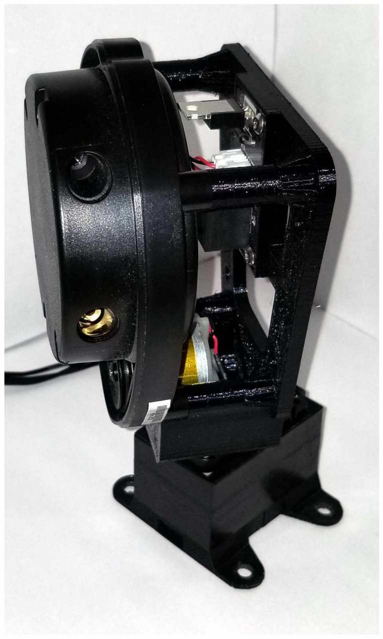


Figure 2.6: 3D LIDAR Assembly

2.3.2 Full Assembly

An annotated picture of the fully-assembled VS-LIDAR system is shown in Figure 2.7. The system has two 3D LIDAR components: the topmost one pans in a way that primarily maps the region in front of the robot, and the bottommost one pans in a way that primarily maps the regions to the left and right of the robot. Additionally, the bottom LIDAR is oriented at a 45° angle from the horizontal plane so that its center of rotation points towards the ground directly in front of the robot. The LIDAR provides real-time mapping coverage of a small region about its center of rotation, so this placement allows the robot to always have some idea of what is directly in front of it.

The two LIDARs, along with the IMU, are held together with a frame made of steel plate, aluminum tubing, and 3D printed brackets. This frame is itself rigidly attached to platforms mounted on the Create 2 mobile base. These rigid and static placements allow for more reliable fusion of the LIDAR, IMU, and wheel odometry data: the static placement means the system does not have to track changes between the coordinate frames of the various sensors, and the rigidity means the system does not need to worry as much about mechanical stresses on the system skewing the relationships between the frames. Also noteworthy is that the IMU is placed on the Create 2's center of rotation. This allows the IMU to more accurately track the rotation of the robot, as the angular velocities and orientation estimates provided do not have to be transformed to account for an offset between the IMU and the center of rotation. Table 2.1 gives a breakdown of the cost of the VS-LIDAR system, and shows both the cost of just the 3D LIDARs (LIDARs, servos, and control boards) and the total cost of the VS-LIDAR test platform. Note, this does not include the costs of 3D printed components or frame construction materials, as these are bound to vary.

Table 2.1: Cost of Components in VS-LIDAR Test Platform

Component	**Cost**
Revo LDS (x2)	~$200.00
Dynamixel Servo (x2)	$ 90.00
BNO055	$ 34.95
STM32F767zi Board	$ 22.54
LIDAR Interface Boards	~$ 40.00
Create 2	$199.99
Acer C720 Chromebook	$279.99
Genius Webcam	$ 49.99
Total (3D LIDAR)	$352.54
Total (All)	$917.46

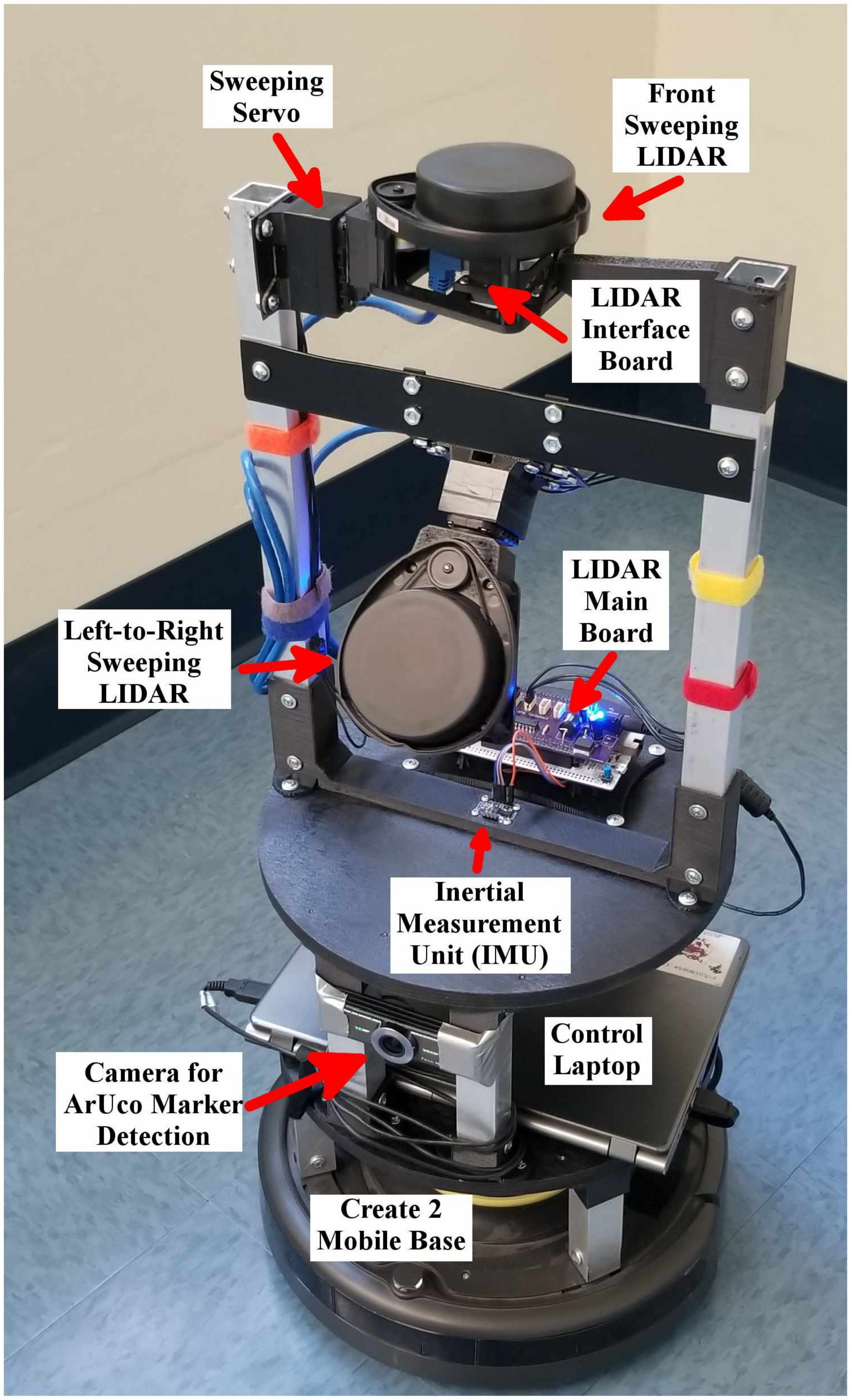


Figure 2.7: LIDAR Test Platform with Significant Features Labeled

Chapter 3 Software

3.1 Robot Operating System

To hasten development time and enable components of the VS-LIDAR system to be easily integrated into other robotics projects, the system makes extensive use of Robot Operating System (ROS) [29]. ROS in an open source software framework designed to facilitate the creation of complex robotic systems by providing general purpose tools and libraries that prevent roboticists from having to "reinvent the wheel." Additionally, ROS defines a set of standards for both robotics software and robotics design in general, that not only result in more robust robots but also allow the developer community to easily share code and apply it to other projects. While community provided libraries and tools for ROS don't typically follow any specific release schedule, ROS's core set of tools and libraries is updated about every year or so to improve various aspects of the software. The ROS distribution corresponding to the Ubuntu 14.04 operating system that the robot's control laptop is using is known as ROS Indigo Igloo. This version was chosen because it is a long term support (LTS) version, meaning that for five years after its initial release it is guaranteed to be updated to fix bugs and any other problems that pop up. Developing the VS-LIDAR software using an LTS version guarantees that it is built using a well-tested and well-supported set of tools, and also increases its utility to the community.

Control software utilizing ROS is split into a set of programs called nodes, where each node implements a distinct aspect of the system's functionality. For example, a system could have one node for getting data from a camera and another for processing that data. Information passing between nodes is accomplished via a publisher/subscriber scheme, with data packed into standardized message formats for easier sharing. The node in the above example collecting the camera data would publish it, making it available to other nodes, and the processing node would subscribe to this data, letting the ROS system know that it wants to receive new data as it is made available. A master node, common to all ROS-based systems, facilitates the transfer of messages between publishers and subscribers. This setup decouples information production from its consumption, meaning nodes are modular and ROS-based systems are easily reconfigurable and adaptable.

3.2 Frames of Reference

Each sensor in the VS-LIDAR system that provides spatial data does so in its own coordinate frame, so understanding and managing the relationships and transformations between each frame is critical for data fusion. Luckily, ROS provides the library tf2 to help with this, and has set some standards for mobile robot coordinate frames [30]. Coordinate frames are organized in a tree, where each node can have any number of child frames but only one parent frame. To aid in visualizing the relationships between frames in the VS-LIDAR system, this tree is shown in Figure 3.1. Starting at the base of the tree are *laser_frame_1* and *laser_frame_2*, the coordinate frames for the 2D laser data. The servo angle is used to rotate these frames into *base_laser_1* and *base_laser_2*, static frames with their origins at the base of the servos, then a fixed offset is applied to transform each individual *base_laser* frame to a single *base_laser* frame, which has its origin at the bottom of the VS-LIDAR mechanical frame (close to the IMU but slightly lower). Tracking the rotation of each *laser_frame* relative to the fixed *base_laser* frame allows the 2D scans to be transformed into 3D point clouds.

The coordinate frames of the camera and the IMU are *usb_cam* and *imu_link*, respectively. These frames, along with the *base_laser frame*, are all transformed relative to a single fixed point on the robot, *base_link*, whose origin is placed coincident with the origin of *imu_link*. Then, these frames are further transformed relative to *base_footprint*, whose origin is coincident with the origin of *base_link*, except it lies in the ground plane at the base of the robot. Finally, the robot's position and orientation in the environment are tracked by the transformation between the *odom* frame and *base_footprint*, with the origin of the *odom* frame essentially being the origin of an odometric map.

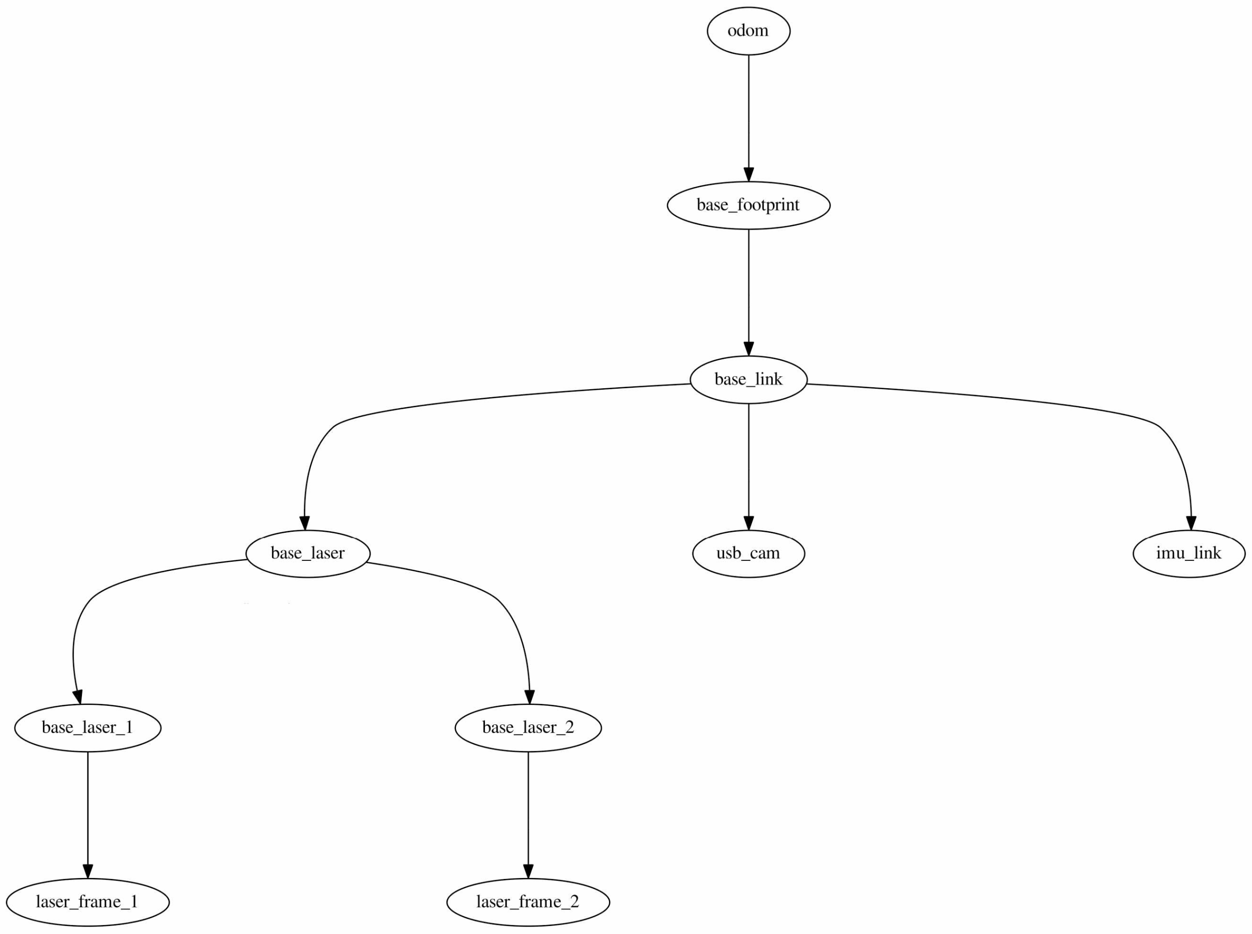


Figure 3.1: VS-LIDAR System Coordinate Frames

3.3 System Overview

The VS-LIDAR system's software packages span across multiple pieces of hardware and operating systems. Figure 3.2 illustrates how the various components interact and are composed to form the system as a whole. On the top layer of this figure is the system's hardware: the Create 2 robot, the mobile base used for testing autonomous navigation; the Genius webcam, which is used to detect an ArUco marker to determine the absolute pose (position and orientation) of the robot; and the microcontroller, which collects data from the IMU, LIDARs, and servos, and controls the panning motion of the servos. A collection of nodes running on the robot's control laptop interface with these devices and implement the main functionality of the VS-LIDAR system:

- **rosserial node**: Preexisting ROS package, receives data from the microcontroller and publishes it to make it available to other nodes. Also sends any reconfigurations to servo panning parameters to the microcontroller.

- **xv11_raw2scan_wtf node**: One per LIDAR, these nodes parse raw LIDAR data packets from the microcontroller and publish it as a standard ROS *LaserScan* message. Also, using the angle of the corresponding servo, they publish a coordinate transform from the corresponding *laser_frame* to the corresponding *base_laser* frame, which allows the LIDAR data to be transformed from 2D to 3D.

- **xv11_raw2imu node**: This node parses raw IMU data and publishes it using the standard *IMU* message.

- **scan2cloud node**: One per LIDAR, these nodes transform the 2D *LaserScan* in the corresponding *laser_frame* into a 3D *PointCloud2* in the *base_footprint* frame. They also filter out any points from within a small bounding box around the robot so that any scans of the chassis are discarded.

- **usb_cam node**: Preexisting ROS package, this node interfaces with the Genius webcam, collecting raw camera data and publishing it for the rest of the system.

- **aruco_mapping node**: Preexisting ROS package, this node detects ArUco markers within the camera's field-of-view and publishes the markers' positions and orientations. This package is capable of detecting multiple markers and making a map with them, but in this system it is used simply for detecting the pose of a single marker.

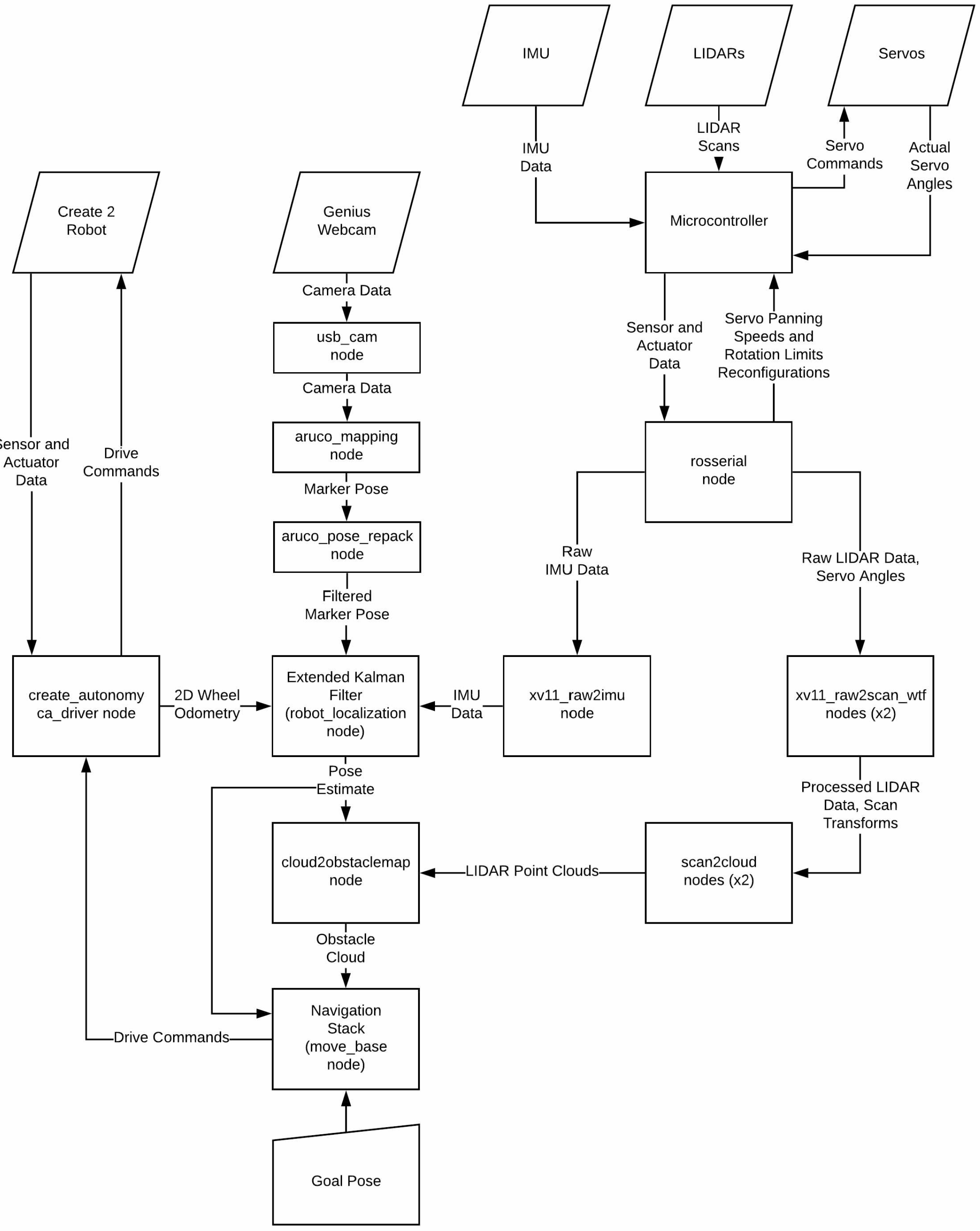


Figure 3.2: VS-LIDAR System Diagram

- **aruco_pose_repack node**: This node subscribes to marker poses from the *aruco_mapping* node. When a marker has been detected, it first checks the ID of the detected marker to guard against false detections. Then, if the robot is close enough to the marker, it corrects the pose to account for the camera's offset from the robot's center of rotation, and then publishes the pose as a *PoseWithCovarianceStamped* message.

- **ca_driver node**: Preexisting ROS package, this node interfaces with the Create 2 mobile base, sending it drive commands and publishing received sensor data, which includes the wheel odometry. As such, this node is subject to change depending on what mobile base the VS-LIDAR is attached to, for example being replaced with a custom driver for UAF's mining robot.

- **robot_localization node**: Preexisting ROS package, this node subscribes to the IMU data, the wheel odometry, and the ArUco marker pose, and fuses them together with an Extended Kalman Filter to produce more accurate odometry.

- **cloud2obstaclemap node**: This node accumulates LIDAR point clouds, using the filtered odometry to account for the robot's movement. At a set rate, it takes these accumulated clouds and filters them so that only obstacles remain. Currently, this is done by setting and minimum and maximum obstacle depth and height, and removing points that are not within one of these two bands, effectively removing the floor and any hanging objects the robot can safely pass under. The node then publishes the obstacle cloud and starts over, deleting the previously accumulated cloud.

- **move_base node**: Preexisting ROS package, this node allows the robot to navigate through its environment using the LIDAR data and the filtered odometry. It constructs maps within the *odom* coordinate frame, which it then uses to plan paths which allow the robot to drive around obstacles to waypoints. These waypoints can be set either by a human user or by a node using the autonomous navigation to complete a specific task.

Other notable nodes that were developed but are not used in the final system:

- **cloud_accumulator node**: Accumulates and then publishes a complete panning cycle's worth of point clouds for the specified LIDAR. This is used to visualize the differences in the point cloud between consecutive panning cycles.

- **cloud_coverage_metrics node**: Computes coverage metrics described in Chapter 5 for a specified LIDAR.

- **adaptive_dither_dqn node**: Implements the neural network described in Chapter 5. Attempts to intelligently dither the rotation speed of the specified LIDAR.

3.4 Microcontroller Firmware

The firmware for the microcontroller was created using ARM's Mbed development environment, which provides a set of libraries to more easily use the microcontroller's hardware as well as implements more complicated functionality such as task scheduling. Like ROS, the Mbed platform also provides tools for community code sharing, facilitating the creation of more complex systems. Additionally, Mbed has a library allowing the microcontroller to communicate with PC-side ROS nodes directly using robust, standardized protocols.

Figure 3.3 shows the process diagram for the microcontroller firmware. Upon system startup, the MCU is configured as a ROS node so that it can communicate with the ROS system on the PC-side. Additionally, the hardware random number generator is initialized so that it can be used to generate random offsets for the servo rotation rates. Next, the IMU is configured, which includes uploading prerecorded calibration data to the IMU so that it can immediately provide accurate orientation estimates without having to first go through its internal calibration routine. This speeds up the system start up, and since the magnetometer is disabled this initial calibration data is still viable if the system is moved to a different location. It should be noted that the IMU will still update this calibration during runtime to further improve the accuracy. In addition to uploading this calibration data, the x, y, and z axes of the IMU's sensors are remapped such that the IMU's x-axis points forward in the *base_footprint* frame, as it is the standard in ROS that in a robot's base coordinate system, the x-axis represents the direction of the robot's forward motion [31]. Since the purpose of the IMU is to measure changes in the robot's motion, it makes sense for the coordinate systems to be aligned in this manner.

The servos are configured next. Communication is established with each servo, their panning parameters (rotation speed and the clockwise and counterclockwise panning limits) are set, and then their angles are initialized to their clockwise panning limits. The left-to-right sweeping LIDAR is initially configured to pan between 105° and 195°, which allows it to sweep 45° to each side of its 150° centerpoint. This provides a good view of the robot's sides, and panning beyond these limits would just result in the LIDAR scans being increasingly obstructed by the robot's chassis while potentially also collecting useless scans of the room's ceiling. The front sweeping LIDAR is set to pan from 145° to 195°, 45° counterclockwise and 5° clockwise of the centerpoint, with the panning range being smaller than that of the left-to-right sweeping LIDAR to allow the area in front of the robot to be scanned more frequently. This counterclockwise limit and bias maximizes the amount of ground in front of the robot scanned without the scans being obstructed by the left-to-right sweeping LIDAR and the chassis, which aids in navigation by allowing the robot to detect obstacles that lie in its path. The small slice of space scanned clockwise of the centerpoint provides coverage of features taller than the robot. Both LIDARs begin their panning at a speed of 7.5 rpm, which was chosen by observing point clouds obtained at different speeds without random

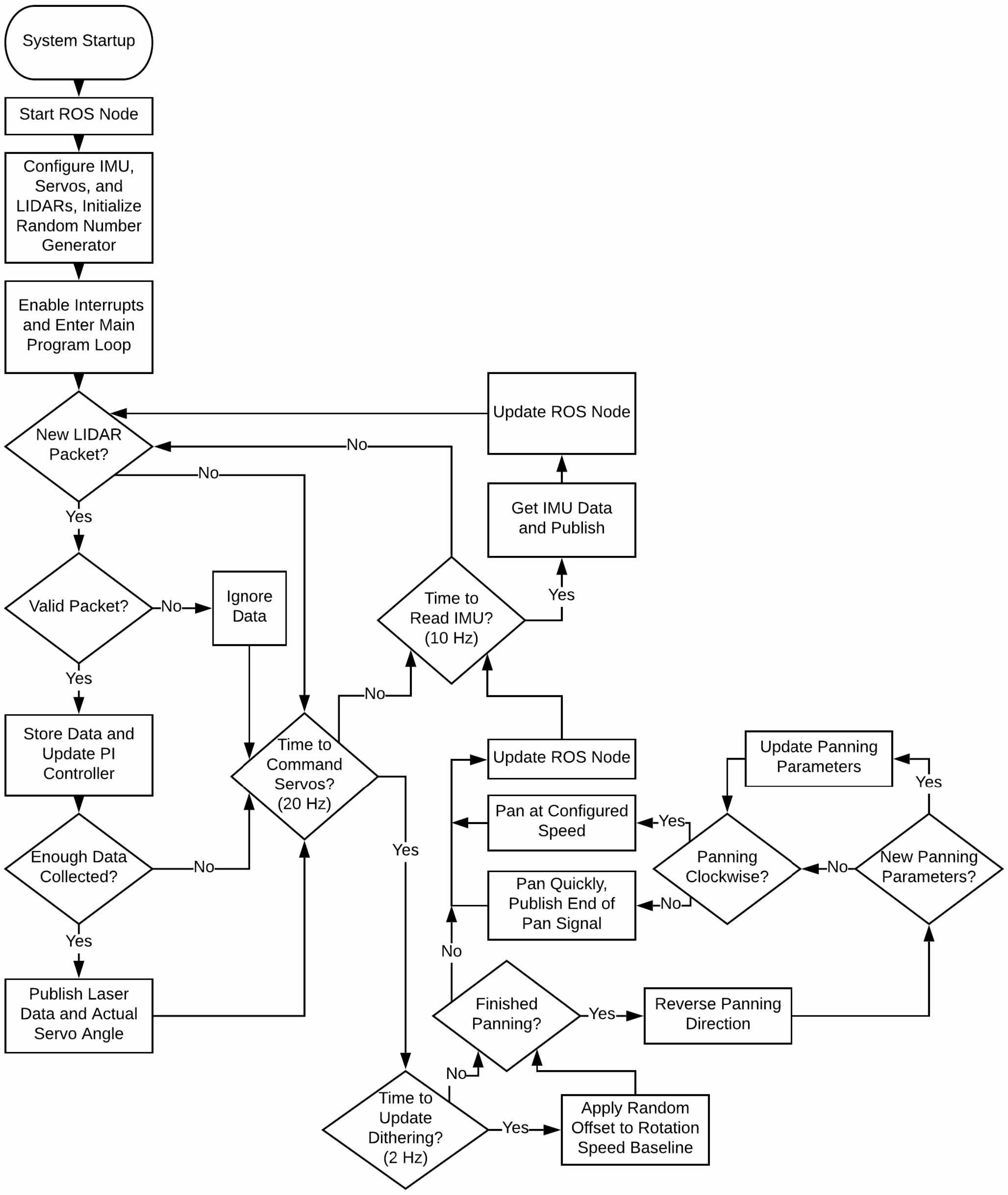

System Startup
Start ROS Node
Configure IMU, Servos, and LIDARs, Initialize Random Number Generator
Enable Interrupts and Enter Main Program Loop
New LIDAR Packet?
Yes
No
Valid Packet?
No
Ignore Data
Yes
Store Data and Update PI Controller
Enough Data Collected?
No
Yes
Publish Laser Data and Actual Servo Angle
Time to Command Servos? (20 Hz)
No
Yes
Time to Read IMU? (10 Hz)
No
Yes
Get IMU Data and Publish
Update ROS Node
Update ROS Node
Pan at Configured Speed
Pan Quickly, Publish End of Pan Signal
Panning Clockwise?
Yes
No
Update Panning Parameters
New Panning Parameters?
Yes
No
Finished Panning?
Yes
No
Reverse Panning Direction
Time to Update Dithering? (2 Hz)
Yes
No
Apply Random Offset to Rotation Speed Baseline

Figure 3.3: Microcontroller Firmware Process Diagram

dithering and choosing a speed which seemed to provide a "good" point density without panning too slow.

Finally, the actual LIDARs are configured, which entails establishing a serial connection with each LIDAR, spinning them up so that they start transmitting data, and starting up the proportional-integral (PI) controller which keeps the LIDAR spinning at its target speed. This PI controller has parameters $K_p = 1.0$ and $T_i = 0.5$, and has a update rate of 500 Hz. It adjusts the PWM signal sent to the speed control circuitry of the LIDAR Interface Board to bring the LIDAR's motor RPM closer to the target speed. The motor can go as fast as 320 RPM before the LIDAR's on-board microcontroller can no longer keep up with the data produced by the sensor, so the target speed in the VS-LIDAR system is set to 310 RPM to provide a bit of a buffer zone. Since each revolution provides 360 distance measurements (one per degree), this results in an average data rate of 1860 distance measurements per second per LIDAR. The motors keep the LIDARs spinning close to the maximum speed so that more data is collected and thus the system can survey the environment more quickly. One familiar with control systems may wonder why a derivative term was not added, why a PI controller was used instead of a proportional-integral-derivative (PID) controller. This is because the derivative term amplifies the noise in the feedback signal, and for this system, the motor RPM measurement was noisy enough that adding a derivative term caused the controller to overshoot more often and with greater magnitude, while providing no clear benefit.

After configuration, the MCU enables interrupts and enters the main program loop. This loop first checks if there is new packet of data available for either LIDAR, and if there is it checks if it is valid by performing the cyclic redundancy check. If it is, it stores the data and uses the reported motor RPM to update the PI controller; otherwise the packet is discarded. The system then checks whether or not enough packets have been collected from that LIDAR to transmit the data to the computer. If so, it publishes the LIDAR data, along with the current angle of the servo. The servo angle will have changed between the first packet of LIDAR data and the last in the set. This is accounted for on the PC side by the *scan_to_cloud* node using the current angle, in conjunction with the angle reported with the previous set of LIDAR data, to estimate the motion and interpolate the points in 3D space; more on this in Section 3.5. Too few packets, and the microcontroller may be reading the servo angles too often, tying up the servo serial bus and interfering with the panning control. Sending packets more frequently also increases the amount of overhead in sending data to the PC. But the more packets collected, the less accurate the motion estimation will become due to variations in the servo's speed. Ultimately, collecting 48 data points (12 packets) before transmitting the data to the PC was chosen as a good balance of these trade-offs. All told, each packet of data transmitted to the PC contains 48 pairs of distance and intensity measurements, the rotational angle of the Revo LDS corresponding to the first pair of measurements, the servo angle corresponding to the last pair of measurements, a timestamp corresponding to the start of this data collection, and, to aid with interpolating the points, the average RPM of the LIDAR's motor while the distance/intensity data was being collected.

After processing any packets from the LIDARs, the system then checks if it time to update the servos, which occurs at set frequency of 20 Hz using a timer. If so, it first checks if it is time to apply a random offset to the rotation rates of the servos, which occurs at set frequency of 2 Hz. The hardware random number generator is used to apply an offset between ± 5 RPM to the 7.5 RPM baseline; this dithering is performed to aid with spreading out points between consecutive panning cycles. Then, the system checks if either servo has reached its clockwise or counterclockwise panning limit, and if so starts panning that servo in the direction of the other limit.

Afterwards, if new panning parameters have been received from the PC-side, the baseline rotation speed and clockwise and counterclockwise limits are updated, provided they are valid. Next, if the servo is now panning clockwise, the system pans at the configured rotation speed, otherwise it pans at a fast speed of 28.5 RPM. Panning to one side quickly instead of having a consistent speed throughout is done because when the servos change direction, the LIDARs start scanning the area that was most recently scanned. It is more desirable to scan areas that were least recently scanned, since it allows the system to detect changes in the environment across the robot's entire field-of-view sooner and with more consistency. Thus, even if the system is panning to one side slowly to obtain a denser cloud, it should still pan to the other side quickly to start scanning areas that were least recently scanned sooner. Starting to pan counterclockwise is also accompanied by publishing an *Empty* message to signal the end of the panning cycle, which is used on the PC-side to compare point clouds from consecutive cycles. Finally, the ROS node is updated, which triggers the process of transmitting published data to the PC and processing any data received from the PC. After controlling the servos, the final step in the main program loop is to read data from the IMU, which it does at a fixed rate of 10 Hz using another timer. This involves obtaining the linear acceleration, angular velocity, and orientation data from the IMU, compiling this data into a timestamped packet, publishing it, and then updating the ROS node once again. Then, the main loop starts anew.

3.5 PC-Side Point Cloud Fusion

Let us now examine in greater detail the process used to transform the 2D LIDAR into 3D point clouds. As mentioned above, two nodes work together to do this, *xv11_raw2scan_wtf* and *scan2cloud*, and there exists a pair for each LIDAR. Upon receiving a raw packet of laser scan data from the *rosserial* node, the *xv11_raw2scan_wtf* node first steps through each distance measurement, and if any lie outside the specified range of the LIDAR (2cm to 6m) it sets that distance and its corresponding intensity measurement to 0. Each pair of measurements is stored in a *LaserScan* message after their values are verified. Afterwards, the average RPM of the LIDAR's motor is used in conjunction with the number of data points collected to estimate the time increment between consecutive measurements, which is used when interpolating the points; this too is stored in the *LaserScan* message. Finally, the timestamp is extracted from the packet and used to stamp the *LaserScan* message with the time that the data collection for that scan started.

The other task that this node performs is to use the servo angle to provide the transform between the specific *laser_frame* and the specific *base_laser*. For brevity, the method of calculating this transform is not discussed here, but it is worth noting that the transform is not timestamped with the same time as the *LaserScan* message. Where the *LaserScan* is timestamped corresponding to the start of the data collection, the transform is timestamped corresponding to the ending of the collection, since that is when the servo angle was read. The ending timestamp is determined by adding to the starting timestamp the time increment in between measurements multiplied by the number of measurements minus one. More accurate timestamps for the transforms improve the accuracy of the point interpolation.

The *scan2cloud* node utilizes the preexisting *laser_geometry* package, which produces a point cloud given a laser scan and a target coordinate frame, and was originally designed for 2D LIDARs on tilting platforms. Using this package is advantageous because instead of just doing a simple transform, it also interpolates the points to account for the rotation of the sensors. To trigger this behavior, it was told to transform the laser data from the specific *laser_frame* to the *base_footprint* frame. This transform branch accounts for the motion of *laser_frame* caused by the servo, but not the motion caused by the robot's driving. It is pointless to do so, since these point clouds are published at an average rate of 38.75 Hz, while the wheel odometry is published at a rate of only 10.2 Hz, so the robot can be considered stationary during the collection of the laser data; regardless, *laser_geometry* isn't capable of accounting for the robot's linear motion anyway. To interpolate the points, *laser_geometry* then looks up two coordinate transforms: the transform from *laser_frame* to *base_footprint* corresponding to the start of the laser data collection, and the same transform corresponding to the end of the collection. Using these transforms, it then performs Slerp, or spherical linear interpolation, on the points. More information on Slerp can be found in [32].

After obtaining the point clouds, the *scan2cloud* node filters out all points that fall within the bounding box of the robot to avoid scanning its own chassis, which is made easy by having the clouds in the *base_footprint* frame. Were this not done, the robot would leave phantom scans of itself behind as it moved around, which could potentially confuse the autonomy. The point clouds are then finally published for use by the rest of the ROS system.

As noted above, these point clouds are then sent to the *cloud_obstacles* node. This node accumulates these clouds, accounting for the motion of the robot by transforming them to be in the *odom* frame. Relying on the *odom* frame like this is necessary, but requires that odometry be extremely accurate over the short term. This could also cause the point clouds to be accumulated incorrectly if the robot is stuck, say, driving into a wall at full power. A solution to this would be the incorporation of a sensor that could quickly tell if the robot is moving or not. The Create 2 has an encoder on its caster wheel, referred to as the stasis sensor, which can tell if the robot is stationary even if the powered wheels are spinning telling it otherwise. However, this sensor wasn't used by the VS-LIDAR system because it is specific to this one mobile platform.

After accumulating clouds for a set amount of time, determined by a set update rate, the node passes the combined cloud through a series of filters. The first filter, a passthrough filter, removes points that fall between a minimum obstacle depth and a minimum obstacle height. This removes driveable areas of the floor/ground from the cloud so the robot's autonomy knows it can drive there. The second filter, also a passthrough filter, removes points that fall outside of a maximum obstacle depth and maximum obstacle height. This removes points on the ceiling or those of a hanging obstacle the robot can safely drive under. The final cloud, containing only obstacles, is then published for use by the navigation stack. The current system uses a minimum and maximum obstacle depth of -0.05 m and -1.0 m, and a minimum and maximum obstacle height of 0.05 m and 1.0 m, respectively. This method of detecting obstacles is quite basic, and doesn't work properly if the robot becomes tilted, as will happen during the robotic mining competition when driving on the uneven terrain. Future versions of this node will use a smarter method of detecting obstacles.

3.6 Localization System

3.6.1 Fusion of Wheel Odometry and IMU Data

To keep consecutive point clouds obtained with the VS-LIDAR consistent with one another while the robot is moving requires that the robot keep careful track of its position and orientation, such that over the short term its odometry is very reliable. If the odometry is too unstable, scans of the same obstacle might wind up in two different places, confusing the robot's autonomous navigation. In initial evaluation of the VS-LIDAR testing platform, it was discovered that while the Create 2's wheel odometry was fine at tracking the linear motion of the robot, it was particularly terrible at tracking the rotational motion. If the robot were to spin in place it would quickly lose track of its orientation, causing there to be noticeable misalignment in the clouds from consecutive panning cycles. In typical robotic systems, consecutive scans would be compared to line them up and correct for this error, but for the VS-LIDAR system collecting a full point cloud takes on the order of seconds, during which time bad rotational odometry could distort the clouds and making error correction harder. This necessitates accurate short term odometry.

The IMU, on the other hand, provides a very accurate orientation estimate, but doesn't track the robot's linear velocities. Thus, these orientation estimates and the linear velocity data from the wheel odometry are fused using an Extended Kalman Filter (EKF). The EKF is a very popular filter in robotics, and in the context of robot navigation is used to estimate the robot's position, orientation, linear velocities, and rotational velocities, either in 2D or in 3D. The EKF estimates the robot's state using an internal kinematic model, which incorporates estimates of the robot's state from individual sensors. To what degree the sensors' estimates are incorporated is determined by variance-covariance matrices which accompany the sensor data, describing the uncertainty in those measurements.

The VS-LIDAR system uses the EKF implemented by the ROS *robot_localization* package; this implementation and the underlying algorithm are described in [33]. This package also implements an Unscented Kalman Filter (UKF), an improvement over the EKF to make it more accurate. However, only the EKF is used in this system, as the UKF was found to require significantly more configuration and fine tuning to get adequate performance.

To be used with the EKF, the error in the wheel odometry and IMU data needs to be modeled to obtain variance-covariance matrices. For the wheel odometry, this is done with the error model described in [28, pp. 270-275]. This model estimates the error in both the velocity (linear and rotational) and pose (position and orientation) of the robot, but for this system it is only necessary to estimate the error in the velocity. The reason for this is that the pose estimates and errors are obtained from integrating the velocity estimates and errors, which is something that the internal motion model of the EKF already does. Because of this, and the fact that the IMU is relied on for orientation information, the only portions of the wheel odometry being input to the EKF are the linear velocities in the robot's x and y directions. One may recall that in ROS the robot's x direction is always facing forward, regardless of what orientation relative to the world the robot is at. This means that for a robot like the Create 2, which cannot translate side-to-side, that the linear y velocity is always zero. Still, it is helpful to fuse this zero velocity in the EKF, since it tells the motion estimator that the robot is never be able to drive side-to-side.

For the IMU data an adequate error model was not found. In testing, the orientation estimate was discovered to be very stable and accurate, while the rotational velocities and especially the linear accelerations were more prone to noise. For this reason, only the orientation of the IMU is fused in the EKF, with the orientation variance and covariances being set to a low, static 10^{-6}, telling the filter that the data is trustworthy. The EKF has the ability to integrate the rotational velocities and the linear accelerations to estimate the robot's pose over time, but testing showed that incorporating this data only degraded the performance of the EKF, even with the associated variances and covariances set relatively high. The data was simply too noisy and was not published frequently enough for accurate integration.

The EKF in *robot_localization* can be configured to operate in either 2D or 3D mode. In the 2D mode, the robot's state variables associated with 3D motion (velocity and rotation about x and y axes, velocity and acceleration along z axis) are set to zero and fused with the EKF. Since the VS-LIDAR testing platform is constrained to operating in planar environments, the EKF is configured to operate in 2D mode. This does not significantly impact VS-LIDAR system, as a mobile platform using it in a 3D environment should be providing its own 3D odometry, and in the case of the Robotic Mining Competition the competition area is approximately planar anyway.

3.6.2 ArUco Marker Localization

Even with the improved odometry, the estimate of the robot's pose will grow without bound. A method of correcting for this drift is required for robots operating over the long term. Additionally, many robots need a means of localizing relative to some static map to complete their mission objectives. For example, for the Robotic Mining Competition a reliable means of locating the bin for material deposition is required. Knowing the location of the bin tells the robot not only where to go to deposit material, but also where it needs to go to mine material, as the competition area is split up into a starting area, an obstacle area, and a mining area. Both of these objectives can be completed by placing static beacons in the environment that allow a robot to know its position and orientation absolutely.

In 2014, researchers at Córdoba University released the open source ArUco library [34]. This system uses a camera to detect markers of known size and estimates the angle of and distance to the marker relative to the camera, which then allows the marker's coordinates to be estimated. On the face of this marker is a 7x7 grid, whose cells are either black or white. The border cells of the grid are black, with a black and white pattern contained within to uniquely identify the marker and to show the camera which way is up. Figure 3.4 shows the marker used in the VS-LIDAR system, with the width and height of the grid being 20 inches. To accurately measure the pose of the marker requires that the camera be calibrated so that lens distortion can be removed from captured images. For this system, the Genius webcam was calibrated at a resolution of 640x480 using the ROS *camera_calibration* package. A higher resolution would've made the marker detection more accurate, but a good calibration with the camera at its native resolution of 1920x1080 could not be obtained. The distortion caused by the wide-angle lens was too severe for the calibration program to handle.

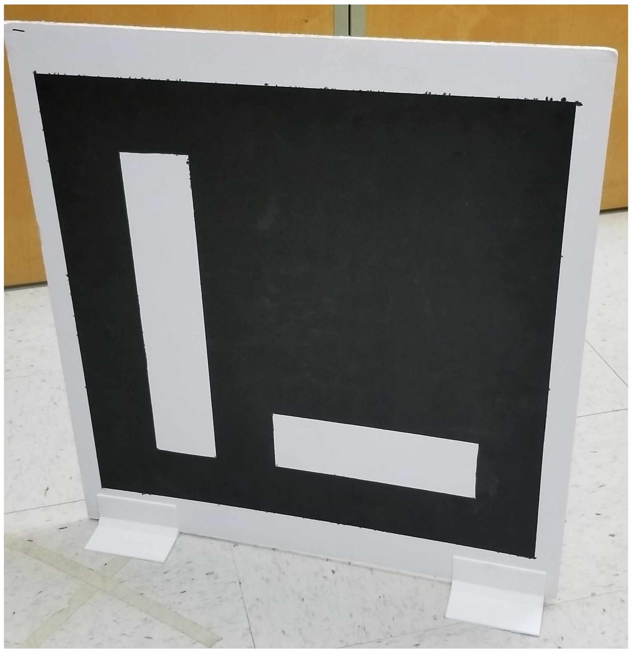


Figure 3.4: ArUco Marker Used in VS-LIDAR System

The VS-LIDAR system uses just the one marker shown, with the position of the marker considered to be the origin of the odometry frame. This makes it straightforward to plan paths relative to the marker, as will be done for the Robotic Mining Competition. To set the marker as the origin of the odometry frame, its pose is fused in the EKF as an absolute pose estimate for the robot.

The actual detection of the markers and their 3D poses is already implemented in ROS with the *aruco_mapping* package. However, the pose has to be processed before it is fused in the EKF, and this is done by the *aruco_pose_repack* node. First, this node obtains the pose of the camera relative to the marker, specified by x, y, and z coordinates, and roll, pitch, and yaw. Then the pose is restricted to 2D, so the z coordinate, roll, and pitch are discarded; since we place the marker ourselves, we know that the marker will always be level with the camera, so 3D information is superfluous. Next, the pose is transformed so that it is relative to the robot's center of rotation, to account for the fact that since the camera is offset from the center, its position relative to the marker changes even though the robot stays in the same place. This is done with Equations 3.1 and 3.2, where *camera_offset* is the distance between the camera and the robot's center of rotation, *yaw* is the detected yaw angle of the marker, x_{camera} and y_{camera} are the x and y coordinates of the camera relative to the marker, and x_{robot} and y_{robot} are the x and y coordinates of the robot relative to the marker, as

$$x_{robot} = -(x_{camera} + camera\_offset * cos(yaw)), \tag{3.1}$$

$$y_{robot} = y_{camera} - camera\_offset * sin(yaw). \tag{3.2}$$

With the yaw of the camera and position of the robot, we now have an absolute estimate for the robot's pose. However, it is not always desirable to integrate this pose with the EKF. Consider Figure 3.5. Here, the marker on the left-hand side denotes the starting position of the robot, the marker on the right-hand side denotes the position of the ArUco marker, and each grid cell is one square meter. The robot was slowly driven towards the marker, and the estimated pose of the robot was traced in green. When the robot was far away from the marker, the pose was very unstable, oscillating by about two meters. This oscillation gradually decreased as the robot got closer and closer to the marker. The cause of this variability is the fact that at greater distances, the marker becomes smaller and smaller in the camera's view, being represented by fewer pixels. This decrease in resolution of the marker makes it harder to estimate its pose, especially the marker's rotation. To address this, the *aruco_pose_repack* node throws away all pose estimates where the estimated distance is greater than one meter, as poses obtained closer than this were observed to be pretty stable. Another solution would be to use a higher resolution camera, but this would make the system more expensive, not to mention increase the processing requirements to find markers.

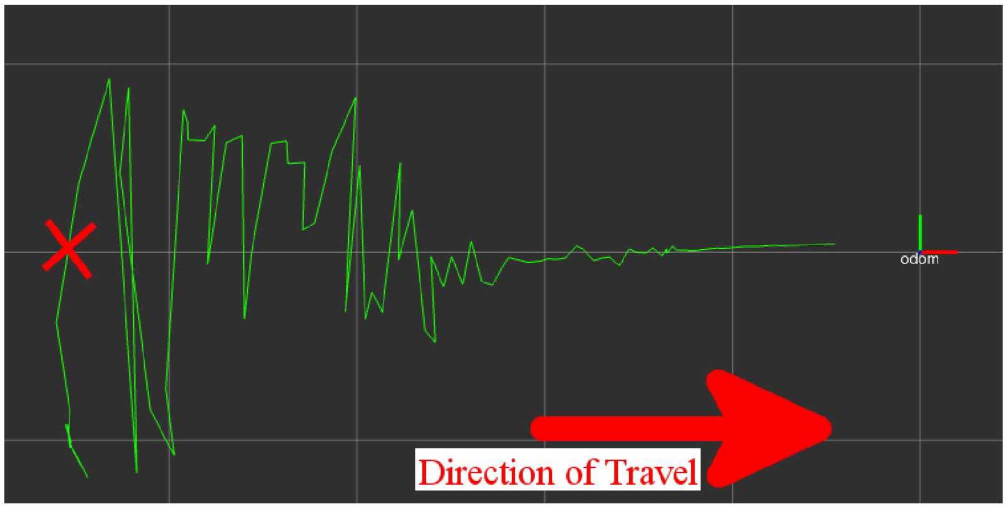


Figure 3.5: ArUco Pose Estimates with No Filtering

When the node has decided to keep a pose estimate, it publishes it to make it available to the EKF. To be compatible with the EKF, the pose needs an associated variance-covariance matrix. The EKF needs to be told that the pose estimate is trustworthy; if the system has been running for a while and significant errors in odometry have been accumulated, if the covariance in the marker pose is too high the EKF wont trust the sudden jump in position over its own internal motion model. Thus, the marker pose is given a low fixed covariance of 10^{-6}. The result of all this can be seen in Figure 3.6, which shows the reported pose of the robot when staring at the marker head-on from a distance of 75 cm, where the *base_footprint* marker shows the pose of the robot and the *odom* marker shows the pose of the ArUco marker. The distance between the two, as measured with the ArUco marker detection, was 75.8 cm; a difference of only 8 mm is certainly within tolerance.

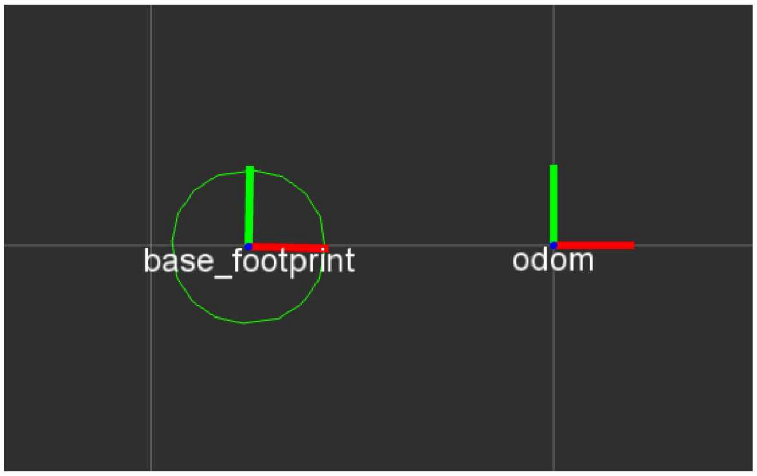


Figure 3.6: ArUco Pose Estimate at a Distance of 75 cm

3.6.3 ROS Navigation Stack: Overview & Configuration

As ROS has been used for indoor mobile robotics for a number of years, there already exists a rich set of packages for autonomous 2D navigation and localization known as the navigation stack. Because the mining robot will be operating in an approximately planar environment, it makes sense to take advantage of these preexisting tools. At the core of this stack is the *move_base* node, which ties together the other packages, implementing a system that, given a waypoint to navigate to in the world and sensor data providing the locations of obstacles, attempts to move the robot to the waypoint. To accomplish this requires two key components, costmaps and path planning. Essentially, the costmaps keep track of where obstacles are in the world and are used to plan paths around obstacles to goal locations. The following sections will go into greater detail about these two components and their configuration.

3.6.3.1 Costmaps

Costmaps are 2D maps which discretize the robot's environment into a grid, and store information on whether each cell is likely to be occupied by an obstacle or not. These maps are built in layers, of which there are several standard types:

- **Static Layer**: This layer extracts obstacles from a map of the environment provided to the navigation stack. This map can either be compiled beforehand or created on the fly as a robot explores its environment; in the previous case, this map would likely contain only macro features such as walls and furniture which are not expected to move.

- **Obstacle Layer**: This layer stores information on which cells in the costmap are occupied by obstacles. The obstacle layer can either be 2D or 3D: in 3D the 2D cell grid is turned into a 3D grid of voxels, or cubes. Given sensor data the obstacle layer performs two primary tasks: marking and clearing. In marking, the system checks whether received points fall within a range defined by a minimum and maximum obstacle height. If so, the cell or voxel that point occupies is marked in the obstacle layer. Clearing obstacles from the costmap is done using raytracing; for each data point, a line is traced from the sensor's origin to that point, and if no data points occupy the cells or voxels that the line passes through they are marked as no longer containing obstacles. The primary purpose of using a 3D obstacle grid is so that raytracing marks and clears obstacles properly when given data from 3D sensors. Regardless, the obstacle layer is projected down to 2D when used for navigation.

- **Inflation Layer**: Using information from the obstacle layer, the inflation layer creates a potential field around obstacles to describe the safety with which a robot can plan a path through the costmap's cells. This potential field can be split up into cells that will definitely result in collision, cells that will possibly result in collision, and cells that will not result in collision. The potential of each cell is given a value from 0 to 254, with 0 being completely free space and 254 being space occupied by a sensor reading of an obstacle. Given a footprint describing the shape of the robot, the costmap calculates the inscribed and circumscribed radii. Cells one inscribed radius or closer an obstacle cell are given a value of 253 to indicate that the robot will be too close to the obstacle, most certainly colliding. From there, the cost is exponentially decayed as the distance from the obstacle cell increases. Cells that are further away than the inscribed radius but closer than the circumscribed radius are decayed to be anywhere between 252 and 128, to indicate the robot could collide with an obstacle if it were oriented in the wrong way. The rest of the cells that are further away than the circumscribed radius but within a user-specified inflation radius are decayed from 127 down to 1. This decay encourages the robot to give obstacles a wider berth; if all cells outside the circumscribed radius were marked as free, the path planning would naturally be greedy and plan the shortest path through the free space, causing it to drive right next to obstacles, possibly colliding with them due to an imperfect view of the world or sensor noise.

To help visualize exactly what a costmap looks like, Figure 3.7 shows the VS-LIDAR looking at some obstacles and Figure 3.8 shows the resulting costmap. In the VS-LIDAR system, costmaps are comprised of a 2D obstacle layer and an inflation layer; as there is no map to provide to the navigation stack a static layer is not used. For navigation, two costmaps are kept: a local costmap which is updated frequently and only stores information on the immediate surroundings of the robot, and a global costmap, which is updated less frequently but stores information about all the places the robot has been.

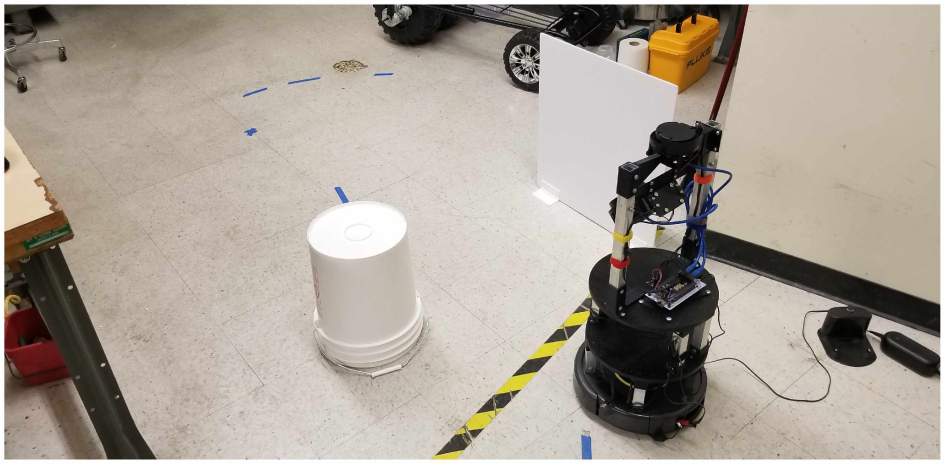


Figure 3.7: Costmap Example Environment

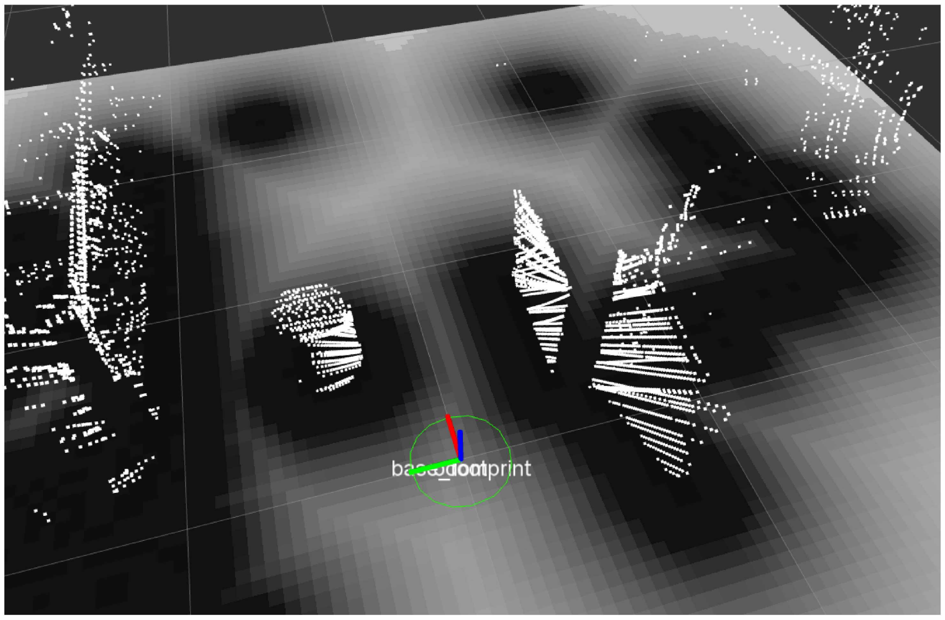


Figure 3.8: Costmap Example

Table 3.1 summarizes the important parameters in the VS-LIDAR's costmap configuration. The inflation radius around obstacles is set to 1 m, which provided the best performance during testing, obstacles within 2.5 m of the robot are added to the costmap and data points within 3.0 m of the robot are used to raytrace points out of the costmap. The raytrace range is slightly longer than the obstacle range to prevent the costmap from adding obstacles to the edge of the map that it cannot later raytrace away. The observation persistence is 6.0 seconds, meaning that even if a cell is cleared by raytracing it will stay marked as an obstacle for that length of time; the reason for this is explained in Section 4.5. The rest of the parameters describe the size, resolution, and update rates of the local and global costmaps. Since the global costmap is much bigger than the local costmap, its resolution is cut in half and its update rate is one-fifth that of the local costmap. As the *cloud2obstaclemap* node only contains obstacles, the specification of minimum and maximum obstacle height is not important so long as the range includes all points in the obstacle cloud.

Table 3.1: Important Costmap Parameters

Parameter	**Value**
Obstacle Inflation Radius	1.0 m
Obstacle Range	2.5 m
Raytrace Range	3.0 m
Observation Persistence	6.0 s
Local Costmap Update Rate	5 Hz
Local Costmap Resolution	5 cm
Local Costmap Size	6 m by 6 m
Global Costmap Update Rate	1 Hz
Global Costmap Resolution	10 cm
Global Costmap Size	100 m by 100 m

3.6.3.2 Path Planners

The second major component of the navigation stack are the path planners, and there are two of these as well. The first is the global planner, which uses the global costmap to plan a path from the robot's starting position to its ending position. This planner uses Dijkstra's algorithm to find a minimum cost path through the costmap's cells. The global path is used as a guide for the second planner, the local planner, which utilizes the local costmap. There are a few local planners available in ROS, but the one used for the VS-LIDAR system uses the Dynamic Window Approach (DWA) algorithm.

In this algorithm, the control space of the robot, i.e. the range of speeds at which the robot can turn and drive, is discretely sampled. Each combination of linear and rotational velocities is used to produce a circular trajectory through the costmap by simulating what would happen if the robot drove at those speeds for a set amount of time. Each of these trajectories are scored based on their proximity to obstacles, their proximity to the navigation waypoint, their proximity to the global path, and the robot's driving speed. Trajectories that cause the robot to collide with obstacles are immediately discarded. Then, the robot is commanded to drive at the velocity associated with the highest scoring trajectory. This process repeats until the robot has reached its goal or has failed in some unrecoverable way. For visualization, an example of these planned paths being used in the VS-LIDAR system is shown in Figure 3.9, where the robot was told to navigate around the white bucket in the obstacle field shown in 3.7. The green circle in the figure represents the footprint of the robot, the blue line represents the global path, and the red line illustrates the local path.

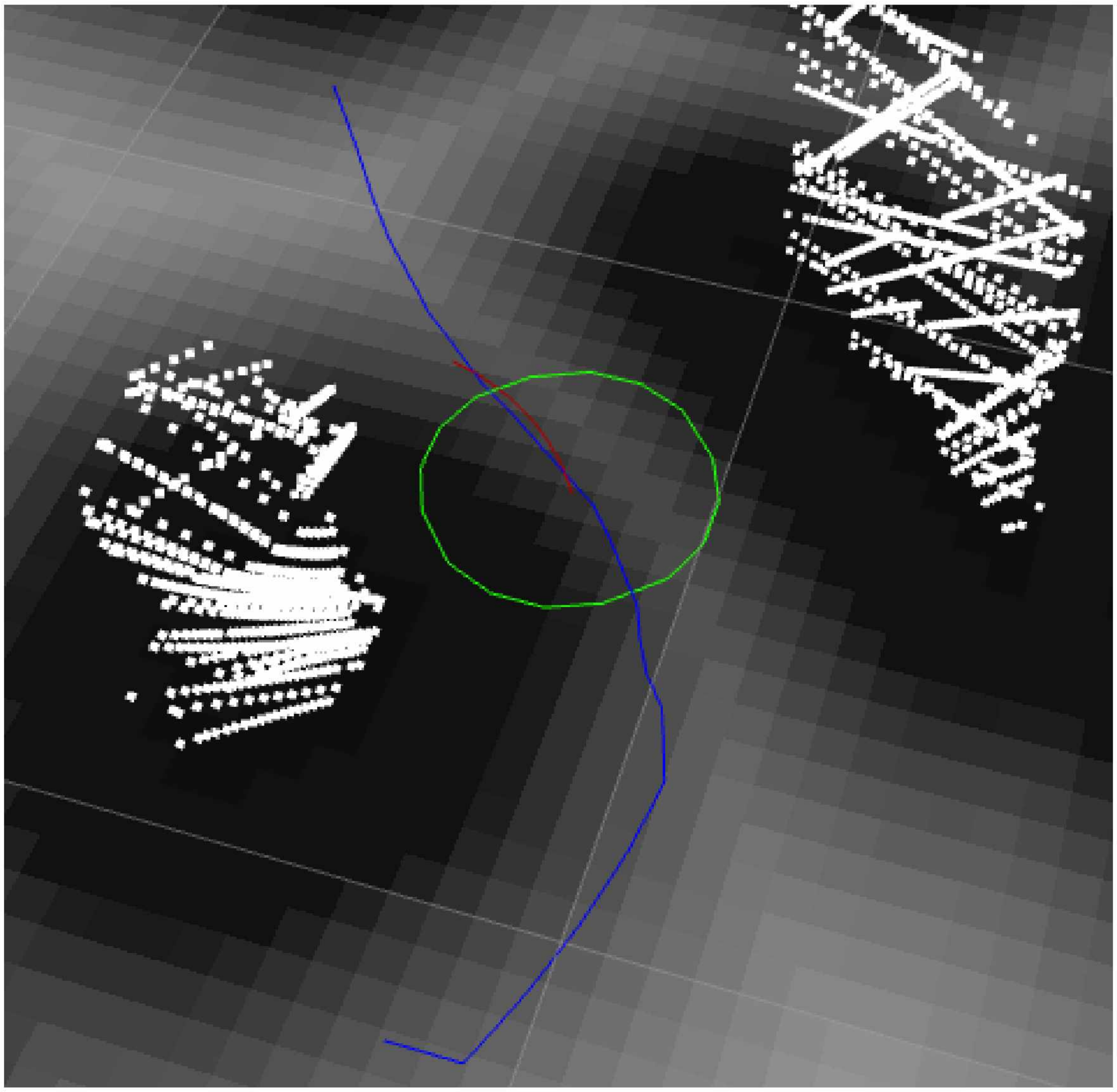


Figure 3.9: Path Planning Example

Table 3.2 summarizes the important parameters in the VS-LIDAR's local planner configuration. The first set of parameters simply describe the allowable motions of the robot. It should be noted that the Create 2 has actual velocity limits of 0.5 m/s and 4.25 rad/s for the linear and angular velocities, respectively, but the speed limits for the planner a set below this. This is to compensate for the slow scanning speeds of the LIDARs, making the system safer by giving it more time to detect obstacles. The yaw goal tolerance and xy goal tolerance simply state how far the robot has to be from a goal point to consider the navigation successful, and these help if the waypoint ends up being right on top of an obstacle. The final set of parameters controls the forward simulation of the DWA algorithm. Sim time is how far out in the future the trajectories should be simulated. If the value for this is too low, then the system won't have time to plan optimal trajectories through tricky spaces like narrow passageways; however, since the trajectories are simple circular ones, if the sim time is too long it will produce inflexible paths. From testing, a sim time of 2 seconds seemed appropriate. The sim granularity is simply how often along the trajectory the system checks whether the path has collided with an obstacle or not. How finely or coarsely the robot's allowable linear and rotational velocities are discretized is controlled by the Vx and Vtheta samples parameters. A low value for Vx samples was found to be sufficient due to how slowly the robot moves. Vtheta samples was made higher than Vx samples because of the wider range of allowable turning speeds.

Table 3.2: Important Local Planner Parameters

Parameter	**Value**
Max Velocity X	0.15 m/s
Min Velocity X	0.05 m/s
Max Velocity Theta	3.5 rad/s
Acceleration Limit X	2.0 m/s^2
Acceleration Limit Theta	2.0 rad/s^2
Yaw Goal Tolerance	0.157 rad
XY Goal Tolerance	0.25 m
Sim Time	2 s
Sim Granularity	0.2 s
Vx Samples	6
Vtheta Samples	20

Chapter 4 System Performance

4.1 VS-LIDAR Characteristics

This section summarizes some of the basic characteristics of the VS-LIDAR system. To start with, Figure 4.1 provides an illustration of the VS-LIDAR's field-of-view (FOV), with the FOV of the front-sweeping LIDAR highlighted in blue and the FOV of the side-to-side sweeping LIDAR highlighted in red. This illustration assumes there to be a ground plane beneath the robot to make it more realistic. One can think of these views as being constructed by rotating the planar slice of the environment obtained from each LIDAR about their servo's axis of rotation. As the plane sweeps through space it constructs the 3D view. It is important to note that this FOV is not instantaneous, but rather it is the FOV of one complete panning cycle for each LIDAR. One can see that the side-to-side sweeping LIDAR does not capture much of the scene in front of the robot, but obtains a wide view of things to the sides of the robot. The front-sweeping LIDAR, on the other hand, can see much more of the space in front of the robot and some space behind the robot as well, but does not see as much to the sides of the robot. What it does see to the side is about a meter off of the ground, so its side view is not as useful as the view from the side-to-side LIDAR.

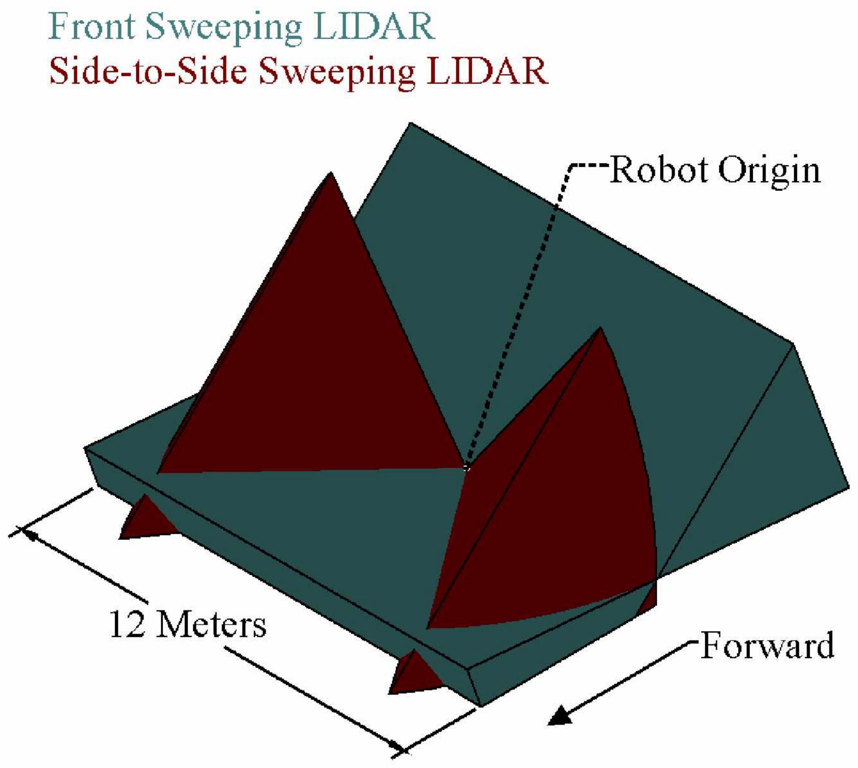


Figure 4.1: VS-LIDAR Configured Field-of-View

Table 4.1 lists some basic specifications of the VS-LIDAR system. In comparison to the Kinect v1 and v2, whose distance measurements alone take up 18.5 MB/s and 31.3 MB/s, respectively, the bandwidth usage of the VS-LIDAR is very low, making it much easier for the control laptop to process the data and interface with other peripherals on the bus. The LIDARs can be panned a maximum speed of 114 RPM, but testing showed 25% of this and lower to be more than adequate. The average power consumption of the VS-LIDAR, set to pan at a baseline speed of 7.5 RPM, was 4.2 W, though one can expect this value to increase if they pan the servos faster. The last parameter describes the minimum bounding box the sensor could be fit inside while still providing clearance for the LIDARs.

Table 4.1: VS-LIDAR Basic Specifications

Specification	**Value**
Data Points per Second	3700
USB Bandwidth Usage	17.3 kB/s
Maximum Panning Speed	114 RPM
Average Power Consumption (7.5 RPM)	4.2 W
Minimum Sensor Bounding Box (w x l x h)	22 cm x 22 cm x 32 cm

4.2 Odometry Tests

To determine the level of improvement in odometric localization from fusing the wheel odometry with IMU data, a test was performed where the robot was driven in a full lap around one floor of our building, with the robot starting and stopping in the same position. The robot was driven by a human operator at a speed of around 0.5 m/s. Figure 4.2 shows the path of the robot as tracked by both the raw wheel odometry and by the filtered odometry, with the obtained map overlaid for better visualization of the testing region. One can see from this figure that the filtered odometry performed much better than unfiltered odometry, with the unfiltered ending position being 19.2 m away from the actual ending position, and the filtered ending position being 2.0 m away. By examining the path of the unfiltered odometry, one can see that most of this error is attributable to incorrect tracking of the robot's rotation. Even when the robot is driving straight, the wheel odometry thinks the robot is arcing slightly to the left, and over time these errors accumulate. With the IMU providing a superior orientation estimate, drift in the robot's rotation becomes much less a concern. However, one can see that this filtering is unable to correct for drifts in position. Without loop closures, this drift would grow without bound.

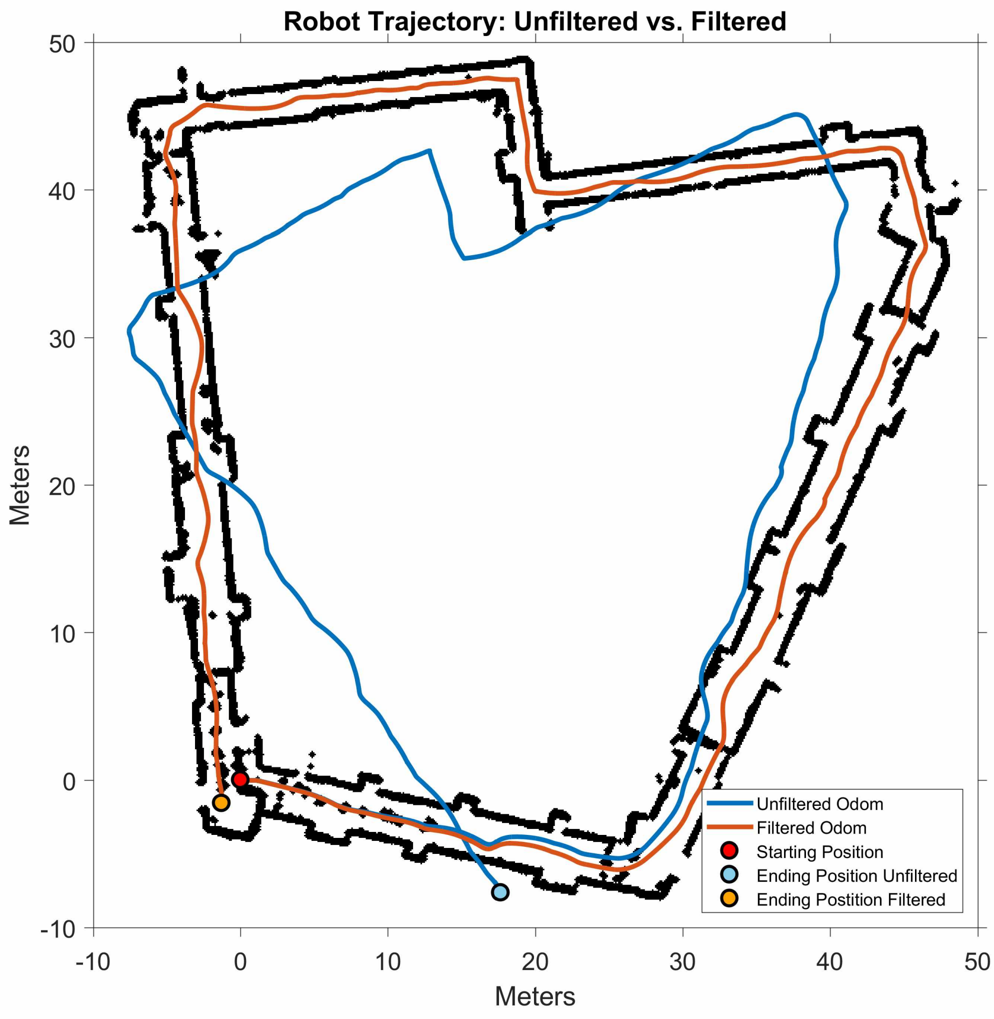


Figure 4.2: Comparison of Filtered Odometry to Unfiltered Odometry

To demonstrate the use of the ArUco marker to correct for drifts in position, the robot was driven along the same path as before, this time with an ArUco marker placed near the starting position. The robot was started facing the ArUco marker, and was again driven by hand with a speed of about 0.5 m/s. Figure 4.3 shows the path of the filtered odometry. When the robot saw the marker again at the end of its run, its position jumped back to its starting position. One can see in the lower right corner of the figure, however, that there was a brief amount of overshoot in its position estimate before it settled on the correct value. This suggests that perhaps the ArUco pose filtering should be a bit more robust than a simple distance threshold, if a brief incorrect position turns out to be a problem.

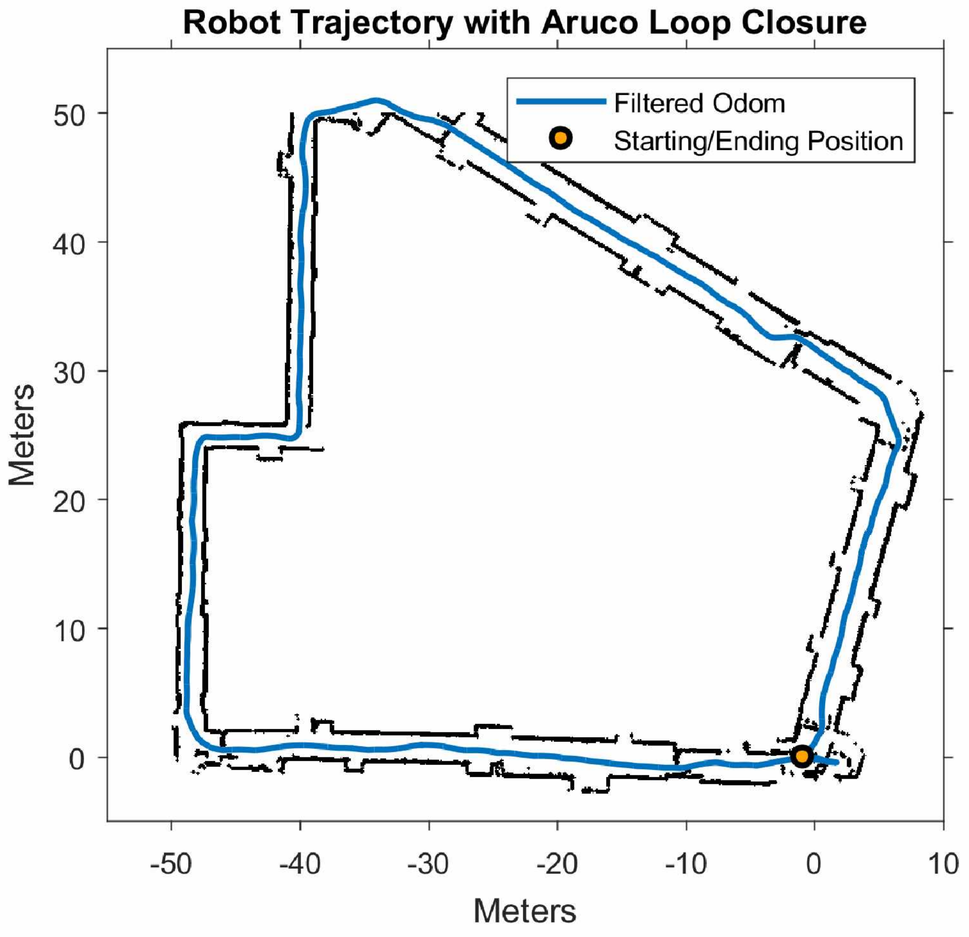


Figure 4.3: Using ArUco to Perform Loop Closures

4.3 Stochastic Scan Dithering

To illustrate the effect of randomly dithering the LIDAR rotation rates, an experiment was performed where the robot was kept stationary and the point clouds from two consecutive panning cycles were collected. A small (10 cm x 7 cm x 9 cm) 3D printed head was placed about a meter away from the robot in the field-of-view of the side-to-side sweeping LIDAR; this setup is shown in Figure 4.4. Figure 4.5 (a) shows the resulting cloud without dithering, and Figure 4.5 (b) shows the results with it.

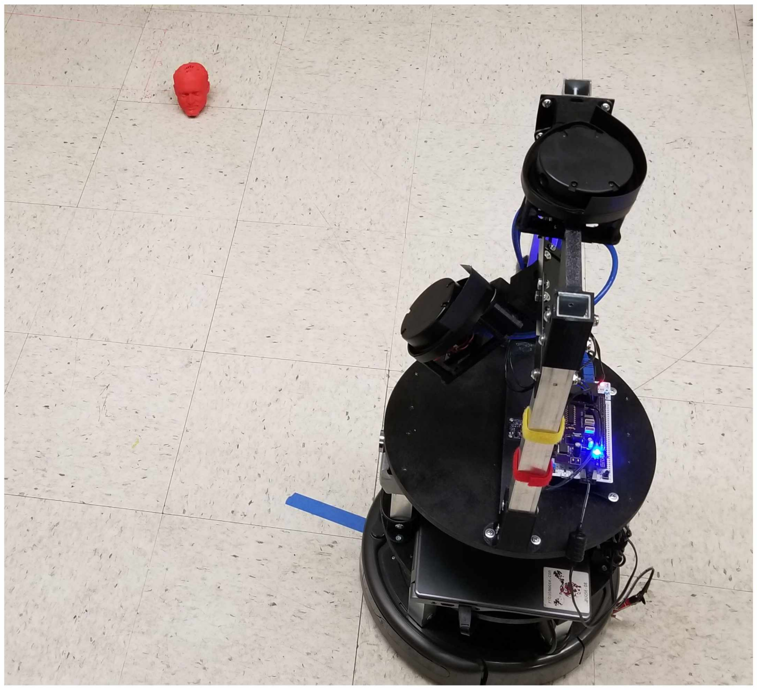
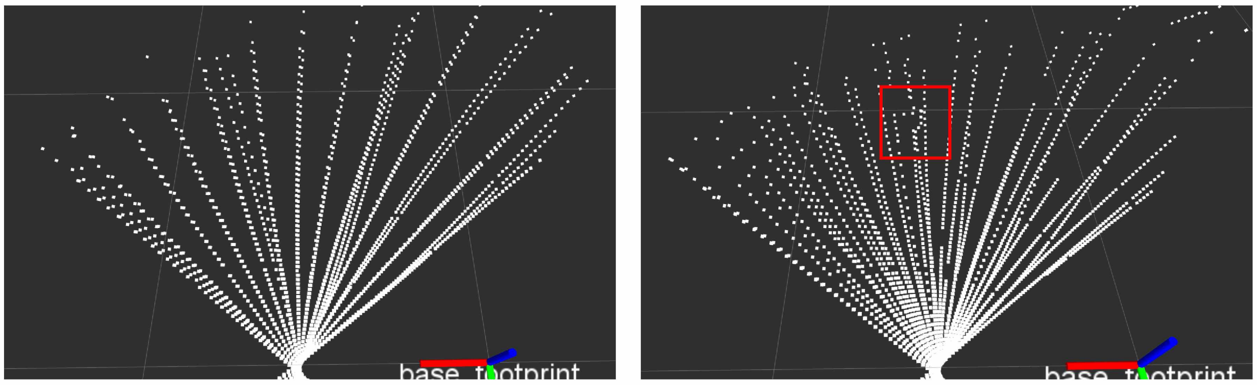


Figure 4.4: Random Dithering Test Setup



(a) Without Random Dithering (b) With Random Dithering

Figure 4.5: An Illustration of the Effect of Dithering on Point Cloud Data

One can see from Figure 4.5 (b) that with random dithering the scan lines are nice and spread out. The system was able to detect the head, as noted by the perturbation in the scan lines marked by the red box. In comparison, one can see from Figure 4.5 (a) that without random dithering the scan lines are much more clumped together, leaving large slices of unscanned space. In fact, the head happens to fall within one of these unscanned slices, so without dithering the system did not detect it. Thus, it is clear that random dithering improves the detection of small obstacles. However, the random nature of the dithering meant that the scan-line placement wasn't always optimal.

Sometimes the lines would be dithered very well, and sometimes not so much. Additionally, the system didn't always fail to detect the head without dithering, as over time imperfect timing control of the panning would cause the scan lines to shift around on the map. However, it was clear from observing both dithered and non-dithered clouds for a while that on the whole, random dithering was able to detect the head much more consistently.

4.4 Obstacle Detection Test

The objective of this test was to determine the obstacle detection capabilities of the front-sweeping LIDAR and side-to-side sweeping LIDAR individually, and show how the two sensors work together to make up for their individual downsides. For this test, the VS-LIDAR was placed in a stationary position while a variety of obstacles were placed around it: a measuring tape; a coffee cup; a small, yellow box; a tube of sanitizing wipes; a white, upright panel; and a Fluke instrument case. Figure 4.6 shows this setup. Each LIDAR was panned at a baseline speed of about 7.5 RPM, with a random offset between ±5 RPM applied every 500 ms to help distribute the points more evenly throughout the scanned region. Point clouds from each sensor were observed to see which obstacles they could see and which they couldn't.

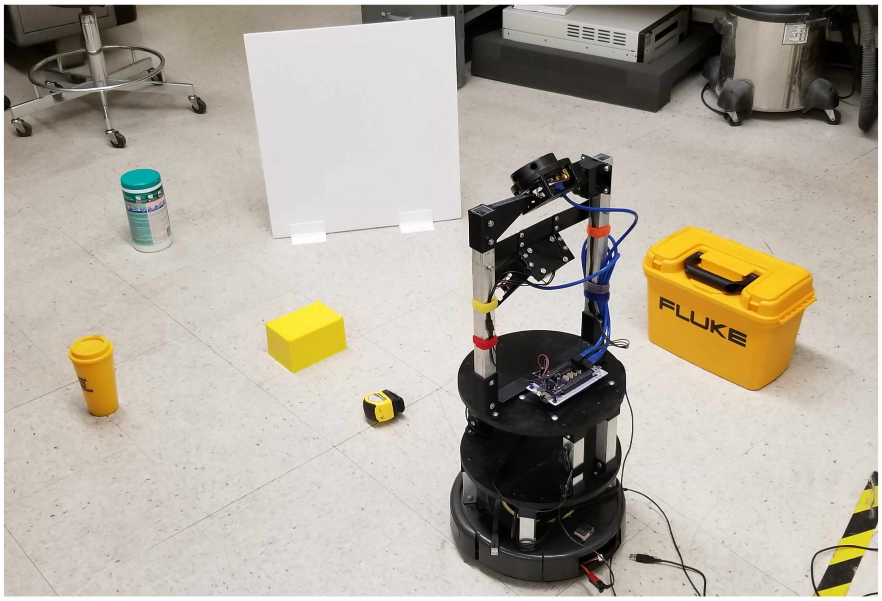


Figure 4.6: VS-LIDAR Obstacle Detection Test Setup

The following point clouds represent two consecutive panning cycles worth of point cloud data, and the marker with the red, green and blue axes represent the position of the robot, with the red axis pointing forward, with both of these things to make it easier for the reader to connect the data to the real-world scene. Figure 4.7 shows the point cloud obtained with the front-sweeping

LIDAR. As one can see, this LIDAR gives a fairly wide view of what is in front of the robot, being able to see the white panel, the tube of sanitizing wipes, the coffee cup, and the yellow box. However, it is unable to see the measuring tape, which is more directly in-front of it, or the Fluke instrument case, that is to its right. Now, the sensor could be made to see the measuring tape if the side-to-side LIDAR was removed and the front LIDAR was placed further forward and made to pan more deeply, but the primary purpose of the front-sweeping LIDAR is to quickly scan area in front of the robot to capture information about upcoming obstacles, either static or dynamic, so that the system can more effectively plan paths to navigation goals. To detect all obstacles on its own its panning range would have to be extended, lowering the update rate of the sensor, forcing the robot to have to drive more slowly and further delaying its detection of both static and dynamic obstacles.

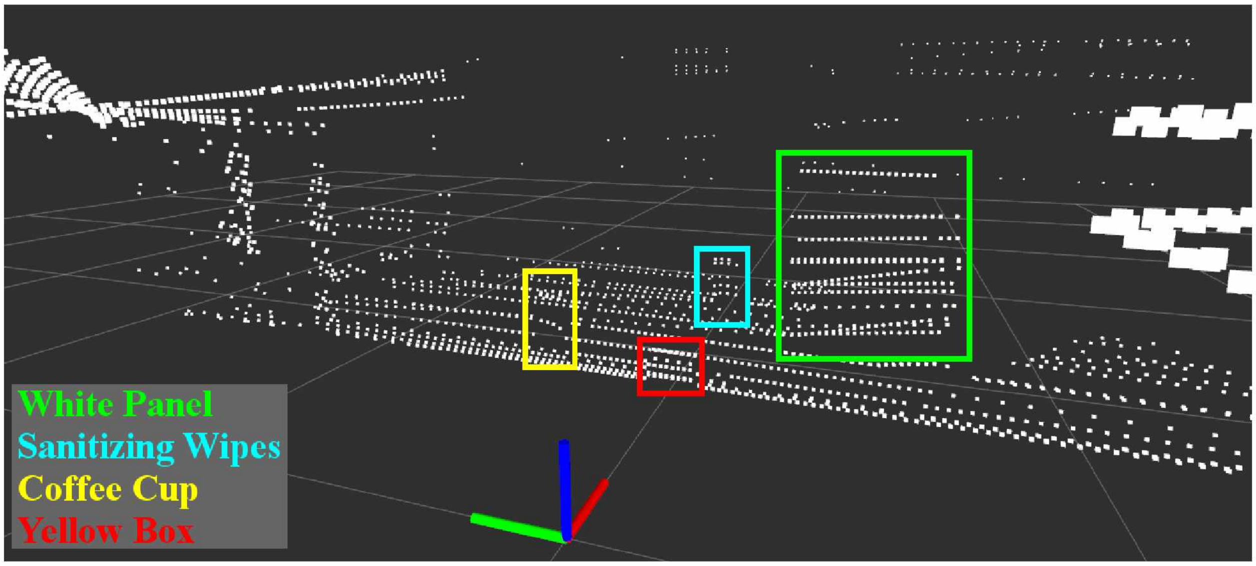


Figure 4.7: Obstacles Detected with Front Sweeping LIDAR

Now consider Figure 4.8, which shows the point cloud obtained from the side-to-side sweeping LIDAR. This sensor was able to see the coffee cup, the tape measurer, the Fluke instrument case, and a portion of the white panel. As opposed to the front-sweeping LIDAR, which is designed to give a wide view of what is in front of the robot, the side-to-side sweeping LIDAR is designed to not only give a detailed view of what is to the sides of the robot, but is angled downward so that a spot directly in-front of the robot is always updated in real-time. Also, by observing the coffee cup, one can see that the side-to-side sweeping LIDAR maps close, small obstacles with much greater fidelity than the front-sweeping LIDAR panning at the same speed. This was not some chance observation caused by the random scan dithering, but was in-fact consistently observed throughout the experiment and is present without dithering as well. The higher fidelity and real-time scan region of the side-to-side sweeping LIDAR allow the robot to quickly respond to obstacles the front-sweeping LIDAR could've missed, while the side-view allows the robot to more quickly plan paths that involve rotations. Of course, this LIDAR has a longer panning range than the front-sweeping LIDAR (90° vs. 50°) and its real-time update region is quite small, so it is less adept at dealing with dynamic obstacles.

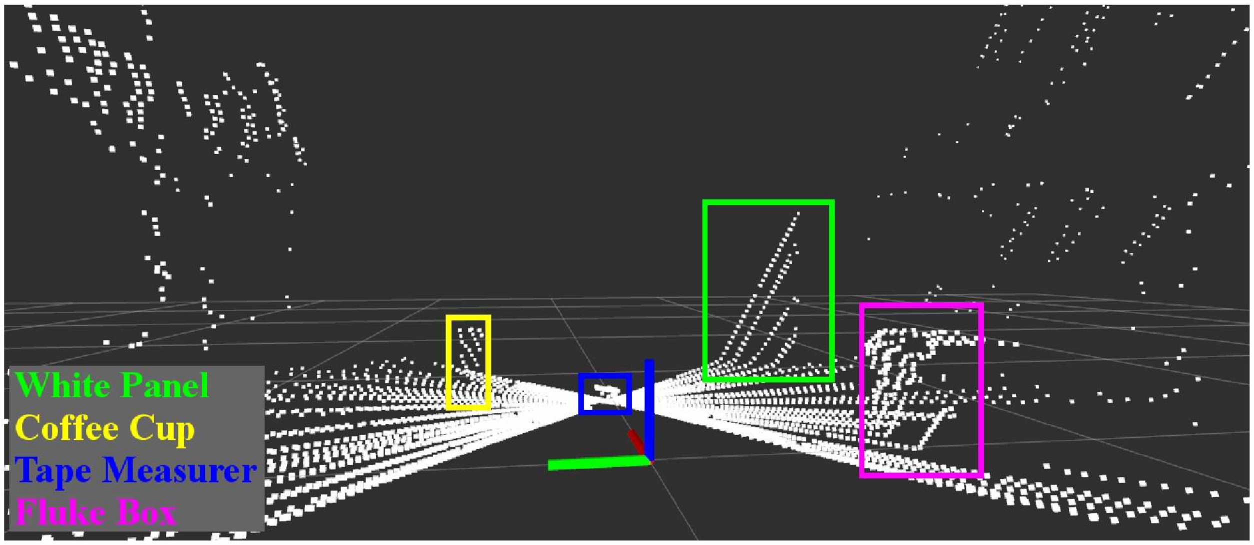


Figure 4.8: Obstacles Detected with Side-To-Side Sweeping LIDAR

Figure 4.9 shows the combined view of the obstacle field from the two sensors. Using a pair of LIDARs gives the VS-LIDAR system a wide field-of-view of its environment and provides it with a much better chance of detecting obstacles. This also allows the system to rotate the LIDARs more quickly than would be possible with just one LIDAR, as a single LIDAR system would have to pan more slowly to get a good view of the field, which allows the point clouds to have a better refresh rate and allows the robot to complete its tasks more quickly.

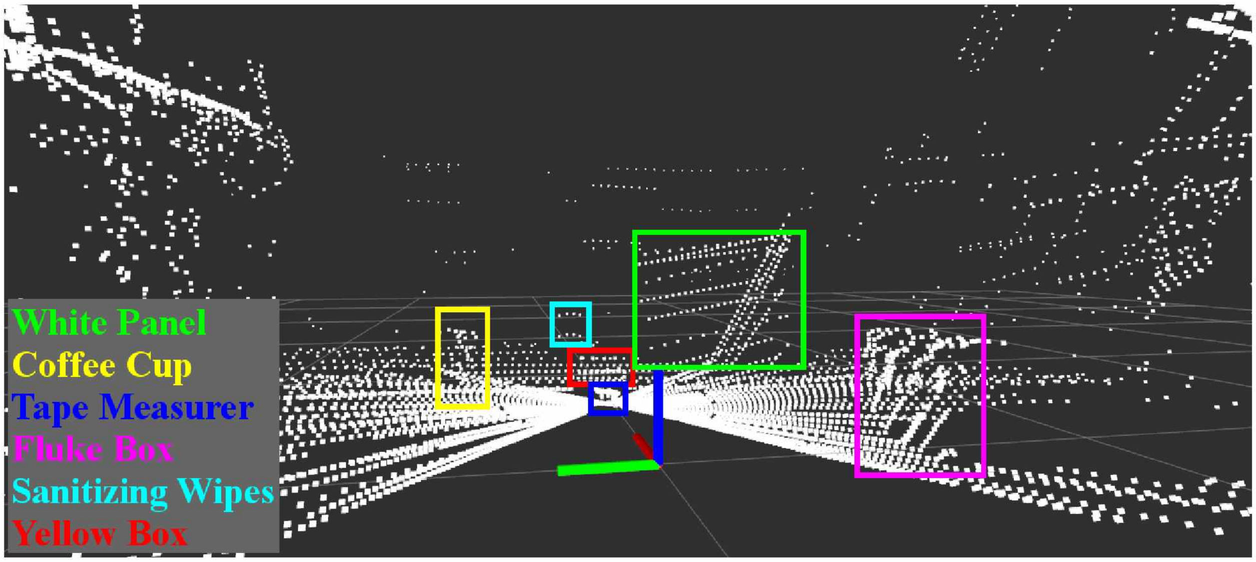


Figure 4.9: View of the Obstacle Field Using Both LIDARs

4.5 Navigation Tests

To demonstrate the possibility of using the VS-LIDAR system for autonomous navigation, the obstacle course shown in Figure 4.10 was constructed. With the robot starting in the position shown, it was given a navigation waypoint near the door. Small obstacles — the white mug and the small tea box — were specifically placed so that the robot would run into them if it was simply following the edges of the larger objects. These navigation trials were performed several times.

In a majority of trials the robot would expertly navigate the course, carefully skirting the obstacles to reach its destination; however, it some trials it would hit the tea box and/or the coffee mug, sometimes even clipping the green box. This was not an issue of not seeing these obstacles, as the robot was moving and scanning slowly enough that it saw them pretty consistently. The problem was that sometimes it would stop next to a small obstacle to plan its way around it, and the LIDARs would no longer be able to see it. This would cause the costmap to forget about the obstacle and thus the robot would drive right into it, thinking it to be empty space. Additionally, the robot would sometimes forget about obstacles that weren't in its field-of-view, but this wasn't consistent.

The cause of this is the fact that the obstacle maps are being made in 2D. When the system doesn't see an obstacle, but sees a point behind it, it traces a ray from the robot to that point and clears all costmap cells in-between. Since the 2D obstacle map projects all points to the XY-plane for obstacle marking and raytracing, this can remove obstacles it can't see. A workaround to this is to increase observation persistence so that obstacles stay within the costmap a set amount of time, giving the robot time to navigate around unseen obstacles. However, this can still fail if the

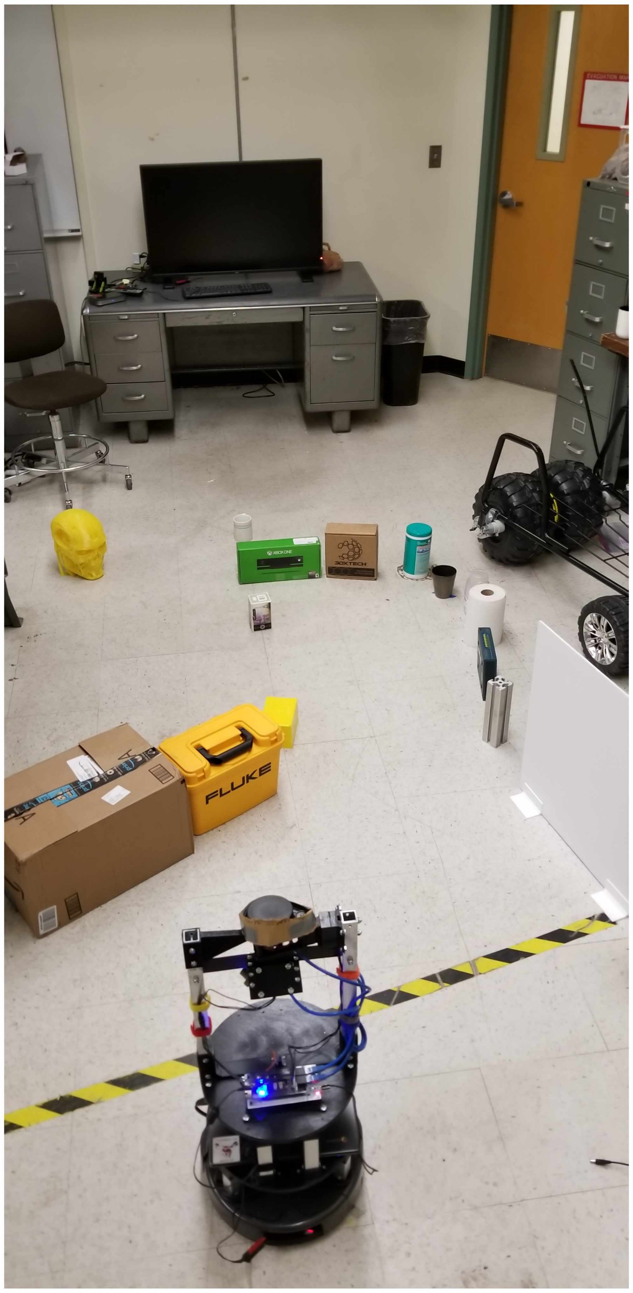


Figure 4.10: Obstacle Course for Navigation Testing

robot takes too long to navigate around, makes the system more sensitive to outlier data points, and makes the system slower to clear space that moving obstacles used to occupy.

The more natural solution is to use 3D obstacle maps and 3D raytracing, since then obstacles outside of the robot's field-of-view shouldn't be phased out of existence. An attempt was made to configure the costmaps to build 3D obstacle maps, using the *LaserScan* data as input instead of the point clouds so that rays would be traced from the origin of the laser sensor itself instead of from the robot's origin. In 2D, it was fine to start the raytrace from the robot's origin rather than the sensor origin since everything is projected down to the XY-plane, but when raytracing in 3D space it is much more important to start at the data's exact origin so that the correct voxels are cleared. However, the 3D obstacle maps never quite worked. In particular, the raytracing still didn't work properly, as the costmap was hardly able to clear obstacles at all.

Back to the 2D obstacle mapping, a quick set of tests was performed to evaluate the dynamic obstacle avoidance capabilities of the robot. In these trials, the robot was given a navigation waypoint and a person would get in its way, sometimes walking in front of it from the side and sometimes walking towards it through its waypoint. Given that humans are relatively large obstacles that the front-sweeping LIDAR is almost certain to see immediately, in all cases the robot was able to stop, navigate around the individual, and continue towards its goal.

In conclusion, these tests demonstrated that the VS-LIDAR system has the potential to be used for autonomous operation in areas where both static and dynamic obstacles are present. However, the ROS navigation stack requires significantly more configuration and testing to work robustly with these sensors, or a custom navigation package needs to be created that takes into account the unique configuration of the VS-LIDAR. The first solution is preferred, since the ROS navigation stack is constantly maintained and updated, and already in use by a large number of robotic systems.

4.6 Detection of Black Obstacles

As previously mentioned, LIDAR sensors have trouble detecting black objects since they absorb laser light. It depends on what type of black color it is: for example, the VS-LIDAR was able to detect a black coat just fine, but in general one can expect a harder time. To demonstrate this shortcoming of the VS-LIDAR system the ArUco marker was placed 0.5 m away from the robot (indoors). The resulting point cloud is shown in Figure 4.11. One can see the system was unable to see the black parts of the marker. The same result was obtained when the marker was 1.5 m away or more. However, it was discovered that at 1.0 m away the front-sweeping LIDAR was able to detect most of these black spots, save for a band near the bottom, as shown in Figure 4.12 (a), and by slightly tilting the marker backwards the whole of it could be seen, as shown in Figure 4.12 (b). This lead to a suspicion that the system could scan black surfaces as long as the laser hit them at an angle that would reflect the non-absorbed light nearly directly back towards the imaging sensor, and sure enough with the laser's scanning plane perpendicular to the marker's planar face it was able to see the black section with much more consistency, out to about two meters. Thus, it appears that these LIDARs were made just powerful enough to detect black surfaces head on, and that the act of rotating them compromises their ability to see black surfaces consistently. To clarify, the system can see black obstacles that are at about the same level as the front-sweeping LIDAR, as it will hit them head on, but will miss those low to the ground if they are the wrong type of black. This is unfortunate, but may not be a terribly big deal. Depending on where the VS-LIDAR is operating, most obstacles the robot will typically encounter will probably not be so uniformly black as to be completely invisible. This is especially true for the Robotic Mining Competition, where there will be no black obstacles whatsoever.

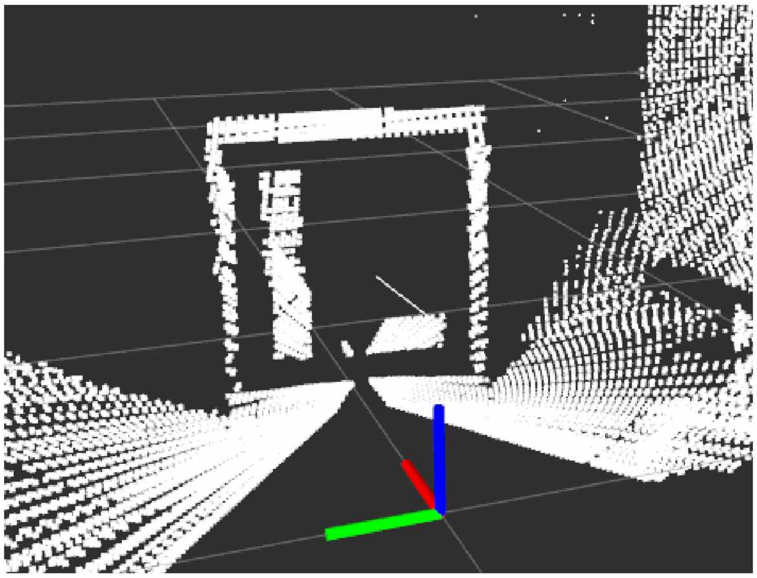
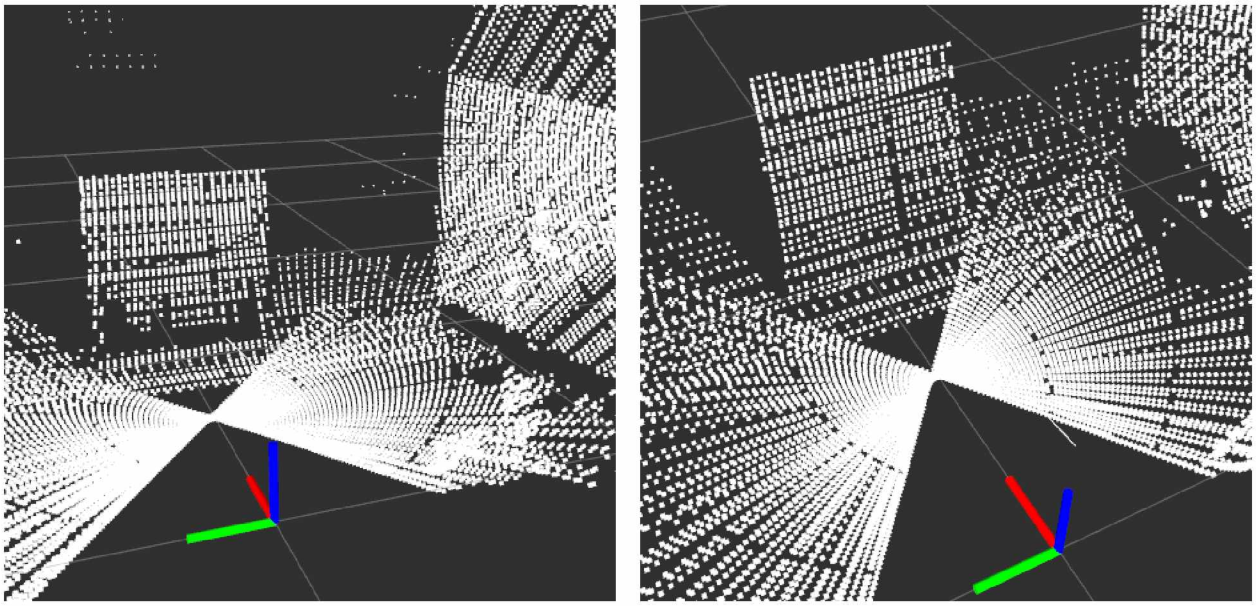


Figure 4.11: LIDAR View of ArUco Marker from 0.5 Meters Away


(a) Without Tilt

(b) With Tilt

Figure 4.12: LIDAR View of ArUco Marker from 1.0 Meters Away

4.7 Performance in Sunlit Environments

A primary reason for developing the VS-LIDAR system was to create a low-cost 3D sensor capable of operating in a wider range of environments than cheaper depth cameras. In particular, depth cameras have more trouble operating in the presence of sunlight, being primarily designed for indoor spaces. To illustrate this, the capability of the VS-LIDAR to detect a target outdoors was compared to that of a Kinect v1, a structured-light depth camera, and a Kinect v2, a time-of-flight depth camera. The target was a white rectangle, 22 inches wide by 23.75 inches tall, and in all tests was placed 1.5 meters away from the sensor under investigation, with the sensor looking directly at the target.

The first test took place on a somewhat overcast day, with the ambient light level at the target being around 10,000 lux, as measured by the TMD4906 ALS optical module inside the Samsung Note 8 smartphone. This test involved the Kinect v1 and the VS-LIDAR system, and the testing setup for the VS-LIDAR is shown in Figure 4.13. The VS-LIDAR was able to see the target and the surroundings with ease, as shown in Figure 4.14, while the Kinect was unable to see anything whatsoever. However, it was discovered in this test that the increased illumination level increased the noise in the LIDAR data, resulting in a significant increase in the number of erroneous distance measurements when the LIDARs were pointed towards the sky. This garbage data can be easily seen as the sparse points around the left edge of Figure 4.14, where there was, in-fact, empty space. This data would be easy to remove with statistical outlier filtering, but no such filtering is done in the current system. The reason for this is that if the LIDARs are scanning quickly, a small obstacle might be represented by just a few points and could get removed by such filtering. It is much safer for the robot to detect false positives then to accidentally get rid of an obstacle.

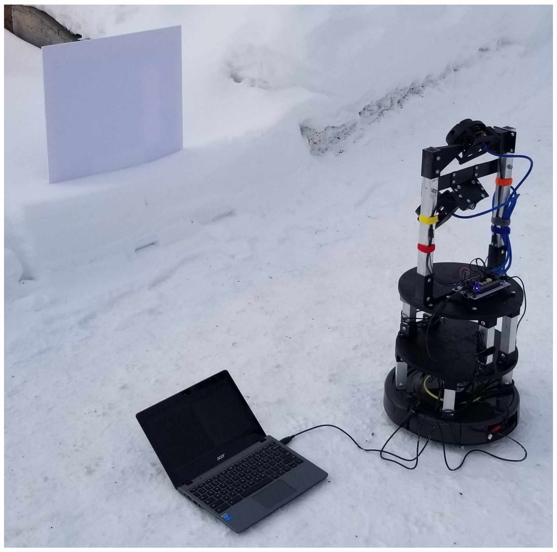


Figure 4.13: Outdoor Perception Test #1 Setup VS-LIDAR

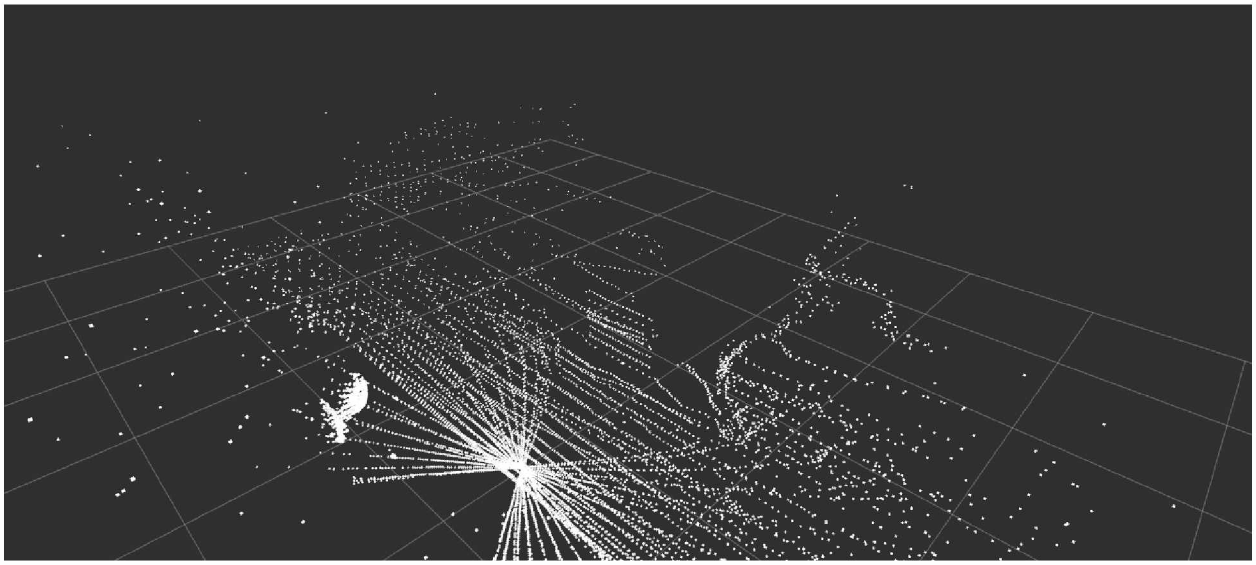


Figure 4.14: VS-LIDAR Outdoor Point Cloud on Overcast Day

The test was performed again with just the VS-LIDAR on a brighter day, when the sensor and marker could be placed in direct sunlight, no cloud or fog cover. The ambient light level measured at the marker was about 120,000 lux, and the VS-LIDAR was placed the same distance away as before. Figure 4.15 shows the resulting point cloud, with the sensors being shaded but with the rest of the surroundings illuminated. Based on visual inspection, the number of erroneous data points didn't seem to increase from the previous outdoor trial, and while the LIDARs were able to see the pavement and some of the surrounding snowbanks, the marker was simply too bright for the sensors to detect. Of greater negative impact, the LIDARs would shut off their laser emitters when hit with direct sunlight, and they wouldn't come back on until the whole LIDAR sensor was reset. At some point, the LIDAR's microcontroller must decide that its laser isn't powerful enough to overcome the ambient light and that it might as well give up. Note, this behavior was replicated in the lab by shining an intensely bright light at the LIDAR, so it is not caused by some special property of sunlight.

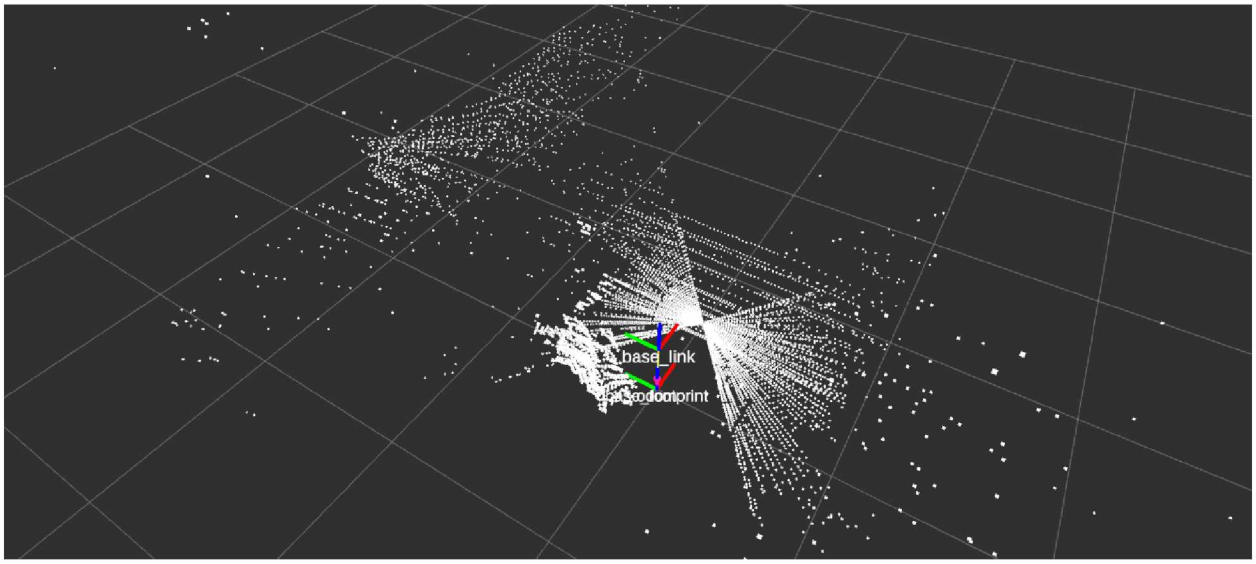


Figure 4.15: VS-LIDAR Outdoor Point Cloud in Direct Sunlight

A final outdoor test compared the VS-LIDAR with the Kinect v2. As before, the marker was placed 1.5 meters away from each sensor, and the illuminance at the target was measured to be about 180,000 lux. This setup is shown for the VS-LIDAR in Figure 4.16 and for the Kinect v2 in Figure 4.17, with Figure 4.18 showing the Kinect's field-of-view. A strip of cardboard was placed behind the front-sweeping LIDAR to shield it from direct sunlight. Figure 4.19 shows the resulting cloud for the Kinect v2 and Figure 4.20 shows the resulting cloud for the VS-LIDAR. Both sensors were for the most part unable to see the wall or the marker as they were too bright, though the Kinect was able to see a small portion of the wall shaded by the marker and some of the marker's base. It also saw some of the nearby ground. The LIDARs, on the other hand, were able to see much more of the ground further away that the Kinect, consistently getting scans up to three meters away (each grid cell in the cloud figure is one square meter).

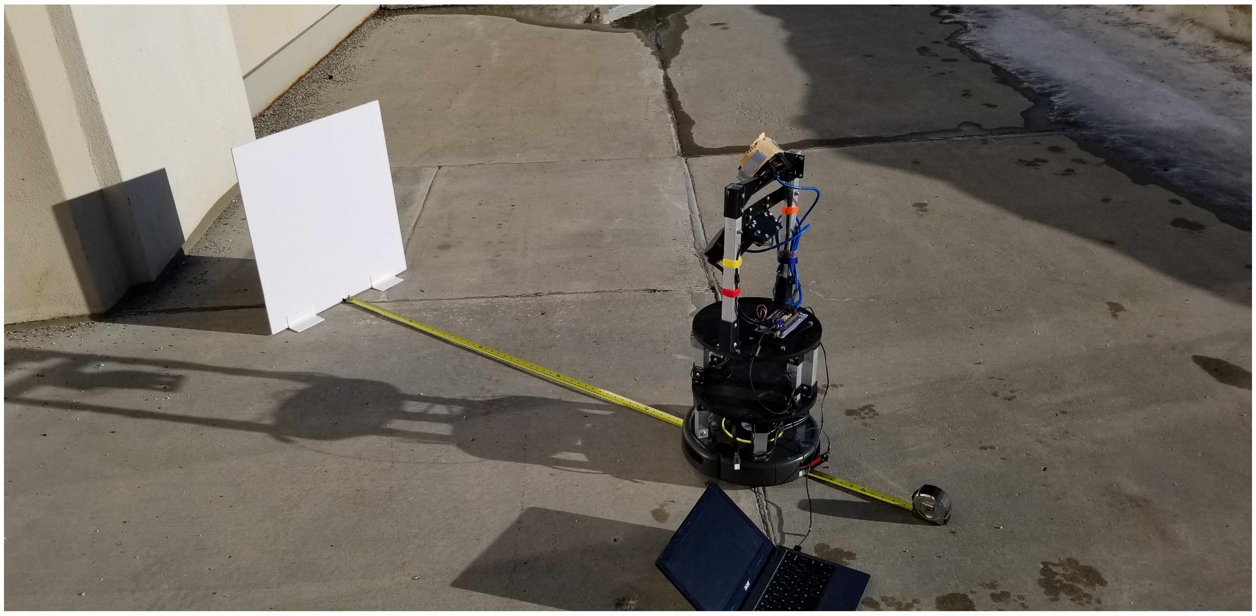


Figure 4.16: Outdoor Perception Test #3 VS-LIDAR Setup

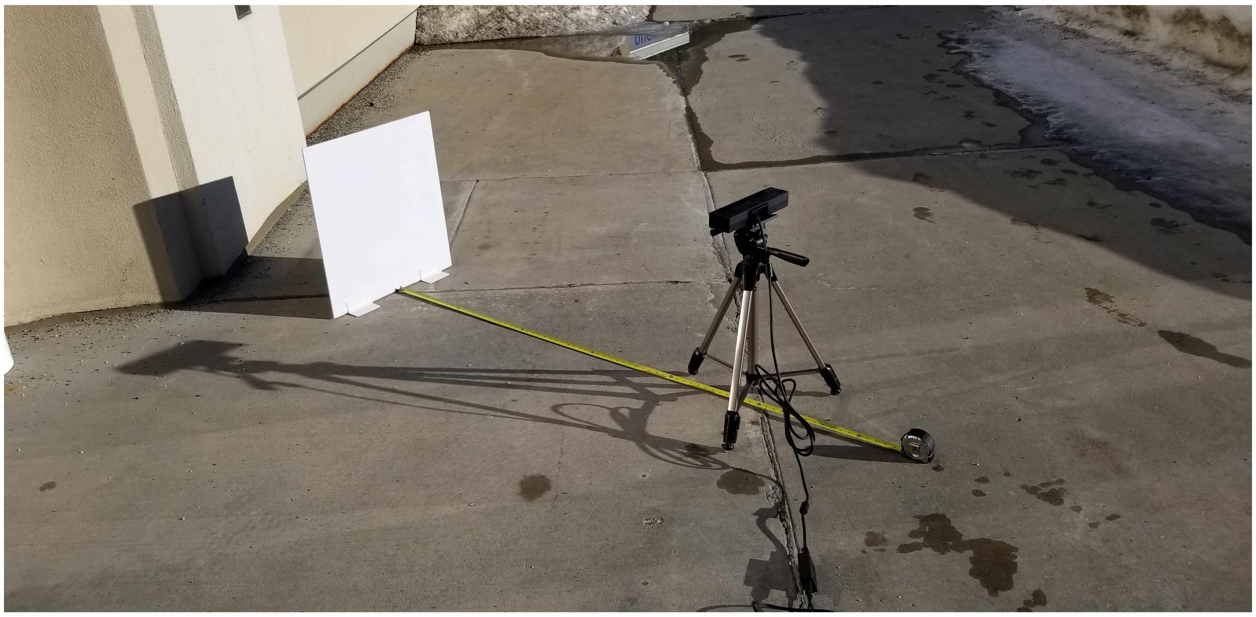


Figure 4.17: Outdoor Perception Test #3 Kinect v2 Setup

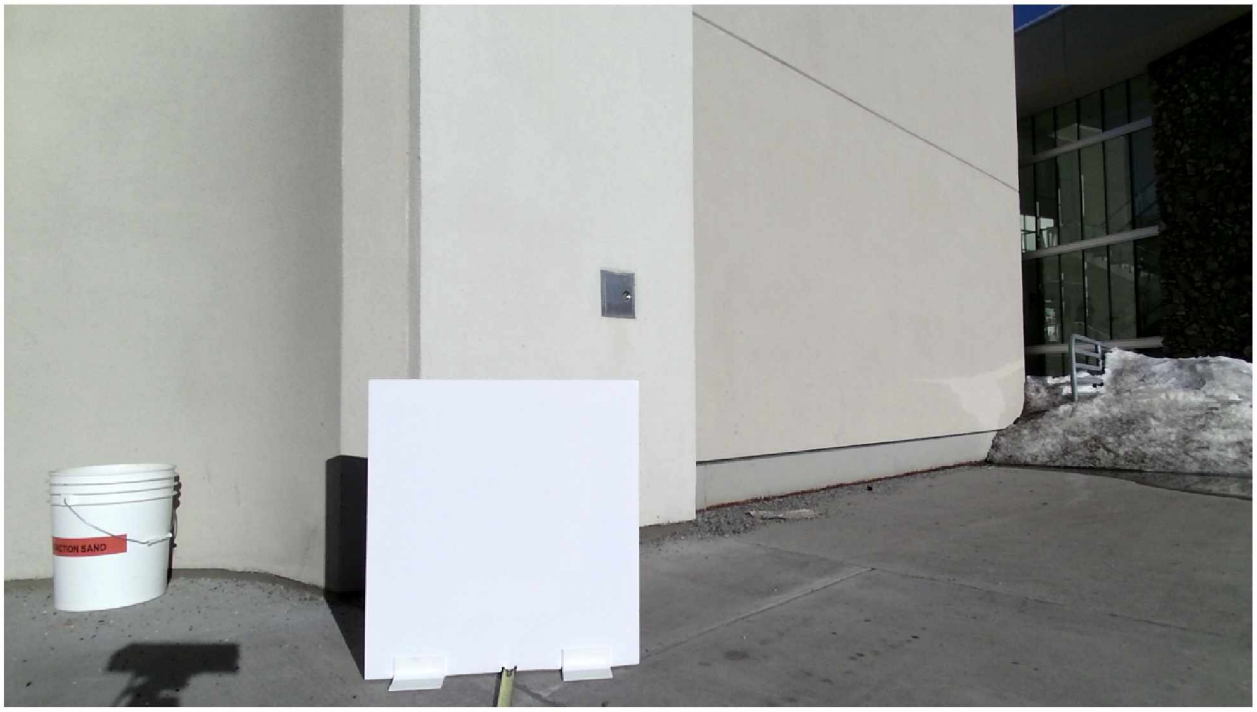


Figure 4.18: Outdoor Perception Test #3 Kinect v2 View From Color Camera

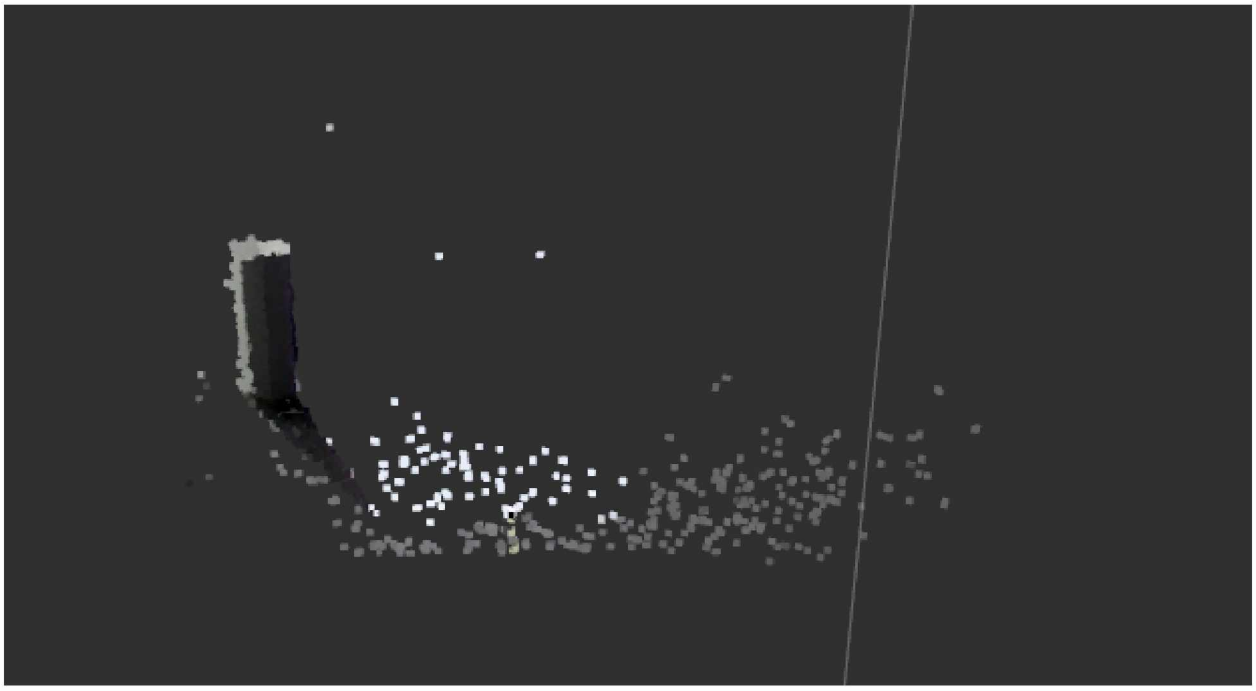


Figure 4.19: Outdoor Perception Test #3 Kinect v2 Cloud

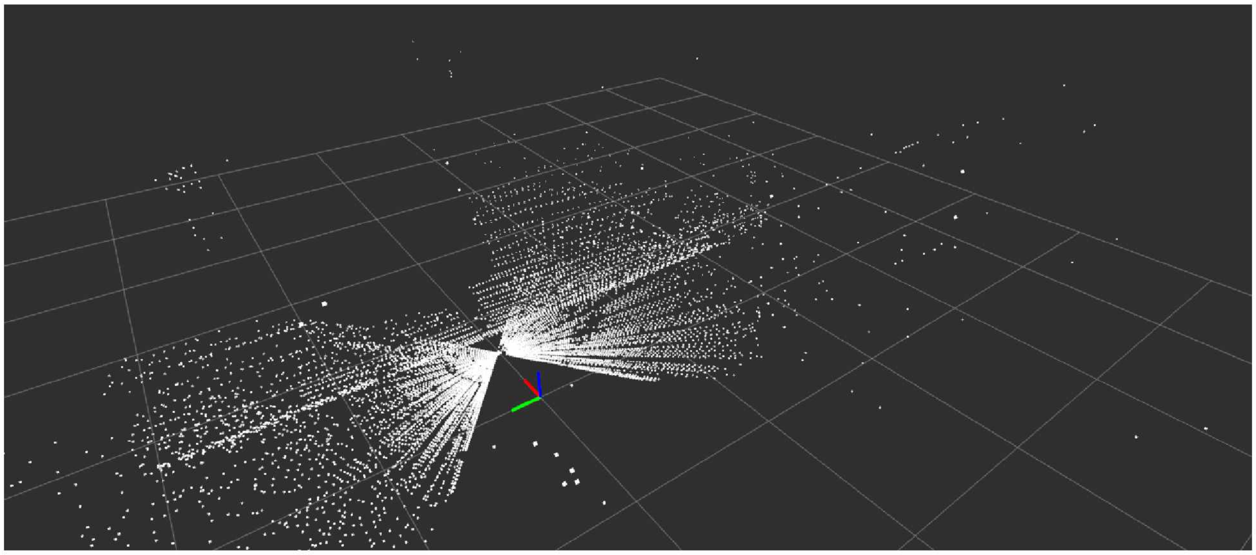


Figure 4.20: Outdoor Perception Test #3 VS-LIDAR Cloud

Thus, the performance of the VS-LIDAR is degraded less in the presence of sunlight than cheaper structured-light and time-of-flight depth cameras. However, it is unable to detect super-bright objects, and the sensors shut off if they are hit by direct sunlight. Curious if the second problem could be solved by giving the sensors a "pair of sunglasses," a quick test was performed where a piece of cut up sunglasses lens was placed over the laser emitter. This prevented the LIDARs from shutting down when hit with direct sunlight, but it also reduced the amount of usable data points, and the curve of the lens distorted the point cloud. So this shortcoming seems solvable with optical shielding, but it appears a cheap, quick fix would be too suboptimal, so the covering will probably have to be designed around the specific properties of the laser light.

4.8 Distance Measurement Comparison

The purpose of this test was to compare the depth measurement accuracy of the VS-LIDAR, Kinect v1, and Kinect v2 as a function of distance. The same white marker from the sunlight tests was placed in front of each sensor at distances ranging from 0.25 m to 4.0 m in 0.25 m increments (indoors), the measured distance was recorded, and the maximum absolute error (MAE) between the measured distance and the actual distance was calculated. Figure 4.21 shows the setup for this test for the VS-LIDAR with the marker placed at a distance of one meter away, with a measuring tape being used to obtain the ground truth for the measurements. The results are shown in Table 4.2 and Figure 4.22. As expected, the VS-LIDAR starts out strong at short distances as it uses triangulation, with the Kinect v1 and Kinect v2 unable to detect the marker until they were 0.5 m and 0.75 m away, respectively. However, after about 1.5 m the accuracy of the VS-LIDAR started rapidly growing, to where at a distance of 4 m the measured distance was 30 cm greater. This is in contrast to the Kinect v1, whose error started picking up around 2.75 m and was 7 cm off at 4 m, and the Kinect v2, which was within 1 cm of the actual distance the whole time.

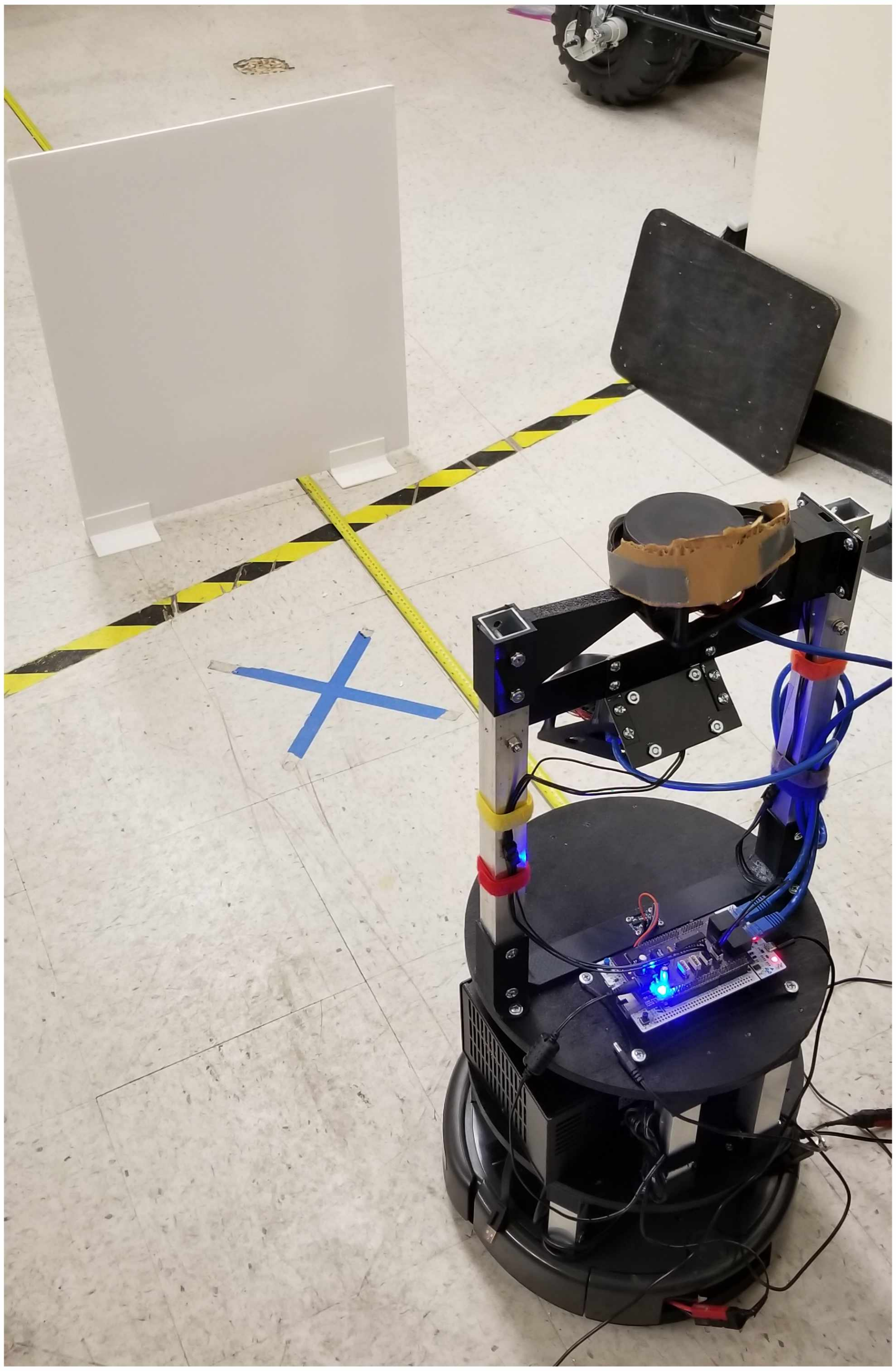


Figure 4.21: Distance Measurement Accuracy Test Setup

Table 4.2: Distance Measurement Accuracy for VS-LIDAR and Kinect v1, and Kinect v2

	VS-LIDAR		**Kinect v1**		**Kinect v2**	
Distance (m)	**Meas. (m)**	**Error (m)**	**Meas. (m)**	**Error (m)**	**Meas. (m)**	**Error (m)**
0.25	0.25	0.00	-	-	-	-
0.50	0.50	0.00	0.50	0.00	-	-
0.75	0.75	0.00	0.74	0.01	0.75	0.00
1.00	1.00	0.00	0.99	0.01	0.99	0.01
1.25	1.25	0.00	1.24	0.01	1.24	0.01
1.50	1.51	0.01	1.50	0.00	1.49	0.01
1.75	1.78	0.03	1.74	0.01	1.75	0.00
2.00	2.05	0.05	2.00	0.00	2.01	0.01
2.25	2.32	0.07	2.25	0.00	2.26	0.01
2.50	2.59	0.09	2.50	0.00	2.50	0.00
2.75	2.87	0.12	2.75	0.00	2.75	0.00
3.00	3.17	0.17	3.02	0.02	3.00	0.00
3.25	3.47	0.22	3.28	0.03	3.25	0.00
3.50	3.77	0.27	3.52	0.02	3.51	0.01
3.75	4.04	0.29	3.80	0.05	3.76	0.01
4.00	4.30	0.30	4.07	0.07	4.01	0.01

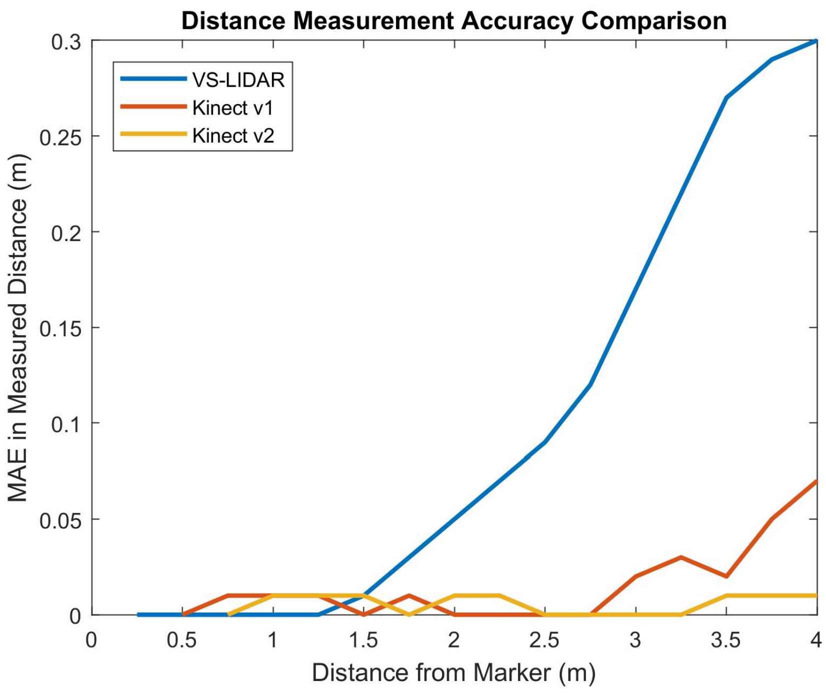


Figure 4.22: Distance Measurement Accuracy for VS-LIDAR, Kinect v1, and Kinect v2

Thus, while the VS-LIDAR is able to see obstacles much closer to itself than the Kinect v1 and v2, it has worse accuracy at maximum range. However, after measuring the distance with the VS-LIDAR in several more trials, the error as a function of distance was found to be very consistent. Since it is known that the error in distance measurement for triangulation LIDARs is proportional to the square of the distance, the error was modeled with a second-order polynomial using MATLAB. Using this model an equation was obtained for estimating the actual distance given the measured distance, where d_{meas} is the measured distance, $error$ is the estimated error in that measurement, and d_{est} is the rectified distance estimate:

$$d_{est} = d_{meas} - error = -0.029(d_{meas})^2 + 1.0361d_{meas} - 0.0057. \tag{4.1}$$

The *xv11_raw2scan_wtf* node was modified to rectify the distance measurements with this model, then a new set of distance measurement trials were performed for evaluating this rectification. The results are shown in Table 4.3. One can see that the rectified distance measurements are much more accurate, and that while the error still starts to climb towards the end, the measured distances are much closer to the actual distances than before.

Table 4.3: Distance Measurement Accuracy for VS-LIDAR after Error Modeling

Distance (m)	**Meas. (m)**	**Error (m)**
0.25	0.25	0.00
0.50	0.50	0.00
0.75	0.76	0.01
1.00	1.00	0.00
1.25	1.25	0.00
1.50	1.50	0.00
1.75	1.75	0.00
2.00	2.00	0.00
2.25	2.25	0.00
2.50	2.50	0.00
2.75	2.75	0.00
3.00	3.00	0.00
3.25	3.24	0.01
3.50	3.47	0.03
3.75	3.72	0.03
4.00	3.96	0.04

To summarize the results of these tests, the VS-LIDAR system is much more capable of scanning nearby objects than the Kinect sensors. However, the error of the distance measurements quickly increases as the distance increases, especially in comparison to the Kinect v2, which had practically no error over the range of measured values. This error was found to be deterministic, though, so the distance measurements were easily rectified with a second-order polynomial function. Even without rectifying though, having a bit of error in the measurements at long range is not that big of a problem. Long range accuracy is only necessary for robots that move quickly; the VS-LIDAR is designed for slow moving robots, so as it gets closer to obstacles it will have time to rescan them and get a more accurate distance reading. At the cost of some long range accuracy, the VS-LIDAR is able to detect obstacles much closer to it than the Kinect sensors, which is important for safely navigating through the environment.

Chapter 5 Case Study: Adaptive Scan Dithering

5.1 Introduction

Throughout the development of the VS-LIDAR system, there has always been the question of what panning speeds and rotation limits should be applied to each LIDAR to obtain an optimal balance between the system's real-time capabilities and the 3D spatial coverage of its point clouds. Additionally, it is clear that the stochastic scan dithering is suboptimal. We wondered if there was an intelligent way to dither the sensors, such that consecutive scans of the same region would spread the scanned points around more evenly. During a panning cycle, the sensor would hit regions that were missed in the previous panning cycle, guaranteeing an increase in point cloud resolution and decreasing the chances of failing to see a small obstacle. We were also curious if this intelligent dithering could be used by the robot while moving in addition to while stationary. It is easier to quantify improvements due to dithering while stationary, as without it scan lines from consecutive panning cycles will tend to overlap one another, but it is much less obvious when the robot is moving, since its perspective of the environment and its distance to obstacles changes constantly.

In the book *Topographic Laser Ranging and Scanning: Principles and Processing*, the authors illustrate the effectiveness of a specific dithering pattern when performing aerial surveys using a LIDAR-equipped unmanned aerial system (UAS) [35, pp. 150]. This dithering not only helps the data points be more equally distributed, but also improves the detection of powerlines and building edges. But finding an effective dithering pattern for ground-based systems is much less straightforward. Consider the long-distance, birds-eye view that an aerial surveyor enjoys; from the top-down, the UAS has an approximately 2D perspective, so dithering patterns need only be two dimensional. For a ground-based robot, however, a 2D dither pattern is no longer sufficient, as a robot's perspective of features in the environment can change drastically as it is allowed to move about. Thus, dithering 3D point cloud data for a system moving through its environment rather than above it requires a movement-aware dithering algorithm that is adaptable to significant changes in perspective.

Neural networks are systems that are loosely modeled on how the human brain works, and research into them is very active because they can theoretically be used to model any function. They consist of a collection of nodes called neurons that are connected to one another. Neurons can have one or more numerical inputs, and each input is assigned a weight. Inputs are multiplied by their corresponding weight, then summed together to obtain a single numerical result. This result is then modified using some activation function (of which there are many different types) and then output by the neuron. Most neural networks consist of layers of neurons, where the outputs of neurons in one layer feed forward into the inputs of neurons in the next layer. Each of these networks has an input and output layer, with any number of layers in the middle referred to as hidden layers.

Research into neural networks has been waxing and waning in popularity since it first began in 1943 [36], and up until recently the community at large thought that neural networks had little prospect of being useful for artificial intelligence. But in 2012, researchers at the University of Toronto trained a deep convolutional neural network to classify 1.2 million images from the ImageNet LSVRC-2010 contest into 1000 distinct classes [37]. The network was able to provide the correct classification for the images on the first try 62.5% of the time, and the correct classification was within the top five best guesses 83% of the time. They entered their network in the ImageNet LSVRC-2012 contest, and the correct answer was in the top five best guesses 84.7% of the time, considerably higher than the next best competitor at 73.8%; so much so, in fact, that their results led to a renaissance in using neural networks in artificial intelligence research. This revival has seen the use of large neural networks being branded as "deep learning."

Since then, neural networks have made their way into many different fields and industries. For example, Google bought the technology rights to the University of Toronto's image classifier and used it to automatically tag photos uploaded by users of their Google+ social media platform [38]. Google also employs a different type of neural network known as a recurrent neural network for speech and text recognition [39]. Recurrent neural networks, as opposed to convolutional neural networks, have memory, making them useful for finding patterns in sequences of inputs. In general, convolutional neural networks have become popular for image processing and recurrent neural networks have become popular for processing text and speech.

These recent successes for neural networks have been made possible in part by the vast quantity of data that now exists to train them. But no such data exists for training robots to perform complex tasks, such as playing ping-pong or flipping pancakes, so researchers have to rely on other methods to teach robots how to behave. A popular method of doing so is called reinforcement learning, where robots learn to perform a task based on trial-and-error, much like humans do, and good behaviors are reinforced by rewards. By learning how to maximize these rewards, the robots ideally learn how to best perform their tasks. Fueled by the surge in popularity of deep learning, researchers at Google DeepMind caused another groundbreaking advancement in the field with their paper *Human-level control through deep reinforcement learning* [40], where they combined deep learning with reinforcement learning to train a neural network to play 49 Atari 2600 games based on just the pixels on the screen and the total score. Their neural network significantly outperformed earlier reinforcement learning game playing algorithms and reached a level comparable to professional human players.

Considering these recent successes, we were curious if deep reinforcement learning could be used to teach a neural network to adaptively dither the LIDARs. Given the current state of the robot and the coverage of the point cloud, this adaptive scan dithering neural network would periodically determine panning parameters that increased unique point cloud spatial coverage while not causing an unacceptable decrease in the real-time capabilities of the system. In the end, the neural network was unable to adaptively dither the sensors in a meaningful way, but it is believed that these efforts provide useful insights into this problem.

5.2 Adaptive Scan Dithering Process Overview

To reduce the scope of the problem, efforts focused only on dithering the rotation speed of the left-to-right panning LIDAR. Only dithering the rotation speed of a single LIDAR, not the pair, and not dithering the rotation limits results in a significant decrease in the number of outputs the neural network is required to have. As seen previously, the left-to-right panning LIDAR gets a fairly dense scan of the floor surrounding the robot, much more so than the front panning LIDAR. It is easier to visualize the effect of dithering if the points lie on a planar surface, as it reduces the problem down from three dimensions to two. Since the floor is typically planar, the point clouds from the left-to-right sweeping LIDAR simplify the process of evaluating the dithering.

Figure 5.1 provides a basic overview of the process the adaptive dithering neural network goes through to determine which speed to pan the LIDAR at and to train itself on how to make these determinations, and this network was designed with guidance from [41]. This process happens iteratively, with each iteration being triggered by the end of a panning cycle. Starting on the left-hand side of the figure, the network receives as input the state of its environment, consisting of the average linear and rotational velocities of the robot during the current panning cycle, the panning speed during the current cycle, and the point cloud coverage metric computed using the cloud from the previous cycle combined with the cloud from the current cycle. The network has an output associated with each panning speed the LIDAR is allowed to take, where each output is a number known as the Q-score for that action. The Q-score is the network's estimate of the sum total of all future rewards that will be gained by performing the associated action in the current state, assuming all subsequent actions are ideal. Thus, the action with the highest Q-score is what the network thinks is the best course of action given the input. Allowable speeds range from 2.5% of full speed (2.85 RPM) to 25% of full speed (28.5 rpm) in 2.5% increments, giving the network fine control over the speed while preventing the LIDAR from panning too fast or too slow. The goal of the network is to learn an optimal LIDAR dithering policy such that, given the current state of the environment, it can predict which speed to pan the LIDAR at to balance between panning speed and coverage.

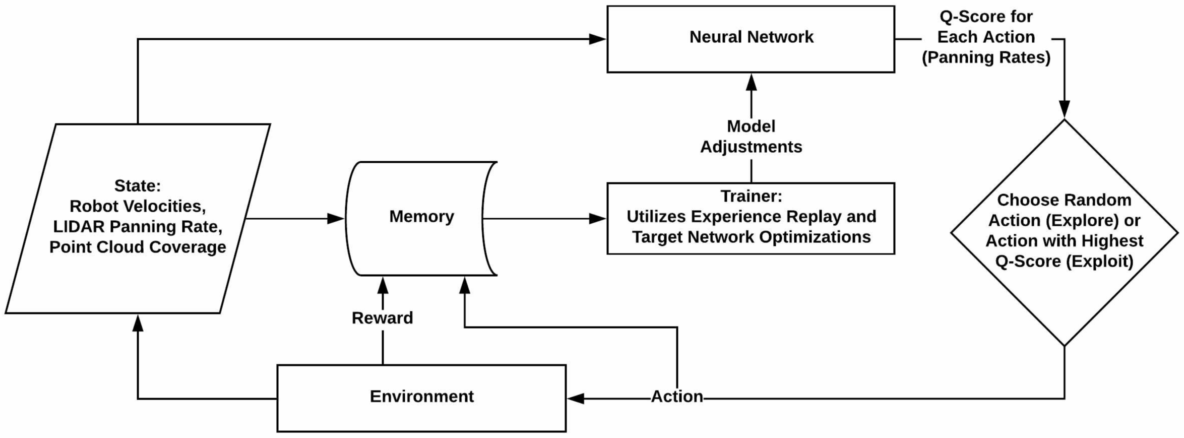

Neural Network
Q-Score for Each Action (Panning Rates)
Model Adjustments
State: Robot Velocities, LIDAR Panning Rate, Point Cloud Coverage
Memory
Trainer: Utilizes Experience Replay and Target Network Optimizations
Choose Random Action (Explore) or Action with Highest Q-Score (Exploit)
Reward
Environment
Action

Figure 5.1: Adaptive Scan Dithering Process Loop

With these probabilities in hand, the network then decides whether to choose a random action (referred to as exploration), or to choose the action with the highest associated probability (referred to as exploitation). The rate at which the network explores vs. exploits is controlled by the hyperparameters ϵ, ϵ_{decay}, and ϵ_{min}, where hyperparameters are parameters set before training which both describe the network and control the learning process. ϵ is the probability of performing a random action, and always starts at one. After each iteration, ϵ is set equal to $max(\epsilon * \epsilon_{decay}, \epsilon_{min})$. This ensures that the network explores heavily in the beginning to discover which actions work well for a given situation, but that as time goes on it is more likely to prefer the good actions it has already discovered. However, setting a minimum value for ϵ such that random actions still have a reasonable chance of occurring means the network will always be searching for better solutions.

Once the network has chosen which action to take, it performs that action and waits for the next panning cycle to complete. When this occurs, it is ready to evaluate the result of its action. It calculates an immediate reward for the action with the new point cloud metric and the speed at which it panned the LIDAR. This iteration is then stored as a sample in memory, consisting of the old state of the environment, the action taken, the reward, and the resulting new state. The cycle then begins anew with the new state being fed into the neural network. After enough samples have been collected, training the neural network begins. At the end of each cycle, a batch of samples is randomly selected from the memory. Using a batch of samples makes the training go more quickly, and using random samples instead of the most recent ones helps prevent network biasing caused by the fact that consecutive samples are likely to be highly correlated. The technique of training with random samples like this was developed by DeepMind and is referred to as experience replay.

Now, the actual training utilizes another optimization developed by DeepMind, which improves training convergence by using a target network in addition to the regular network. Both networks have identical structure, but where the regular network predicts what speed to pan the LIDAR at, the target network predicts what panning speed the network should predict. For each sample in the batch, the target network predicts which action to take given the old state. The target model then determines the overall reward for the action, by first predicting the maximum Q-score of the new state, which represents the total expected future rewards, then by discounting this reward by the hyperparameter γ and adding it to the immediate reward gained from the action, such that $r_{total} = r_{immediate} + \gamma * max(Q_{new})$. This is referred to as the Bellman Equation in literature, and the discount is applied to future rewards to account for non-determinism in the environment; the more random the environment is, the more future rewards should be discounted.

Finally, the old state and the action/reward predicted by the target network are used to train the regular network. Stochastic gradient descent is used to optimize the weights of the network, and how much the weights are optimized for each batch is controlled by the learning rate hyperparameter r. A lower value of r is preferred, because too high a value can cause the optimizer to overshoot, which can prevent the network from converging. After the regular network has been trained with the batch of samples, the system checks how many cycles it has been since the last time the target model was updated, and if it has been long enough the weights of the regular network are copied over to the target network. The number of iterations between updates is controlled by the hyperparameter τ. Using the target network and only periodically updating it helps prevent divergence and oscillations in the dithering policy.

5.3 Coverage Metrics

By far the hardest part of working on the adaptive scan dithering was coming up with a satisfactory metric for determining the 3D spatial coverage of the point cloud, and unfortunately one was never discovered that was quite good enough. The requirements of the metric were that it would classify the coverage in a way that was robust to changes in the environment and encouraged the network to tend towards keeping the points spread about. Since the point of dithering is to spread out points that would be otherwise clumped together, the first metric tested utilized the standard deviation in the Euclidean distance between points in the cloud.

The input to the algorithm is the last two panning cycles worth of point clouds, combined into a single cloud, so that the LIDAR has had the chance to scan the same region twice and can therefore determine if the dithering was effective or not. With the point cloud taken as a whole, it is virtually guaranteed that points will not be evenly spread throughout, since the sensors will hit obstacles and regions with their own distinct surfaces and edges, and at different distances. For example, one can see in Figure 4.15 how the snowbank and the ground have distinctly different point distribution patterns, and the standard deviation would be thrown off if the empty space between the two distinct features was included. The solution to this is to first split the point cloud up into unique regions and features using the Euclidean Cluster Extractor from the Point Cloud Library [42]. Point patterns for clusters are more consistent than the point pattern of the whole cloud, and getting rid of large empty distances improves the accuracy of the standard deviation computations.

For each point in each cluster, the standard deviation between that point and all its nearest neighbors (specified either by a certain number of nearest neighbors or nearest neighbors a certain distance away) is calculated. These are then used to determine the average standard deviation in Euclidean distance for a point and its nearest neighbors for the entire cluster. The idea here is that the lower the average standard deviation is, the more ideally the points are spread out, so the dithering AI works to minimize this value. However, in testing any useful information in this metric seemed to get drowned out, with it hardly changing, regardless of what the speed of the LIDAR was or if any form of dithering was active or not. This is believed to be the result of the averaging, and the fact that the point resolution along the laser's center of rotation is significantly higher than the resolution about the servo's center of rotation.

The second metric developed and tested is significantly more involved. Essentially, starting with the point clouds from the previous and current panning cycles as before, it identifies the largest planar region that has been scanned, constructs a coverage map over that region, then computes the ratio of the scanned area to the total area of the region. Step by step:

1. Start with the point clouds from the previous panning cycle and the current panning cycle, combining them into one accumulated cloud.

2. From that cloud, extract the largest planar component, which for the left-to-right panning LIDAR is most often the floor. For aforementioned reasons, it is easiest to determine the need for dithering by examining planar surfaces.

3. Filter out statistical outliers so they cannot throw off the calculation of the region's size in upcoming steps.

4. Compute the convex hull of the planar cloud to get a bounding polygon around the cloud.

5. Draw a bounding rectangle around the bounding polygon, and use it discretize the region into a grid with a set resolution.

6. Iterate through the grid. Count the number of cells occupied by one or more points, then count the number of cells that are empty and also within the convex hull. Add the two to get the total number of cells within the hull.

7. Compute the coverage metric as the number of occupied cells divided by the number of cells within the hull. This metric ranges from 0 to 1, with 0 representing no coverage and 1 representing full coverage.

8. Repeat this process after the next panning cycle is completed.

The resulting coverage map for one iteration, with a grid resolution of 2.5 cm, is shown in Figure 5.2, where the occupied cells are colored black, the empty cells within the hull are dark gray, the cells outside the hull are light gray, and the bounding polygon for the region is outlined in green. This metric has promise, being much more sensitive to changes in panning speed than the previous one, and was used the most when attempting to train the dithering AI. However, in its current state it proved to be unstable. Figure 5.3 shows how this metric varies over time (in seconds) with the LIDAR panning at a fixed speed, with the robot stationary and the environment stationary. Under these conditions, the coverage metric should be relatively constant, but it in-fact can vary by more than 15%. The cause in this instability is due to variation in the size of the convex hull, caused by the furthest reaches of the planar cloud extending and contracting. Some of this extending and contracting was caused by the statistical outlier filtering, but without the filtering outlier points throw the region's shape off much more. Drawing a simple, static bounding box around the cloud would achieve more consistent results while the robot is stationary, but would fall apart if the robot moved around at all, which is why a more dynamic method of determining the boundary is necessary. Thus, this metric could work for training the neural network, but a more consistent and stable method of outlining the covered region is necessary.

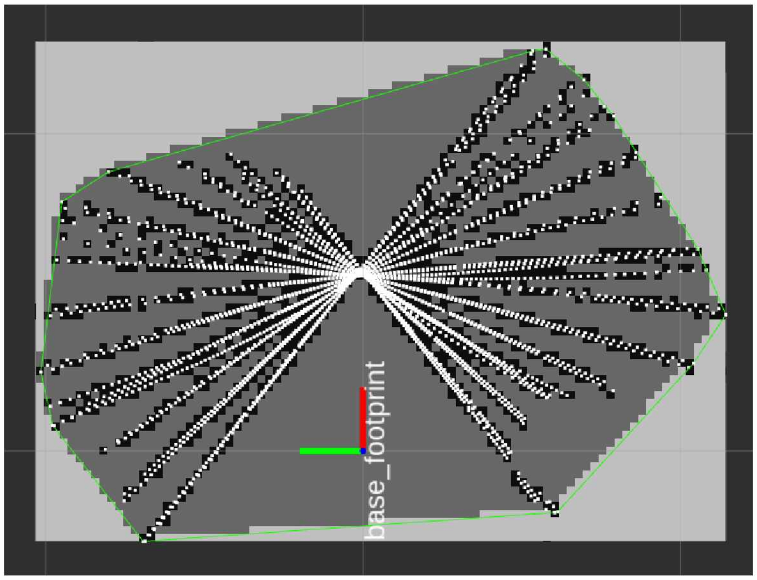


Figure 5.2: Point Cloud Coverage Map for Computing Coverage Metric

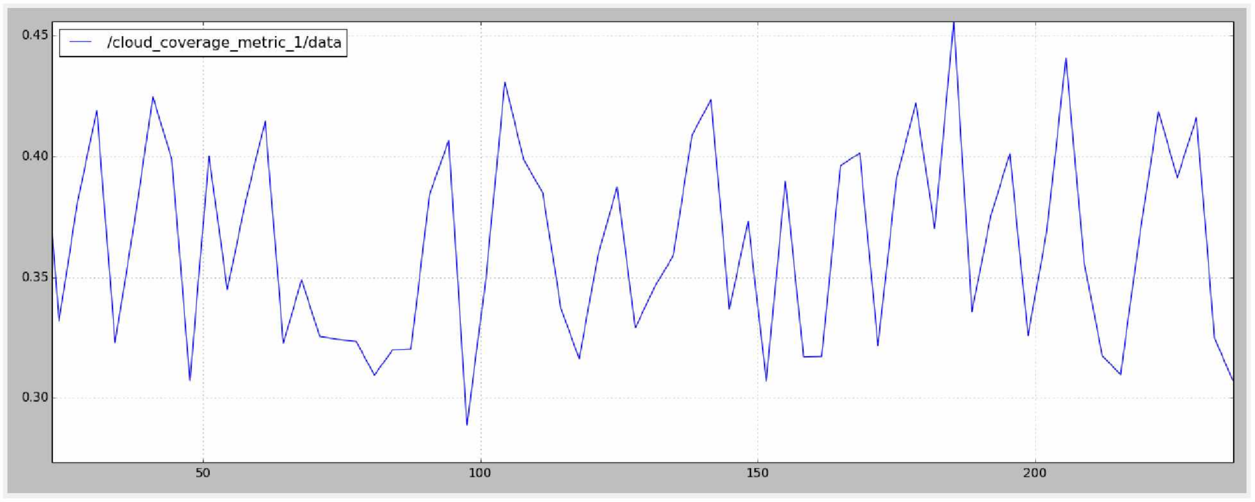


Figure 5.3: Plot of Cloud Coverage over Time with Fixed Viewpoint and Speed

5.4 Reward Function

The reward function provides the way for the network to determine the immediate reward associated with each action, given the current state of the environment. In real-world environments, the reward can't usually be expressed as simply as the total score at the end of a game, so careful design of the reward function is critical. As the goal of the neural network was to balance out point cloud spatial coverage and LIDAR panning speed, the reward was a function of these variables, and consisted of two distinct components: a reward for the raw coverage and a discount applied based on the speed of the LIDAR. The raw coverage reward was calculated as $r_{coverage} = coverage^{1.5}$, where coverage ranged from 0 to 1, with 0 being no coverage and 1 being full coverage. This curve, as seen in Figure 5.4 (a), places more weight on higher coverages, but not so much so that sacrificing a bit of coverage for a higher panning speed is unacceptable. The velocity discount is calculated as $v_{discount} = (v_{panning})^{0.5}$, where $(v_{panning})$ ranges from 0.025 to 0.25 as previously mentioned. This curve, as seen in Figure 5.4 (b), discourages the use of very slow panning speeds while making medium speeds more acceptable. The raw coverage reward and the velocity discount are combined to compute the total reward with:

$$r_{total} = 2 * r_{coverage} * v_{discount} = 2 * coverage^{1.5} * (v_{panning})^{0.5}. \tag{5.1}$$

Note that the multiplication by two simply normalizes the reward to be between zero and one, just to make the equation a bit nicer. A much more natural interpretation of the reward function can be seen in Figure 5.5. One can see that higher rewards are given to the system for higher coverages and higher speeds, but that relatively high rewards can still be obtained at medium speeds if it is too

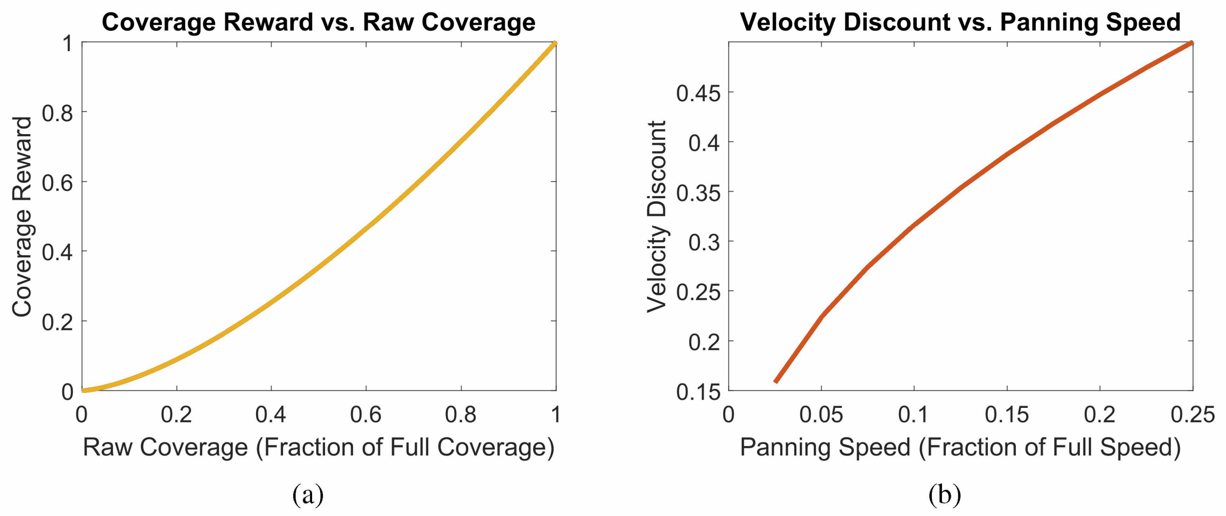


Figure 5.4: Raw Coverage Reward and Velocity Discount Curves

hard or simply impossible for the network to get high speeds and high coverage, which is almost certainly the case.

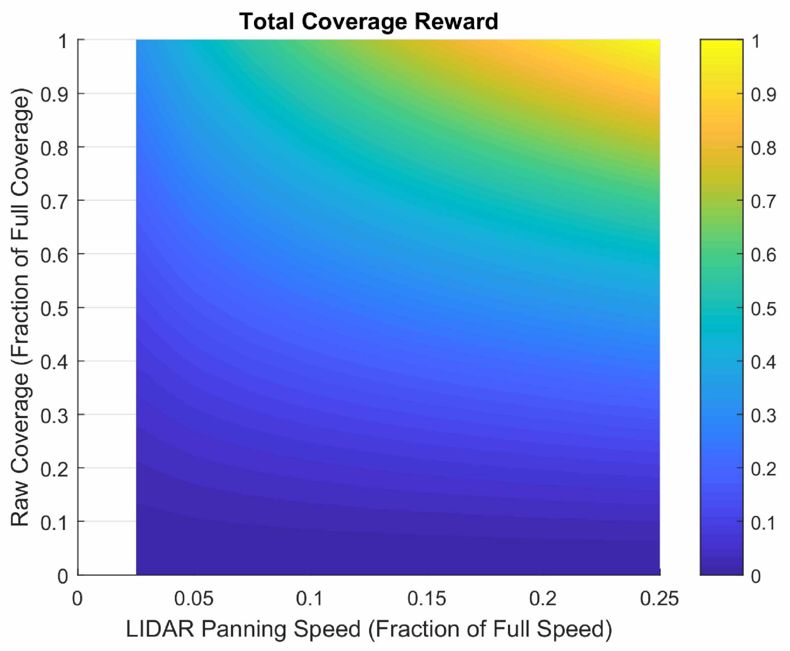


Figure 5.5: Total Coverage Reward as a Function of Cloud Coverage

5.5 Network Configuration

With the intent of the network, the coverage metric, and the reward function established, the next step was to actually implement the network proper. This was done using Keras, a popular high level framework for developing neural network applications in Python [43]. Keras itself only implements an API to make it easier to use existing artificial intelligence software; the actual engine used for the learning was Google's TensorFlow [44]. Designing the network architecture started by defining the input and output layers. The input layer consisted of four neurons, one each for the average linear velocity of the robot, the average angular velocity of the robot, the current panning speed, and the coverage metric. The output layer consisted of ten neurons, one for each panning speed, ranging from 2.5% of full speed to 25% of full speed in 2.5% increments. Setting up the hidden layers was much more arbitrary, since there is no standard set of rules for how these should be setup. Ultimately two hidden layers were used, with 24 neurons in each. All of these layers were fully connected, meaning for each layer, each neuron is connected to all neurons in the next layer. The decision to use only fully connected layers for this network was made because they appear to form the core of most reinforcement learning networks out there, and the low dimensionality of the input did not seem to necessitate anything else. Each layer, except for the output layer, uses the rectified linear unit as its activation function. For inputs greater than zero, the output of this activation function is simply the input, and the output is zero otherwise. This is the simplest non-linear activation function, and is quite popular in deep learning as it can significantly speed up training time over using other non-linear activation functions. The output layer uses the linear activation function, meaning the output of the layer isn't modified in any way. This is a common choice in deep reinforcement learning, since the outputs are Q-scores which could be negative if the network is penalized for performing "bad" actions.

The final step in designing the network was to choose values for its hyperparameters. Table 5.1 summarizes these choices. γ, which controls the devaluation of future rewards, was set to 0.5, as the real-world operating environment of the robot will be random except perhaps in very controlled experiments. The starting value of 1 for ϵ is obvious as the network should always start by exploring. The values of 0.995, 0.01, and 0.01 for ϵ_{decay}, $epsilon_{min}$, and r, respectively, were chosen from a tutorial on deep-reinforcement learning [41]. The value of 15 for τ was chosen mostly arbitrarily, and corresponds to approximately a minute of robot operation per target model update.

Table 5.1: Neural Network Hyperparameters

Parameter	Value	Description
γ	0.5	Devaluation of future rewards.
ϵ_0	1.0	Initial value of epsilon. Controls the probability that the system explores instead of exploits.
ϵ_{decay}	0.995	Epsilon decay rate.
ϵ_{min}	0.01	Minimum value of epsilon.
τ	15	Number of iterations per target model update.
r	0.1	Learning Rate. Controls how fast network weights are optimized.

5.6 Performance and Remarks

To verify the validity of the coverage metric and reward function, most of the testing involved trying to have the network learn a good dithering policy while the robot was completely stationary. This way, one would be able to tell if the AI could improve dithering at all, and if the configuration, coverage metric, and reward function were valid. To test the network setup, it was trained with just the coverage reward (without the velocity discount), and the network was able to learn that coverage could be maximized by panning the LIDAR as slowly as possible (since slower panning increases point density, covering more area). Unfortunately, the network was unable to learn a full optimal dithering policy. With the velocity discount as described above, the network still defaulted to scanning as slowly as possible. Even with testing out various velocity discounts, the network only converged to one fixed speed, certainly not the adaptive dithering we were aiming for.

Throughout this process, some very important lessons about deep learning were learned. For starters, it is a very complicated and time-consuming field to get into. It seems that one would need an entire team of researchers with years of experience in machine learning already, working non-stop to produce anything remotely practical involving deep-reinforcement learning and robotics. Second, if one is wondering if they should use deep learning, the correct answer is probably no more often than not, since deep learning is so time consuming and for a lot of problems, more traditional algorithms can yield adequate performance. Third, one must have what they want their network to do mapped out to the finest detail, and inputs and reward functions must be excruciatingly shaped to promote that behavior. Even then, these inputs and rewards functions can end up promoting behaviors one does not expect, and the required redesign eats up even more time because one must train the network for a long time to even see if changes fix the issue.

Still, in theory we think some form of adaptive scan dithering would be useful, though deep-reinforcement learning is not necessarily the best way to go about it. Most 3D distance sensors rely on a high density of points to avoid missing useful information about the environment, but because this generates so much data, the point clouds more often than not have to be downsampled significantly before processing. An adaptive dithering AI would allow the sensors to scan with much greater efficiency, reducing computational load on control computers. Additionally, reduced scanning requirements for the sensors would allow them to be produced more cheaply, which would speed up the spread of intelligent robotic systems throughout our society. Thus, we think adaptive scan dithering is a topic that deserves further investigation.

Chapter 6 Conclusions and Future Work

6.1 Conclusions

This thesis described the creation of a novel 3D LIDAR system which can be made at a cost of only about $350. Commercially available LIDARs at this pricing level only scan in 2D, with the next cheapest 3D LIDAR system, the Scanse Sweep 3D scanner, costing twice as much while scanning almost four times slower. Also, when compared to other low-cost 3D sensors like the Kinect v1 and v2, the VS-LIDAR is much more capable of scanning sunlit areas, though it does need some sun shielding in the most extreme outdoor cases, and is much more capable of detecting very nearby objects, which is important for safe navigation and obstacle avoidance. Further, LIDAR sensors work as well in the dark as they do in the light, giving the VS-LIDAR an edge over methods which get 3D data using traditional color cameras, which are sensitive to lighting changes. Additionally, data from the VS-LIDAR takes up much less bandwidth and computational resources than data from the Kinect v1 and v2 and color camera depth measurement techniques, allowing lower cost computing hardware to be used and/or freeing up these resources for other tasks. All this, coupled with the VS-LIDAR's wide 3D field-of-view, make it well suited for 3D obstacle detection and navigation in use-cases where obtaining complete real-time views of the environment isn't a priority, as getting a full view takes on the order of seconds. An example use case would be an autonomous extraterrestrial mining robot, which would be operating in a relatively static environment.

Created in support of this system was a set of custom interface hardware, microcontroller firmware, and software packages that allow the VS-LIDAR system to be easily integrated on any robotic platform using the ROS development environment. The widespread popularity of ROS and the large number of tools and libraries incorporated with it, combined with the low cost of the VS-LIDAR, make this system highly accessible to robotics researchers and hobbyists alike. The ability for other ROS nodes or the user to change the panning parameters for each LIDAR on the fly allows the 3D scanning to be tailored for specific platforms or environments.

Additionally, a mobile testing platform for the VS-LIDAR and an example implementation of autonomous navigation using existing ROS packages were created. This implementation used an Extended Kalman Filter to fuse data from wheel odometry and an IMU into a more accurate estimate of the robot's pose, included a camera and an ArUco marker for correcting odometry errors and for identifying points of interest, and used the ROS navigation stack for obstacle avoidance and path planning. Experiments performed with this setup demonstrated the viability of using the VS-LIDAR for autonomous robotic systems.

Finally, a novel method of increasing the scanning efficiency of rotating 2D LIDARs using deep-reinforcement learning was investigated. It was noted that during consecutive panning cycles scan lines tended to line up with one another while the robot was stationary, and that when the robot was moving optimal scanning patterns are not obvious as the robot's perspective of obstacles changes. The complexity and variety of 3D environments exacerbates this problem. Adaptive scan dithering was proposed as a solution to this problem, whereby a deep-reinforcement learning neural network is used to learn optimal dithering patterns which balance spatial coverage with panning speed, given the current panning speeds, the robot's motion, and a coverage metric which compares point clouds from consecutive panning cycles to evaluate how well described the 3D space is. No such existing metric was found, so several were developed and tested out. Unfortunately, the neural network was unable to produce significant results; however, this research highlighted a new area for investigation.

To summarize:

- Designed a 3D LIDAR system that can be constructed for \$350, four times faster and half the cost of the next cheapest available 3D LIDAR system.
- Created custom hardware, firmware, and software packages allowing the system to be plug-and-play with ROS, making the system accessible to researchers and hobbyists alike.
- Performed experiments demonstrating the system to have better performance at close range and in sunlit environments than other low-cost 3D sensors, though sun-shielding optics for the LIDARs need to be designed to protect them from direct sunlight.
- Demonstrated the advantages of a specific two LIDAR configuration over just using one LIDAR.
  - One LIDAR scans with a wide view in front of the robot quickly to detect near and far obstacles alike and to deal with dynamic obstacles.
  - One LIDAR scans to the sides of the robot in great detail, providing a better view of obstacles immediately surrounding the robot, allowing it to detect small obstacles potentially missed by the other LIDAR and allowing for quicker planning of paths involving rotations.
- Created an example implementation of autonomous navigation by combining existing ROS packages, demonstrating the VS-LIDAR system being used for 3D obstacle avoidance.
- Investigated a novel method of increasing the scanning efficiency of rotating 2D LIDARs termed "adaptive scan dithering," identifying it as a new area for future research.

6.2 Future Work

Though this thesis performed much of the required legwork in developing and testing a low-cost 3D LIDAR system, there is still further work to do. First, this system was designed with NASA's Robotic Mining Competition specifically in mind, with the work on navigation also serving as preparation for the VS-LIDAR's use in autonomous mining. Thus, work that will occur immediately in the future is transitioning the VS-LIDAR from a laboratory setting to a more natural setting with 3D obstacles like rocks and craters. This will require smarter obstacle detection to be performed than simply removing the ground plane, and that the issues with ROS' 3D obstacle maps be solved. Doing these things will demonstrate with greater efficacy the practicality of using the VS-LIDAR for mission-oriented ground robots.

Also, a great opportunity for future work lies in adaptive scan dithering. Developing a method to intelligently scan an environment would be a great boon for robotics, as it would allow sensors to collect less data without a marked decrease in performance. This would allow sensors to be made more cheaply and make it easier to use their data, allowing for their incorporation on a wider assortment of autonomous robots, thus speeding up the proliferation of intelligent systems throughout our society. Future work would have to determine whether deep learning is indeed the way to go, or if their is perhaps some different method that would work better.

6.3 Lessons Learned

The execution of this thesis was a great learning experience regarding the creation of robotic systems. It covered every facet, from the design of mechanical and electrical components, to creating low-level device controllers, to working with higher-level robot control algorithms. It also served as a great crash course in the exploding field of deep learning, and illustrated the great amount of time and effort that goes into making these intelligent systems. Finally, it highlighted the problems that engineers face every day, such as dealing with random, mysterious bugs, valuable experience that is certain to make one more careful and competent when designing systems in the future.

References

[1] H. of Alexandria, *The Pneumatics of Hero of Alexandria, from the Original Greek. Tr. and ed. by Bennet Woodcroft*. London: Taylor, Walton, and Maberly, 1851, Retrieved from the Library of Congress. [Online]. Available: https://www.loc.gov/item/07041532/.

[2] D. Scharstein and R. Szeliski, "A taxonomy and evaluation of dense two-frame stereo correspondence algorithms," *International Journal of Computer Vision*, vol. 47, no. 1-3, pp. 7–42, 2002.

[3] E. S. Jones and S. Soatto, "Visual-inertial navigation, mapping and localization: A scalable real-time causal approach," *International Journal of Robotics Research*, vol. 30, no. 4, pp. 407–430, 2011.

[4] G. Kamberova and R. Bajcsy, "Sensor errors and the uncertainties in stereo reconstruction," in *Workshop on Empirical Evaluation Methods in Computer Vision, Santa Barbara, California*, 1998.

[5] Open Source Robotics Foundation, Inc., "About - Turtlebot". [Online]. Available: http://www.turtlebot.com/about/. [Accessed March 12, 2018].

[6] K. Qian, X. Ma, F. Fang, and H. Yang, "3d environmental mapping of mobile robot using a low-cost depth camera," in *2013 IEEE International Conference on Mechatronics and Automation (ICMA)*, IEEE, 2013, pp. 507–512.

[7] P. Henry, M. Krainin, E. Herbst, X. Ren, and D. Fox, "RGB-D mapping: Using depth cameras for dense 3d modeling of indoor environments," in *12th International Symposium on Experimental Robotics (ISER)*, Citeseer, 2010.

[8] D. S. O. Correa, D. F. Sciotti, M. G. Prado, D. O. Sales, D. F. Wolf, and F. S. Osorio, "Mobile robots navigation in indoor environments using Kinect sensor," in *2012 Second Brazilian Conference on Critical Embedded Systems*, May 2012, pp. 36–41. DOI: 10.1109/CBSEC.2012.18.

[9] J. Hernandez-Aceituno, R. Arnay, J. Toledo, and L. Acosta, "Using Kinect on an autonomous vehicle for outdoors obstacle detection," *IEEE Sensors Journal*, vol. 16, no. 10, pp. 3603–3610, 2016.

[10] L. Li, "Time-of-flight camera–an introduction," *Texas Instruments technical white paper*, no. SLOA190B, 2014.

[11] R. Horaud, M. Hansard, G. Evangelidis, and C. Ménier, "An overview of depth cameras and range scanners based on time-of-flight technologies," *Machine Vision and Applications*, vol. 27, no. 7, pp. 1005–1020, 2016.

[12] F. Blais, J. A. Beraldin, and S. F. El-Hakim, "Range error analysis of an integrated time-of-flight, triangulation, and photogrammetric 3d laser scanning system," in *Laser Radar Technology and Applications V*, International Society for Optics and Photonics, vol. 4035, 2000, pp. 236–248.

[13] Scanse LLC, "Sweep DIY 3D Scanner Kit". [Online]. Available: http://scanse.io/3d-scanning-kit/. [Accessed April 1, 2018].

[14] B. J. Patz, Y. Papelis, R. Pillat, G. Stein, and D. Harper, "A practical approach to robotic design for the DARPA urban challenge," *Journal of Field Robotics*, vol. 25, no. 8, pp. 528–566, 2008.

[15] C. Urmson, J. Anhalt, D. Bagnell, C. Baker, R. Bittner, M. Clark, J. Dolan, D. Duggins, T. Galatali, C. Geyer, *et al.*, "Autonomous driving in urban environments: Boss and the urban challenge," *Journal of Field Robotics*, vol. 25, no. 8, pp. 425–466, 2008.

[16] I. Shim, D. G. Choi, S. Shin, and I. S. Kweon, "Multi lidar system for fast obstacle detection," in *2012 9th International Conference on Ubiquitous Robots and Ambient Intelligence (URAI)*, Nov. 2012, pp. 557–558. DOI: 10.1109/URAI.2012.6463074.

[17] J. Chen and Y. K. Cho, "Real-time 3d mobile mapping for the built environment," in *ISARC. Proceedings of the International Symposium on Automation and Robotics in Construction*, Vilnius Gediminas Technical University, Department of Construction Economics & Property, vol. 33, 2016, p. 1.

[18] S. Lin, "Making of Maps: The cornerstones", *Google Maps Blog*, Sep. 2014. [Online]. Available: https://maps.googleblog.com/2014/09/making-of-maps-cornerstones.html. [Accessed March 22, 2018].

[19] J. Bohren, R. B. Rusu, E. G. Jones, E. Marder-Eppstein, C. Pantofaru, M. Wise, L. Mösenlechner, W. Meeussen, and S. Holzer, "Towards autonomous robotic butlers: Lessons learned with the PR2," in *2011 IEEE International Conference on Robotics and Automation*, May 2011, pp. 5568–5575. DOI: 10.1109/ICRA.2011.5980058.

[20] B. A. Gebre, H. Men, and K. Pochiraju, "Remotely operated and autonomous mapping system (ROAMS)," in *2009 IEEE International Conference on Technologies for Practical Robot Applications*, IEEE, 2009, pp. 173–178.

[21] K. Konolige, J. Augenbraun, N. Donaldson, C. Fiebig, and P. Shah, "A low-cost laser distance sensor," in *2008 IEEE International Conference on Robotics and Automation*, May 2008, pp. 3002–3008. DOI: 10.1109/ROBOT.2008.4543666.

[22] "XV11 Hacking Wiki", Jan. 2014. [Online]. Available: https://xv11hacking.wikispaces.com/. [Accessed April 3, 2017].

[23] Neato Robotics, *Neato XV Series Robot Vacuum User Guide*, 2013. [Online]. Available: https://www.neatorobotics.com/wp-content/uploads/2015/04/XV-Series-User-Guide_EN-SCH_2013.10.2111.pdf.

[24] ROBOTIS, *AX-12/AX-12+/AX-12A - ROBOTIS e-Manual*, AX-12A datasheet, v1.29.00, 2010. [Online]. Available: http://support.robotis.com/en/techsupport_eng.htm#product/dynamixel/ax_series/dxl_ax_actuator.htm.

[25] Bosch Sensortec, *BNO055 - Intelligent 9-Axis Absolute Orientation Sensor*, BNO055 datasheet, v1.4, Jun. 2016. [Online]. Available: https://ae-bst.resource.bosch.com/media/_tech/media/datasheets/BST_BNO055_DS000_14.pdf.

[26] STMicroelectronics, *Arm Cortex-M7 32b MCU+FPU, 462DMIPS, up to 2MB Flash/512 + 16 + 4KB RAM, USB OTG HS/FS, 28 com IF, LCD, DSI*, STM32F767ZI datasheet, Rev. 6, Sep. 2017.

[27] iRobot, *iRobot Create 2 Open Interface (OI) Specification*, Aug. 2016. [Online]. Available: http://www.irobotweb.com/-/media/MainSite/PDFs/About/STEM/Create/iRobot_Roomba_600_Open_Interface_Spec.pdf?la=en.

[28] R. Siegwart, I. Nourbakhsh, and D. Scaramuzza, *Introduction to Autonomous Mobile Robots*. MIT Press, 2011.

[29] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "ROS: an open-source Robot Operating System," in *ICRA Workshop on Open Source Software*, Kobe, vol. 3, 2009, p. 5.

[30] W. Meeussen, "ROS Enhancement Proposal 105: Coordinate Frames for Mobile Platforms", *ros.org*, Oct. 2010. [Online]. Available: http://www.ros.org/reps/rep-0105.html.

[31] T. Foote and M. Purvis, "ROS Enhancement Proposal 103: Standard Units of Measure and Coordinate Conventions", *ros.org*, Oct. 2010. [Online]. Available: http://www.ros.org/reps/rep-0103.html.

[32] E. B. Dam, M. Koch, and M. Lillholm, *Quaternions, interpolation and animation*. Datalogisk Institut, Københavns Universitet Copenhagen, 1998, vol. 2.

[33] T. Moore and D. Stouch, "A generalized extended Kalman filter implementation for the robot operating system," in *Proceedings of the 13th International Conference on Intelligent Autonomous Systems (IAS-13)*, Springer, Jul. 2014.

[34] S. Garrido-Jurado, R. Muñoz-Salinas, F. J. Madrid-Cuevas, and M. J. Marín-Jiménez, "Automatic generation and detection of highly reliable fiducial markers under occlusion," *Pattern Recognition*, vol. 47, no. 6, pp. 2280–2292, 2014.

[35] J. Shan and C. K. Toth, *Topographic laser ranging and scanning: principles and processing*. CRC press, 2017.

[36] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The Bulletin of Mathematical Biophysics*, vol. 5, no. 4, pp. 115–133, 1943.

[37] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems*, 2012, pp. 1097–1105.

[38] C. Rosenberg, "Improving Photo Search: A Step Across the Semantic Gap", *Google Research Blog*, Jun. 2013. [Online]. Available: https://research.googleblog.com/2013/06/improving-photo-search-step-across.html. [Accessed Jan. 26, 2018].

[39] H. Sak, A. Senior, and F. Beaufays, "Long short-term memory recurrent neural network architectures for large scale acoustic modeling," in *Fifteenth Annual Conference of the International Speech Communication Association*, 2014.

[40] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[41] Y. Patel, "Reinforcement Learning w/ Keras + OpenAI: DQNs", *towardsdatascience.com*, Jul. 2017. [Online]. Available: https://towardsdatascience.com/reinforcement-learning-w-keras-openai-dqns-1eed3a5338c. [Accessed Jan. 19, 2018].

[42] R. B. Rusu and S. Cousins, "3D is here: Point Cloud Library (PCL)," in *IEEE International Conference on Robotics and Automation (ICRA)*, Shanghai, China, May 2011.

[43] F. Chollet *et al.*, *Keras*, 2015. [Online]. Available: https://github.com/keras-team/keras.

[44] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 265–283. [Online]. Available: https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf.