Autonomous Golf Cart:

LiDAR Obstacle Detection

Oklahoma State University
ECEN Capstone Design Team
Spring 2016

Katie Hobble
Dante Xiang
Russell Morrow
David Harp

Abstract

This project is intended to support the Mechanical and Aerospace Engineering students working under Dr. Girish Chowdhary in the Distributed Autonomous Systems (DAS) Lab as they continue to develop an autonomous golf cart. The ECEN Capstone Design Team has been tasked with developing software to connect existing systems with a Velodyne LiDAR sensor and detecting obstacles in the surrounding environment in real time. This obstacle detection data will eventually be used by path planning software to improve autonomous driving capabilities of the golf cart system.

Contents

1. Abstract 2
2. Contents 3
3. System Overview 4
4. Our Part of the System 5
5. Challenges Faced 5
6. Provided Hardware and Resources 6
    a. LiDAR Sensor 6
    b. Ubuntu PC 7
    c. PCL (Point Cloud Library) 8
    d. ROS (Robot Operating System) 9
7. Specifications and Scope 10
8. Block Diagram 10
9. Receiving LiDAR Data 11
10. Sorting the Data 12
11. Defining Obstacles 13
12. Creating Obstacle Map 14
13. Testing 14
14. Operating Procedure 15
15. Error Discussion 16
16. Further Implementation 18
17. References 19
18. Appendix A: Source Code 20
19. Appendix B: Wiring Diagram 27

System Overview

Russell Morrow

The DAS (Distributed Autonomous Systems) Lab at Oklahoma State University is responsible for developing several self-piloting vehicles and co-robot learning tools, including several small aircraft, simulation equipment, and a self-driving golf cart. The golf cart will be the focus of this project. It is intended to be a fully autonomous vehicle utilizing input data from multiple sources, including GPS, accelerometer, and LiDAR sensor data. When the user inputs a destination, the golf cart should determine a clear path to the destination and drive there autonomously.

The golf cart project is ongoing, involving the work of several different teams in both the MAE and ECE departments over the course of several years. At the beginning of the Spring 2016 semester (when this team began work with the DAS golf cart), an onboard computer and the steering, braking, and power systems had all been implemented, but some other systems had yet to be fully developed. The purpose of this ECEN Capstone Design project is primarily to help implement the LiDAR sensor.

Our Part of the System

Katie Hobble

The main goals of this project were to establish communications with the LiDAR sensor and to use that data to detect obstacles in the environment. Additional path-planning and vehicle control software, which are beyond the scope of this Capstone Design project, will need to be developed before the golf cart is ready to drive autonomously. The primary focus of this project was to improve the vision systems of the golf cart using the LiDAR sensor provided.

Challenges Faced

- Getting up to speed on existing systems. Before we could define the goal of this project, we had to learn about the subsystems of the golf cart that had already been implemented. Determining how our project would fit together with future software projects dictated many of our design choices, like the choice of programming language.
- Multiple teams across different departments were simultaneously working on different components of one system. We were lucky enough to have several different people working on the golf cart this semester, so it took cooperation to coordinate simultaneous work on various sometimes related subsystems.
- We were restricted to the hardware provided by DAS. Thankfully, they provided us with all the materials necessary to achieve our goal. The LiDAR sensor hardware and the onboard computer provided to us for this project will be discussed in greater detail in the next section.
- We were limited to C++ and ROS to ensure compatibility with existing and future systems. Luckily, we had some experience with C++ programming across the team, but ROS was entirely new to all of us.
- Much of our work this semester was devoted to optimizing the output frequency (processing speed) of our algorithm. Our DAS counterparts in the MAE department requested an updated two-dimensional map output once per second (1 Hz). Our algorithm is currently outputting an updated map once every 200 milliseconds (5 Hz), which matches the rotational frequency of the LiDAR sensor itself.
- Determining an appropriate range of z values to include in the region of interest being scanned was an important consideration in reducing the amount of noise in the data due to dust and other small particles, reflections, and other noise in the incoming data stream.

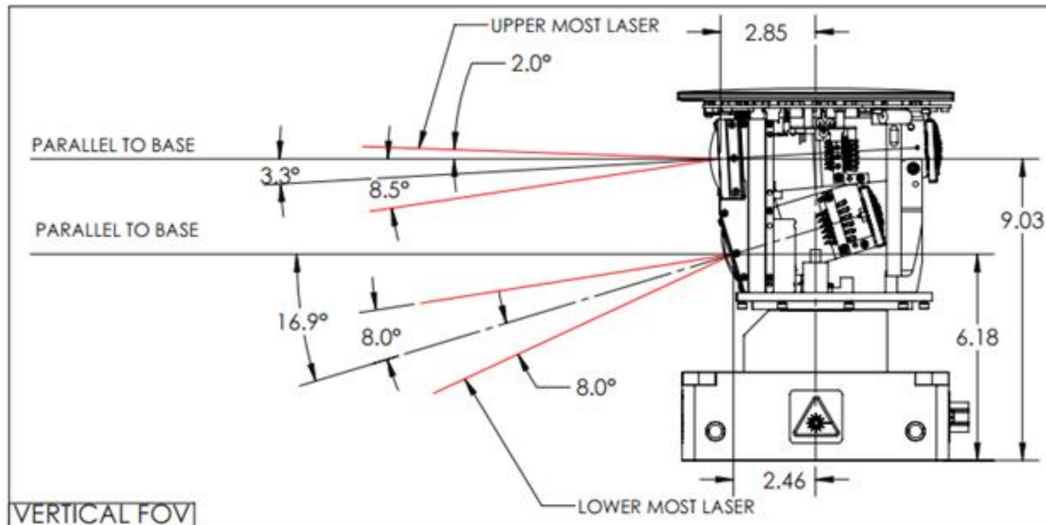
Provided Hardware and Resources

LiDAR Sensor

David Harp

The sensor being implemented for this project is a Velodyne HDL-64E LiDAR sensor. As seen in the images above [1], the front of the sensor shows the upper and lower banks of laser emitters and receivers. There are 64 sets of laser emitters and receivers altogether (32 in the upper bank and 32 in the lower bank). The laser receivers are split into two groups of 32 near the center of the sensor, and laser emitters are split into four groups of 16 to the sides of the receiver banks. Each laser fires with a 10 ns pulse width and a firing frequency of 34,375 Hz. The back panel shows that the sensor transmits data over a simple ethernet (CAT5) connection. It returns 2.2 million data points per second, outputting 100 Mbps UDP Ethernet packets [1].

The HDL-64E uses Class 1 lasers, which are deemed eye safe for humans. Each laser fires at a slightly different frequency, near the 905 nm wavelength, in the near-infrared spectrum. The lasers are fired with dynamic power selection to improve intensity data resolution at longer ranges, but overall power consumption is typically 60 Watts. It can operate on 12-32 VDC at operating temperatures between -10° and 65° C [1].



The sensor has a vertical field of view of 26.8° with an angular resolution of 0.08°. It sees from 2° above azimuth (horizon) to 26.8° below azimuth with 64 equally spaced angular subdivisions . At its current position mounted on top of the golf cart, that means that the LiDAR sensor can't see the ground within a 5 meter range. Additional short range LiDAR or ultrasonic sensors are recommended to improve vision within 5 meters. Despite its blind spots in very short range, the HDL-64E is rated at a 50 meter range for pavement and a 120 meter range for cars and foliage [1].

This is the same type of LiDAR sensor used in the Google self-driving car. In addition to the Google car, HDL-64E sensors are used in a variety of industries, including mining, digital mapping, and urban planning. However, automotive applications are the primary focus right now, with companies like Ford, Volvo, and TomTom all pursuing projects involving Velodyne LiDAR sensors [2].

Ubuntu PC

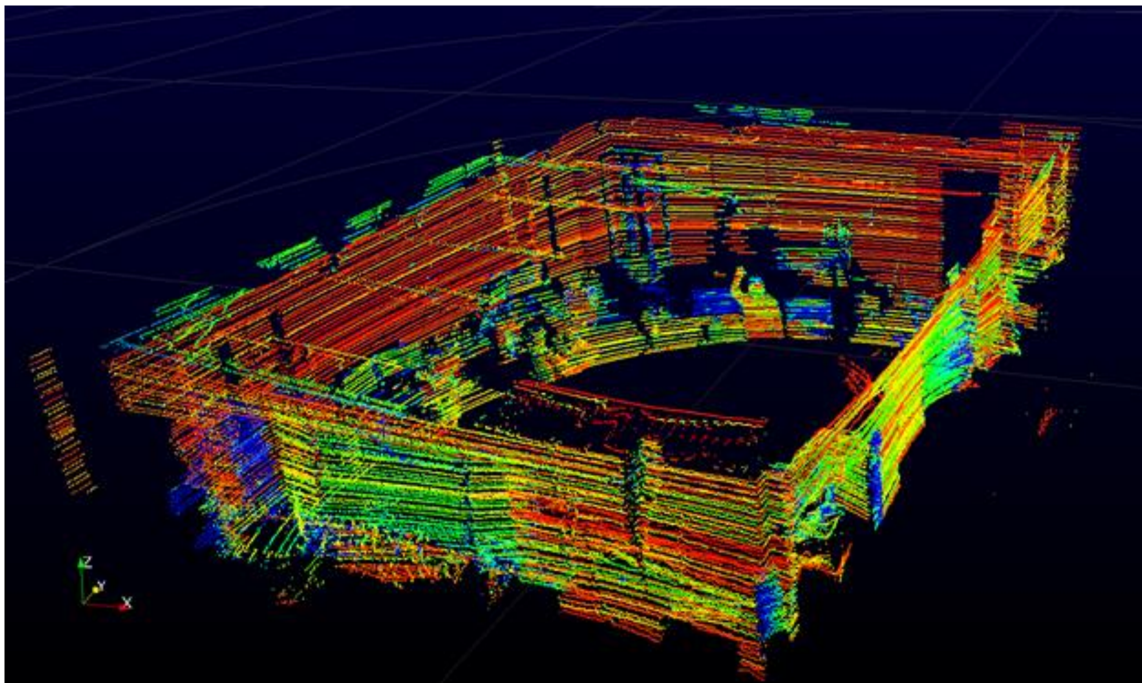
Katie Hobble

A computer running Ubuntu 14.04.3 LTS was provided by the DAS lab for this project. The provided computer is a custom built PC using an Intel i7-4790S CPU and 16GB of RAM. This platform offered exceptional data processing speed and memory flexibility. ROS requires a Linux operating system, so Ubuntu was chosen as a common operating system for all the software associated with the golf cart. Since C++ is compatible with ROS, C++ was selected as the common programming language for all golf cart systems. Python is also compatible with ROS, but since more Oklahoma State University students are familiar with C++, the DAS team decided to use C++ as the common language instead.

PCL (Point Cloud Library)

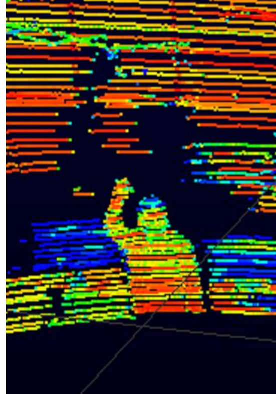
Dante Xiang

The Point Cloud Library is a standalone, large scale, open-source project for 2D/3D image and point cloud processing [3]. In our project, data received from the LiDAR sensor comes in as a “Point Cloud” data type. Each time a laser fires and its receiver gets a return, the sensor’s driver software uses the angle from azimuth (horizon), the angle of rotation, and the intensity of the laser to calculate the location of the datapoint as an x, y, z value. The LiDAR sensor’s current position is the system’s reference (origin) point. From the sensor’s current position as it is mounted to the golf cart, positive y is directly in front of the golf cart, positive x is 90 degrees to the right of the golf cart, and positive z is directly up.



This point cloud image of the DAS Lab here at OSU gives an idea of how the LiDAR sees. On the left side of this image, a sliver of the hallway outside the room is visible through a narrow window in the door. The interior of the supply closet can also be seen in the opposite corner of the lab. In the back corner stands a standard 19” server rack. There are also desks, chairs, computers, and some other visible clutter around the room.

A person can be seen waving near the center of the frame. Note his shadow on the wall behind him. The LiDAR uses light to see, and it can’t see through solid objects. The sensor simply doesn’t return any data for areas that are shadowed behind other objects in the environment.

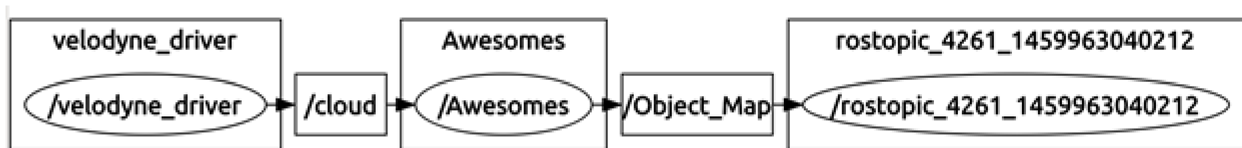


It also can't see anything closer than 5 meters unless the object is elevated high enough that it encounters a laser, which is why the person has no legs; he's standing just inside minimum range. These blind spots are an important consideration when designing path-planning software that relies on LiDAR data.

ROS (Robot Operating System)

Russell Morrow

ROS is an open-source robotics platform with a variety of built-in functionality for applications in robotics. Several existing systems on the golf cart use the ROS platform, so this project required ROS as well to ensure compatibility with other software. There are many tools in ROS that are used in the system itself. For this project ROS was used to collect raw data from the LiDAR sensor by subscribing to its data stream ROS is also used to publish the output of our system, making it available to other pieces of software [4].



This image shows the network topology of the ROS modules used for this project. The */velodyne_driver* node is the LiDAR sensor itself. It uses the /cloud publisher to allow other software access to the data stream containing the information detected by the LiDAR sensor. The /Awesomes node contains the algorithm developed for this project, and it uses the /Object_Map publisher to make its output data available to other programs.

Specifications and Scope

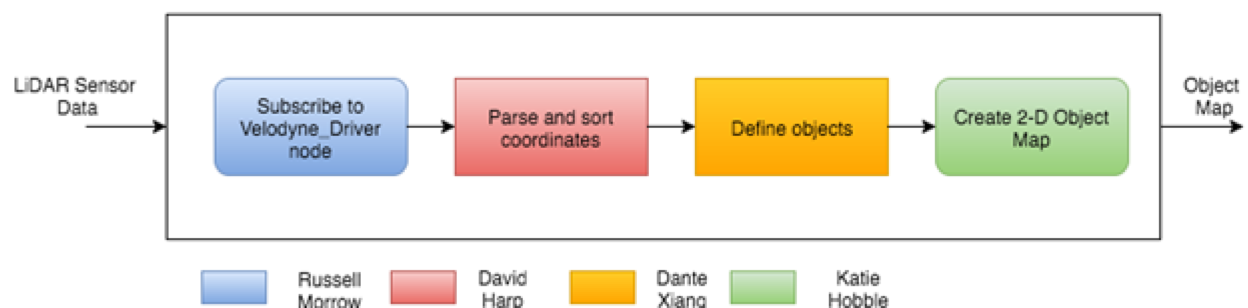
Note: These specifications were made assuming the LiDAR system is operating under ideal conditions (a flat surface with objects placed on ground) and providing valid data.

1. Data Acquisition
    1.1. Algorithm shall receive data from the LiDAR in an *xyz* coordinate system format.

2. Object Detection
    2.1. Algorithm shall detect objects between 5 and 25 meters from the golf cart.
    2.2. Algorithm shall flag an object that is at least 5 cm above the ground plane.
    2.3. Algorithm shall provide a 2-D map of *xy* coordinates, each with a corresponding Boolean value to indicate if an obstacle presides at that coordinate.

3. Data Transfer
    3.1. Algorithm shall provide output at least once per second.

Block Diagram



Receiving LiDAR Data

Russell Morrow

To receive the data from the LiDAR sensor, our software "subscribes" to the */velodyne_driver* node in our ROS network. The subscriber is a functionality built into ROS that allows a program or system to access the data stream of another device in the system. Likewise, the publisher is a part of ROS that is used to output a data stream so that other software can then use this output data in their system. Later, a publisher will be used to output the two-dimensional obstacle map created in Katie's block.

The */velodyne_driver* node contains a publisher that publishes the *xyz* coordinates of the data points detected by the LiDAR sensor. By subscribing to this node's publisher, we can see the raw data coming in from the sensor:


```
sensing@sensing-All-Series: ~
header:
  seq: 65966
  stamp:
    secs: 0
    nsecs: 0
  frame_id: map
height: 1
width: 384
fields:
  -
    name: x
    offset: 0
    datatype: 7
    count: 1
  -
    name: y
    offset: 4
    datatype: 7
    count: 1
  -
    name: z
    offset: 8
    datatype: 7
    count: 1
is_bigendian: False
point_step: 16
row_step: 6144
data: [128, 228, 149, 191, 243, 50, 73, 192, 253, 155, 146, 190, 0, 0, 128, 63,
30, 42, 161, 191, 27, 29, 88, 192, 215, 209, 165, 190, 0, 0, 128, 63, 210, 88, 1
51, 191, 130, 51, 73, 192, 15, 234, 146, 190, 0, 0, 128, 63, 172, 251, 162, 191,
 232, 115, 88, 192, 255, 148, 166, 190, 0, 0, 128, 63, 203, 228, 152, 191, 165,
81, 73, 192, 37, 95, 147, 190, 0, 0, 128, 63, 4, 67, 164, 191, 110, 20, 88, 192,
 247, 109, 166, 190, 0, 0, 128, 63, 147, 66, 154, 191, 171, 50, 73, 192, 45, 134
, 147, 190, 0, 0, 128, 63, 114, 231, 165, 191, 63, 45, 88, 192, 13, 227, 166, 19
0, 0, 0, 128, 63, 81, 46, 156, 191, 12, 200, 73, 192, 99, 151, 148, 190, 0, 0, 1
28, 63, 154, 140, 167, 191, 76, 69, 88, 192, 37, 88, 167, 190, 0, 0, 128, 63, 14
, 165, 157, 191, 206, 197, 73, 192, 117, 229, 148, 190, 0, 0, 128, 63, 156, 26,
169, 191, 119, 62, 88, 192, 53, 166, 167, 190, 0, 0, 128, 63, 72, 204, 129, 191,
 234, 216, 75, 192, 98, 254, 131, 190, 0, 0, 128, 63, 62, 6, 130, 191, 171, 53,
76, 192, 6, 110, 132, 190, 0, 0, 128, 63, 37, 37, 131, 191, 17, 195, 75, 192, 15
2, 35, 132, 190, 0, 0, 128, 63, 169, 56, 131, 191, 243, 225, 75, 192, 207, 72, 1
32, 190, 0, 0, 128, 63, 200, 145, 132, 191, 126, 203, 75, 192, 6, 110, 132, 190,
 0, 0, 128, 63, 200, 145, 132, 191, 126, 203, 75, 192, 6, 110, 132, 190, 0, 0, 1
28, 63, 37, 195, 133, 191, 208, 118, 75, 192, 207, 72, 132, 190, 0, 0, 128, 63,
119, 78, 134, 191, 149, 78, 76, 192, 79, 77, 133, 190, 0, 0, 128, 63, 143, 148,
135, 191, 7, 24, 76, 192, 79, 77, 133, 190, 0, 0, 128, 63, 225, 7, 135, 191, 122
, 64, 75, 192, 207, 72, 132, 190, 0, 0, 128, 63, 142, 96, 136, 191, 99, 40, 75,
```


The raw data arrives as a series of numbers corresponding to *xyz* coordinates. Each set of four numbers (enclosed in red rectangles above) represents a single *x, y*, or *z* coordinate detected

by a laser. The first set (128, 228, 149, 191) represents the *x* location of a data point. The second set (243, 50, 73, 192) represents the *y* coordinate of the same data point. The third set (253, 155, 146, 190) represents the *z* coordinate of the same point. After that, the data stream continues with the *x* coordinate of the next data point.

When taking in this data, we disregard any *xyz* value that is above the Golf Cart itself so when saving these points we take only the points that are outside of the cart and at the height of the Golf Cart or lower. The z coordinate range that is used for this algorithm is controlled by the *minbound* and *maxbound* variables. Adjusting those values will provide a greater or lesser region of interest for the obstacle detection algorithm, which may be useful for different applications of this golf cart. The current region of interest was chosen to reduce noise in the environment. Refer to the "Testing" section of this report for further details about the *minbound* and *maxbound* variables.

After we save at least 200 packets, a full rotation has been made, so the LiDAR can create a snapshot of the entire space around the golf cart. Once the algorithm has received a full 360° view of the surroundings and recorded each data point in the region of interest, that data is sent to David's block to parse through the data and sort each data point.

Sorting the Data

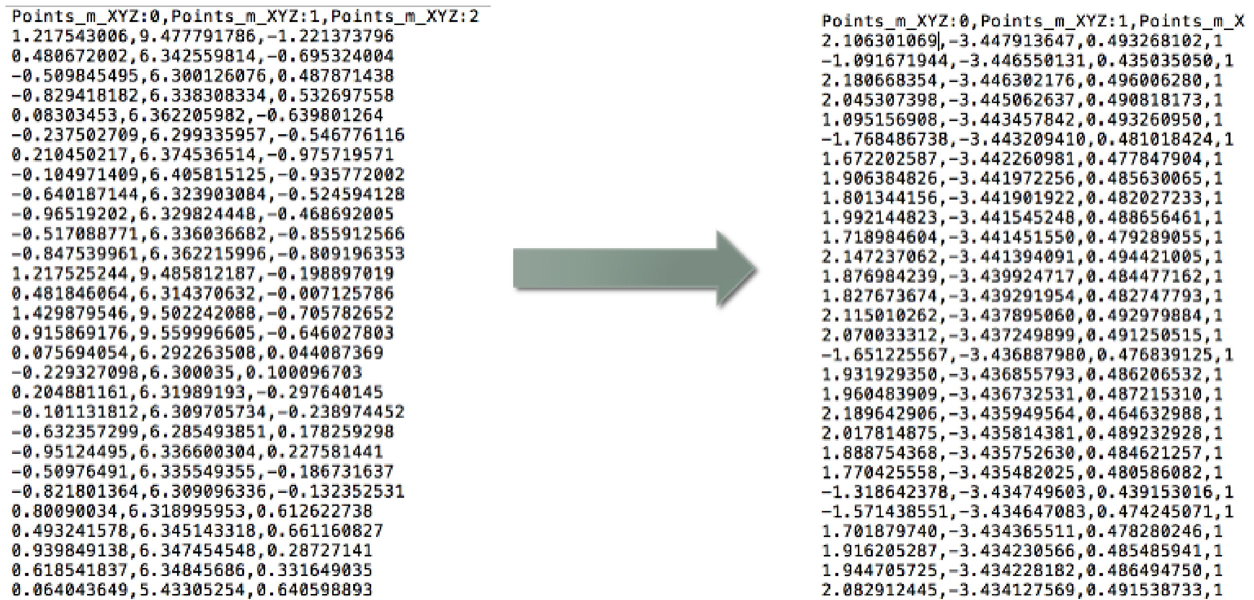
David Harp

In order to make comparisons more efficient in the next block, we first need to sort the raw data. By implementing our own comparison method, our algorithm sorts the incoming sensor data first by x coordinate, then by y coordinate. This comparison function will be used by Dante's obstacle definition block to find points with the same *x* and *y* values. Refer to the relevant code below; comments are in blue:

```
bool my_cmp(const point& a, const point& b) {
 //This checks to see if both coordinates have same X coordinate
        if(a.x-b.x<0.00001 && b.x-a.x<0.00001)
//If so, return y coordinate comparison
        return a.y<b.y;
        else
 //If not, return x coordinate comparison
        return a.x<b.x;
}//my_cmp
```

Using pointwise comparisons, data points that are near each other in the *x* direction (within 10 micrometers) are then compared by *y* value. Data points that are within 10 micrometers of each other in the *y* direction are grouped together and sorted accordingly. This

sorting provides us a more efficient way to compare the *z* values of data points that are near each other in the *x* and *y* directions, which will be useful for the next step.



```
Points_m_XYZ:0,Points_m_XYZ:1,Points_m_XYZ:2
1.217543006,9.477791786,-1.221373796
0.480672002,6.342559814,-0.695324004
-0.509845495,6.300126076,0.487871438
-0.829418182,6.338308334,0.532697558
0.08303453,6.362205982,-0.639801264
-0.237502709,6.299335957,-0.546776116
0.210450217,6.374536514,-0.975719571
-0.104971409,6.405815125,-0.935772002
-0.640187144,6.323903084,-0.524594128
-0.96519202,6.329824448,-0.468692005
-0.517088771,6.336036682,-0.855912566
-0.847539961,6.362215996,-0.809196353
1.217525244,9.485812187,-0.198897019
0.481846064,6.314370632,-0.007125786
1.429879546,9.502242088,-0.705782652
0.915869176,9.559996605,-0.646027803
0.075694054,6.292263508,0.044087369
-0.229327098,6.300035,0.100096703
0.204881161,6.31989193,-0.297640145
-0.101131812,6.309705734,-0.238974452
-0.632357299,6.285493851,0.178259298
-0.95124495,6.336600304,0.227581441
-0.50976491,6.335549355,-0.186731637
-0.821801364,6.309096336,-0.132352531
0.80090034,6.318995953,0.612622738
0.493241578,6.345143318,0.661160827
0.939849138,6.347454548,0.28727141
0.618541837,6.34845686,0.331649035
0.064043649,5.43305254,0.640598893
```

```
Points_m_XYZ:0,Points_m_XYZ:1,Points_m_X
2.106301069,-3.447913647,0.493268102,1
-1.091671944,-3.446550131,0.435035050,1
2.180668354,-3.446302176,0.496006280,1
2.045307398,-3.445062637,0.490818173,1
1.095156908,-3.443457842,0.493260950,1
-1.768486738,-3.443209410,0.481018424,1
1.672202587,-3.442260981,0.477847904,1
1.906384826,-3.441972256,0.485630065,1
1.801344156,-3.441901922,0.482027233,1
1.992144823,-3.441545248,0.488656461,1
1.718984604,-3.441451550,0.479289055,1
2.147237062,-3.441394091,0.494421005,1
1.876984239,-3.439924717,0.484477162,1
1.827673674,-3.439291954,0.482747793,1
2.115010262,-3.437895060,0.492979884,1
2.070033312,-3.437249899,0.491250515,1
-1.651225567,-3.436887980,0.476839125,1
1.931929350,-3.436855793,0.486206532,1
1.960483909,-3.436732531,0.487215310,1
2.189642906,-3.435949564,0.464632988,1
2.017814875,-3.435814381,0.489232928,1
1.888754368,-3.435752630,0.484621257,1
1.770425558,-3.435482025,0.480586082,1
-1.318642378,-3.434749603,0.439153016,1
-1.571438551,-3.434647083,0.474245071,1
1.701879740,-3.434365511,0.478280246,1
1.916205287,-3.434230566,0.485485941,1
1.944705725,-3.434228182,0.486494750,1
2.082912445,-3.434127569,0.491538733,1
```

LiDAR data before and after sorting

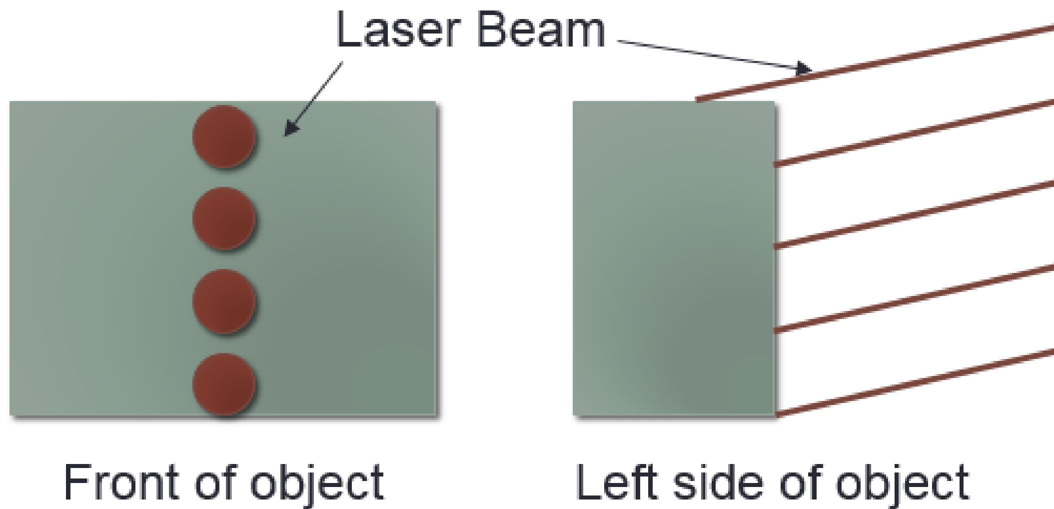
Defining Obstacles

Dante Xiang

In the obstacle definition block we count how many different *z* values are there at the same *xy* point. After calling the sort function from the previous block, the algorithm compares data points with the same *x* and *y* coordinates, and counts how many *z* values are present at each location. If there are more than one datapoint at any *x* and *y* location with multiple *z* values (greater than the threshold height of 5 cm), the system flags those data points as obstacles.

```
//Loop through vector to find objects
 for(int i=0;i<pointCloud.size();i++)
 {
        //Count the number of Z's at a particular XY region
        if (abs(p.x-pointCloud[i].x)<1 && abs(p.y-pointCloud[i].y)<1
                && abs(p.z-pointCloud[i].z)>0)numZs++;
        else numZs=0;
```

It makes sense in the real world that multiple laser returns from the same *x* and *y* location means the *xy* location we are looking at has a certain height that the car cannot drive over. The algorithm requires at least two different *z* values at the same *xy* point in order to filter out some of the noise caused by dust and other small particles in the environment.



An example of the obstacle definition algorithm counting z values

Creating Obstacle Map

Katie Hobble

Once coordinates of objects are determined, the x and y coordinates of each point flagged as an obstacle is stored into a String Stream which will then be outputted after the entire Point Cloud is searched through and sorted. This String Stream object is stored into the ROS publisher's message data object, which is then published to the golf cart's ROS network.

The system then outputs the current clock time, so we can monitor how frequently the system is outputting data. It then clears several values that will be used as counters in the next cycle, like the number of packets received and the number of z values at the current point. Finally, it clears the point cloud vector, which helps keep system memory available for the next cycle.

An actual Point Cloud object was not chosen due to our output not having any z coordinates, but we have included the Point Cloud object in our output stream as a pass-through of the original input data, which is helpful for testing. However, our algorithm's actual output publisher will use the String Stream object type. The String Stream object consists of signed floating point values, starting with the x coordinate, a comma, then the y coordinate, and finally a new line terminator. This structure readable by other ROS subscriber nodes, such as the visualization program RVIZ.

This allows us to visualize both the raw sensor data and also the output data of our algorithm. The system provides a new output map every 200 milliseconds, meaning its output frequency is 5 Hz, the same frequency that the LiDAR sensor itself rotates at. This means that

the algorithm outputs a full map for each rotation of the sensor. There is not noticeable latency between raw data and algorithm output, but for high-speed applications, further attention to the latency introduced in the system may be required.

Operating Procedure

Russell Morrow

To operate our program, simply run the file “DanteStart” in a terminal, or double-click the icon on the desktop. This runs a script that starts the obstacle detection algorithm. Once the software is running, other programs on the system will have access to the output data stream by subscribing to the /Object_Map publisher.

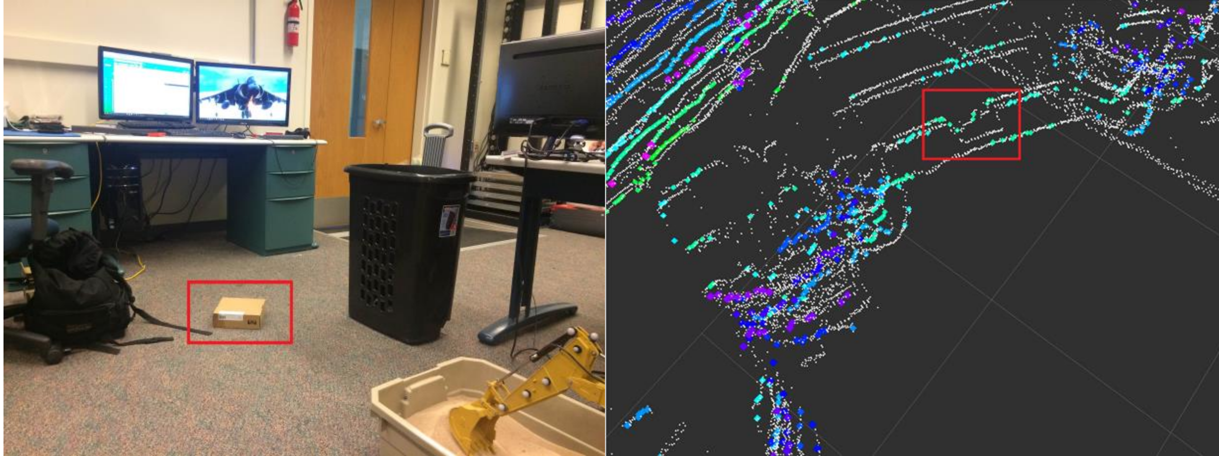
Certain variables will be useful to know about for maintenance or for upgrading the system. The double variables *minbound* and *maxbound* set the minimum and maximum allowable z values in the region of interest to be scanned for obstacles. Increasing the *maxbound* parameter might be useful for testing LiDAR vision with unmanned aerial vehicles or other airborne obstacles.

DataOut.csv has been included in our code so that output will be available in a comma-separated values (.csv) format if needed. In its current state, our algorithm works with or without it, but it has been included in order to make integration with future path-planning software less costly in terms of development time.

Testing

Dante Xiang

To test the frequency of the system’s output, we included a clock_t object that monitors the elapsed time between output cycles. The timer and a few other variables are reset after each time the system publishes its output, so we can monitor the output time of each cycle. The system is currently outputting data every 200 ms, well below the required frequency of 1 Hz. Increasing the region of interest by increasing the value of *maxbound* will likely increase processing time, which may affect output frequency and the effectiveness of path-planning software, so adjusting *maxbound* should be done in small increments, with thorough testing to ensure proper system timing. The current region of interest was chosen specifically to reduce noise in the output data, so be aware that changes to *minbound* may introduce additional error as well.



To test whether the system could reliably detect an object at the minimum specified elevation, a 5 cm tall box was rendered in RVIZ, the point cloud visualization tool built into ROS. As seen in this screenshot, the sensor clearly sees the top edge and the bottom edge of the box. The white data points show the sensor's raw data, while the colored data points represent the output of our obstacle detection algorithm. In the raw data, both the top edge and the bottom edge of the box are visible. However, in the obstacle detection output, only the top edge is being displayed. Similar tests were performed throughout the range of the system (5-25 meters) to confirm functionality.

Error Discussion

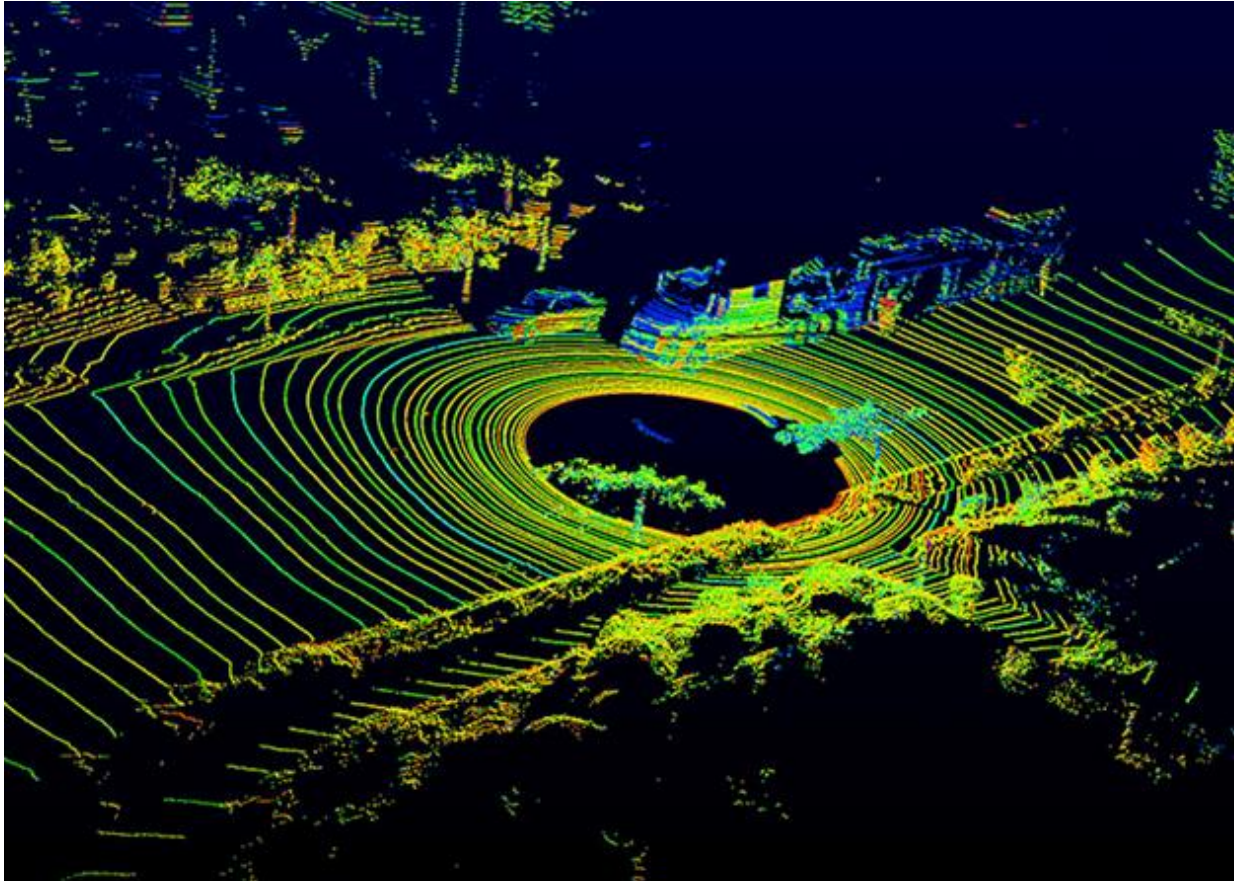
David Harp

Because the Velodyne HDL-64E uses lasers firing at a fixed angle relative to the golf cart, data points farther away from the golf cart will tend to be fairly far apart, while data points at short and medium range (5-25 meters) tend to be fairly close together. That is to say, image resolution improves at close range and degrades at distance. That is why a maximum range of 25 meters was selected for this project. It includes enough data to make useful decisions about the immediate environment at typical golf cart speeds.

Top speed of the golf cart is 15 mph (6.7 meters per second), so it takes 0.746 seconds for the golf cart to travel the minimum LiDAR range of 5 meters. Since our algorithm provides output every 200 milliseconds, the system has 546 milliseconds to respond to an obstacle detected at the minimum range of 5 meters. As long as path-planning software can adjust course within that half-second window, the golf cart should be safe to use in an unlikely worst-case scenario.

Typical golf cart operation will not be at top speed, nor should we expect to typically see obstacles that suddenly appear at close range. However, due to the large blind spot immediately

around the golf cart, safety concerns are still problematic. Implementing additional, smaller LiDAR sensors like the Velodyne HDL-32E or their smaller Puck model would considerably improve the safety of the golf cart.



This image [2] demonstrates the decrease in resolution as range increases.

Our testing has shown that within the specified operating range of 5-25 meters, the sensor can safely detect an obstacle 5 cm above the golf cart's ground plane. Beyond that range, usable data can still be collected from the */velodyne_driver* node, but after path-planning software has been integrated, additional testing will be required to determine a safe maximum operating range for LiDAR sensor data.

The sensor is rated to 120 meters, so although it was not specifically tested for in this project, much of the data coming in from the sensor is in the 25-120 meter range. The HDL-64E is accurate to less than 2 cm according to the manufacturer's datasheet [1], so for detection of large obstacles like cars, trees, and buildings, data in the 25-120 meter range is most likely safe to use for path planning. Until further testing is implemented to determine the safety of detecting small obstacles at long range, we recommend using data only from within the specified range of 5-25 meters.

Future Improvements

Katie Hobble

Additional functionality can be implemented to improve overall golf cart vision. First, since the HDL-64E is mounted on top of the golf cart, there is a roughly 5 meter blind spot immediately around the cart. An additional LiDAR sensor dedicated to short range vision would substantially improve safety. Ultrasonic sensors could also be implemented for short range vision if the cost of a smaller LiDAR sensor is prohibitively high.

Three-dimensional pattern recognition software can be implemented with existing PCL libraries, or new pattern recognition software can be developed, possibly as a project for graduate students in the ECEN department. Some object recognition is already available as open-source software, including several correspondence grouping algorithms, which rely on model templates to identify geometry in the environment. The branching patterns of trees have been modelled mathematically in the *octree* class and others [5].

Other algorithms use eigenvector and eigenvalue calculations to determine certain properties of a cluster of data points, matching them to basic geometric shapes [5]. Other open-source options may be available, and this is an area of active research, so new options should become available over time to improve robotic vision for light vehicle applications like the golf cart.

Improved pattern recognition is likely the best improvement to existing systems that can be implemented for a robust path-planning software to function effectively in recognizing and responding to the most common obstacles around campus: cars, pedestrians, cyclists, and trees. If a novel pattern recognition implementation is proposed rather than implementing existing pattern recognition software, it is recommended to pursue an interdisciplinary project with the Mathematics and Computer Science departments.

Another improvement for golf cart vision is more precise sensor calibration. Several different calibration tables are available from the manufacturer's manual [6] along with some open-source calibration tables that may or may not be better suited to the golf cart. Finding or developing a more precise calibration table optimally suited to this specific application could improve resolution and accuracy of LiDAR data noticeably. Additional academic research in the field of high-precision LiDAR calibration techniques can be found in the work of C. Glennie [7], [8], who suggests several techniques for increasing LiDAR accuracy for both static and high speed applications, and G. Atanacio-Jimenez, et al [9], who suggest a different approach using pattern planes like the floor and walls to calibrate the sensor.

References

[1] *Velodyne HDL_64E Datasheet*, 1st ed. Velodyne, 2016.[Online]. Available: http://velodynelidar.com/docs/datasheet/63-9194_Rev-D_HDL-64E_Data%20Sheet_Web.pdf

[2]"HDL-64E", *Velodynelidar.com*, 2016. [Online]. Available: http://velodynelidar.com/hdl-64e.html. [Accessed: 29- Apr- 2016].

[3]"Documentation - Point Cloud Library (PCL)", *Pointclouds.org*, 2016. [Online]. Available: http://pointclouds.org/documentation. [Accessed: 29- Apr- 2016].

[4]"Documentation - ROS Wiki", *Wiki.ros.org*, 2016. [Online]. Available: http://wiki.ros.org/. [Accessed: 29- Apr- 2016].

[5]"Point Cloud Library (PCL): Module recognition", *Docs.pointclouds.org*, 2016. [Online]. Available: http://docs.pointclouds.org/trunk/group__recognition.html. [Accessed: 29- Apr- 2016].

[6]*HDL-64E S2 Manual*, 1st ed. Velodyne, 2016.

[7]C. Glennie and D. Lichti, "Static Calibration and Analysis of the Velodyne HDL-64E S2 for High Accuracy Mobile Scanning", *Remote Sensing*, vol. 2, no. 6, pp. 1610-1624, 2010.

[8]C. Glennie, "Calibration and Kinematic Analysis of the Velodyne HDL-64E S2 Lidar Sensor", *Photogrammetric Engineering & Remote Sensing*, vol. 78, no. 4, pp. 339-347, 2012.

[9]G. Atanacio-Jimenez, J. Gonzalez-Barbosa, J. B., F. J., H. Jimenez-Hernandez, T. Garcia-Ramirez and R. Gonzalez-Barbos, "LIDAR Velodyne HDL-64E Calibration Using Pattern Planes", *Int J Adv Robotic Sy*, p. 1, 2011.

Appendix A: Source Code

```
/****************************************************************
Oklahoma State University - Electrical and Computer Engineering
This code is used on the Automated Golf Cart in the DAS Lab

Authors: David Harp, Katie Hobble, Russell Morrow, Dante Xiang
Spring 2016

This code subscribes to the Velodyne LiDAR publisher node, takes
in the PointCloud, sorts it, and checks to see if there are any
objects detected in the PointCloud. It then publishes the x and
y signed floating point values
****************************************************************/

#include "ros/ros.h"
#include "std_msgs/String.h"
#include <stdlib.h>
#include <sstream>
#include <iostream>
#include <fstream>
#include <stdio.h>
#include <pcl/point_cloud.h>
#include <pcl_ros/point_cloud.h>
#include <pcl/point_types.h>
#include <pcl_conversions/pcl_conversions.h>
#include <sensor_msgs/PointCloud.h>
#include <string.h>
#include <cstring>
#include <time.h>
#include <string>
#include <vector>

typedef pcl::PointCloud<pcl::PointXYZ> PointCloud5;
using namespace std;

//Define Global Variables
//Used for proving output time - Create object that keeps track of the current time
clock_t t;
//Used to initially filter noise from incoming PointCloud from sensor. Serves as bounds for
```

```
//Region of interest along the Z axis
double minbound=-9,maxbound=0.1;
//Used as the incoming PointCloud object
pcl::PointCloud<pcl::PointXYZ> cloud2;
//Used as the outputting PointCloud for RVIZ support
pcl::PointCloud<pcl::PointXYZ> Output;
//Used to keep track of number of packets coming in
int num_pacs = 0;
//Our own struct that defines what a point is. Has XYZ floating points
struct point
{
   float x,y,z;
}p;
//Vector used to keep track of objects detected
vector<point> pointCloud;

/****************************************
chatterCallBack
inputs: PointCloud2 msg

THIS IS OUR SUBSCRIBER NODE

This function is called when PointCloud data is received from the driver of the Velodyne sensor.
It saves that data in a global variable (double) called "p" which we will use later to parse
the data. This code is provided partially by the Velodyne Driver node
****************************************/
void chatterCallback(const sensor_msgs::PointCloud2 msg) {
	//This is the datatype that is coming in to our subscriber
	pcl::PCLPointCloud2 pcl_pc;
	//These two PCL functions convert the pcl_pc input into usable PointCloud
	pcl_conversions::toPCL(msg,pcl_pc);
	pcl::fromPCLPointCloud2(pcl_pc, cloud2);

	//Loops through all points in the incoming PointCloud and initially filters any noise in the
Z coordinates
	for(int inc=0; inc<cloud2.points.size();inc++) {
		//Checks to see if current Z coordinate is in appropriate bounds
		if(cloud2[inc].z>minbound && cloud2[inc].z<maxbound) {
			//This comparison removes the noise that surrounds the golf cart (ring of
points around cart)
			 if(abs(cloud2[inc].x)>1.5||abs(cloud2[inc].y)>1.5) {
```

```
                              //Save current coordinate to our own PointCloud object to not
slow down subscriber
                              p.x=cloud2[inc].x;
                              p.y=cloud2[inc].y;
                              p.z=cloud2[inc].z;
                              //This pushes the coordinate back into the vector
                              pointCloud.push_back(p);
                       }//end if
               }//end if
  }//end for

        //The following line adds 1 to the value of a variable called num_pacs.
        //num_pacs counts how many packets have been received and processed from the sensor.
  num_pacs++;
}//end chatterCallBack

//The next section of code is the comparison function for sorting the data received in the last
section.

/*****************************************
my_cmp
inputs: point a, point b
outputs: TRUE or FALSE

Used as custom sort comparison for sorting
XYZ coordinates. Sorts by X coordinate then Y coordinate
*****************************************/
bool my_cmp(const point& a, const point& b) {
        //This checks to see if both coordinates have same X coordinate
  if(a.x-b.x<0.00001 && b.x-a.x<0.00001)
         //If so, return y coordinate comparison
         return a.y<b.y;
  else
         //If not, return x coordinate comparison
         return a.x<b.x;
}//my_cmp


/**********************************
main
```

inputs: int argc, char argv

The main loop of our program.
Sets up publisher and subscriber nodes, sorts incoming PointCloud,
Declares obstacles, and publishes object coordinates

```
***********************************/
int main(int argc, char **argv) {

	//This sets the clock variable to the current time in order to keep track of output speed
	clock_t t3 = clock();

	/* ===SET UP ALL ROS NODES=== */

	//Initializes our node. This is a ROS protocol
	ros::init(argc, argv, "Awesomes");
	//Alert the user the node has been initiated
  cout <<  " Awesomes is setup ;)"<<endl;
	//This grabs a node handle of our node. This is a ROS protocol
	ros::NodeHandle n;
	//This sets up our subscriber node. This is a ROS protocol
	ros::Subscriber sub = n.subscribe("cloud", 1, chatterCallback);
	//This sets up the string XY map we will be outputting. This is a ROS protocol
	ros::Publisher chatter_pub = n.advertise<std_msgs::String>("Object_Map", 1);
	//This sets up the PointCloud we also output. This is a ROS protocol
	ros::Publisher  output_pub = n.advertise<PointCloud5> ("RVIZMap",1);

	/* == SORT AND FILTER POINT CLOUD DATA == */

	//Used to keep track of number of Z's at a particular XY region
  int numZs=0;
	//Once again set the current time
  t3=clock();

	/* == ROS PUBLISHER NODE == */

	//Set up the ROS Publisher output string and StringStream. This is ROS Protocol
  std_msgs::StringPtr str(new std_msgs::String);
  std_msgs::String msg;
	stringstream ss;
  string output="";
```

```
	//The following section of nested loops compares x,y, and z values for similar
	//points, and filters out the points that don't satisfy the criteria for
	//classification as "obstacles"
	//It also avoids redundant data by eliminating some duplicate values.

	//Loop while ROS node is active
    while(ros::ok())
    {
		//Initialize a new PointCloud. We output this as well as a regular StringStream
		PointCloud5::Ptr msg2 (new PointCloud5);
		//Initialize the PointCloud and output values
		msg2->header.frame_id = "map";
		msg2->height = 1;
		msg2->width = 0;
		output="";
		//ROS Protocol to get packets
		ros::spinOnce();
		//If number of packets is above 200, we have a full map
		if(num_pacs>=200)
		{
			//Sort the point cloud vector using David's function
			sort(pointCloud.begin(), pointCloud.end(),my_cmp);
			//Loop through vector to find objects
			for(int i=0;i<pointCloud.size();i++)
			{
				//Count the number of Z's at a particular XY region
				if (abs(p.x-pointCloud[i].x)<1 && abs(p.y-pointCloud[i].y)<1
					&& abs(p.z-pointCloud[i].z)>0)numZs++;
				else numZs=0;
				//end if

				//Check to ensure final point parsed is a part of the region of
interest
				if(p.x!=pointCloud[i].x || p.y!=pointCloud[i].y ||
p.z!=pointCloud[i].z)
				{
					//Region has been parsed, look at number of Z coordinates
found
					if(numZs>0)
					{
```

```
                                                    //Region has more than one Z coordinate, it's an
obstacle!

                                                    //FOR OUTPUT PURPOSES

                                                    //Add the X and Y coordinates to the string stream
                                                    ss<<pointCloud[i].x<<","<<pointCloud[i].y<<endl;
                                                    //Add the XYZ coordinates to the PointCloud we
output (RVIZ)
                                                    msg2-
>points.push_back(pcl::PointXYZ(p.x,p.y,p.z));
                                                    msg2-
>points.push_back(pcl::PointXYZ(pointCloud[i].x,pointCloud[i].y,pointCloud[i].z));
                                                    //Increase the message width
                                                    msg2->width+=2;
                                                    //Reset the number of Z's to 0 for the next region
                                                    numZs=0;
                                            }//end if
                                    }//end if

                                    //Set the next points to look at for the point vector
                                    p.x=pointCloud[i].x;
                                    p.y=pointCloud[i].y;
                                    p.z=pointCloud[i].z;
                            }//end for

                            /* === OUTPUT THE DATA === */

                            //Set the ROS Publisher message data to be the string stream
                            msg.data = ss.str();

                            //Publish the string stream data
                            chatter_pub.publish(msg);
                            //Publish the object point cloud
                            output_pub.publish(msg2);

                            //Reset the output values to be used by the next round of publishing data
                            ss.str(string());
                            ss.clear();
                            //Output the current clock to keep track of how fast algorithm outputs
                            cout<<clock()-t3<<" "<<endl<<CLOCKS_PER_SEC<<endl<<endl;
```

```
                        t3=clock();
                        //Reset number of packets and Z's received for next round of publishing
data
                        num_pacs=0;
                        numZs=0;
                        //Clear the PointCloud vector(or else it fills up and becomes too slow)
                        pointCloud.clear();
                }//end if
   }//end while

        //End of Main
        return 0;
}//end main
```

Appendix B: Wiring Diagram

S3 Hybrid Connector (J3)

J3	WIRE COLOR	FUNCTION
1	RED (16AWG)	SUPPLY VOLTAGE
2	BLACK (16AWG)	GROUND
3	GREEN	PPS PULSE IN
4	ORANGE	FUTURE OPTION
5	BLUE	GPS GND
6	WHITE	GPS IN
7	BLACK	SERIAL IN
8	RED	SERIAL GND

GPS 18x LVC & GPS 18x -5Hz PINOUT

GPS 18x Pin #	Color	Signal Name	Wire Gauge
1	Yellow	Measurement Pulse Output	28
2	Red	Vin	26
3	Black	Ground	28
4	White	Transmit Data	28
5	Black	Ground	26
6	Green	Receive Data	28

Table 1: GPS 18x LVC & GPS 18x-5Hz Wire Pinout