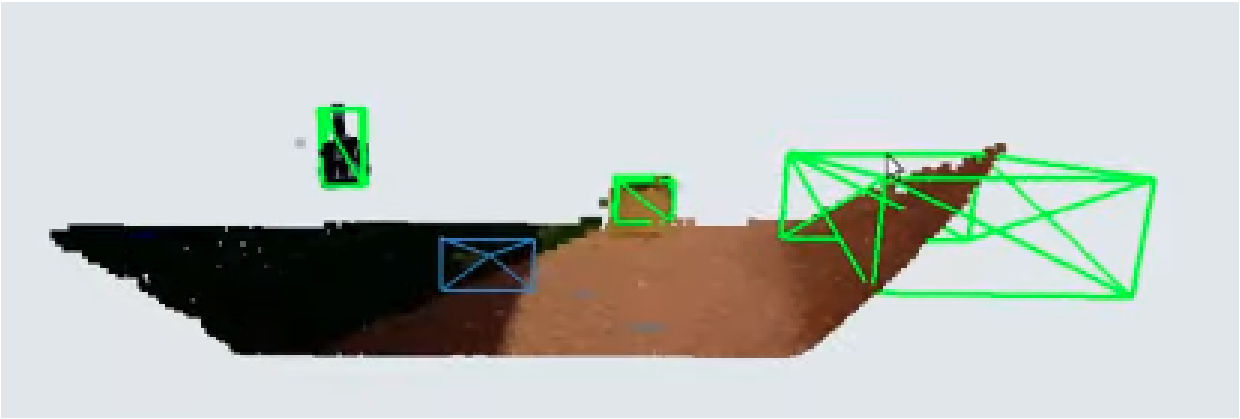
CUDA/GPU Accelerated Point Cloud Obstacle Avoidance Proposal

Motivation

While the current obstacle avoidance system has met the original goals of providing highly detailed information about the rover surroundings and enhancing our ability to autonomously navigate the environment, it faces significant performance bottlenecks due to the limitations of single-threaded, iterative processing techniques used in the CPU. Using the GPU resources of the Jetson TX2 allows us to massively parallelize point cloud processing, and in doing so increase processing speed as well as free up CPU computing resources for other rover applications. The speed gains may also enable us to increase the resolution of clouds we are processing for more precise environment analysis.

Demo 1 (Static Pre-Recorded):
https://www.youtube.com/watch?v=RkddJ96bdQU&feature=youtu.be
Demo 2 (Real-time with Motion):
https://drive.google.com/file/d/1e2u6YJS53IFTPDkuaxjDnSaVE7faO2qX/view?usp=sharing
Demo 3 (Real-time no Motion):
https://www.google.com/url?q=https://drive.google.com/file/d/1Guiwv6iRA8LR44pzmpbwBqzd76JJZjj2/view?usp%3Dsharing&sa=D&ust=1611938305328000&usg=AOvVaw0xRgWQkKXI51TiGJ8wN9Jy

The Demos



Demo 1 shows that the new CUDA point cloud processing pipeline is very comparable to the PCL system in accuracy and function. It can identify discrete objects in the scene and even

compute a 3 dimensional bounding box for these objects. Demo 2 shows how when run in real time, this system offers the same functions of obstacle detection and path planning at a considerable performance improvement. Some issues with the CUDA pipeline are observed in demo 1. The program occasionally splits a single large obstacle into multiple detections. Better parameter tuning is likely to reduce this effect as it did for the PCL pipeline. Additionally, we see that there are some sparse false detections near the ground. This can be improved by refining the ground-plane segmentation procedure and parameters. In summary, the new system seems to provide similar functionality with a large performance gain, subject to further development-time to optimize its robustness.

Description

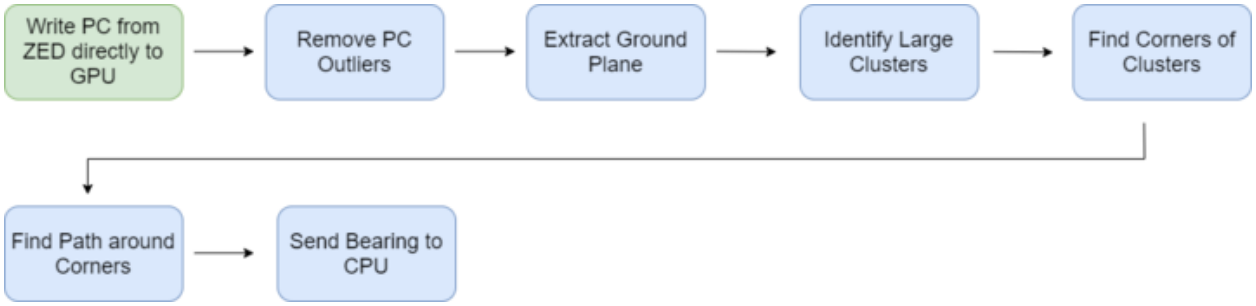
This new CUDA/GPU system has parallelized, GPU compatible implementations of the algorithms used by PCL in our current CPU-based obstacle detection system, and as such was written from scratch. However, on a high-level, the procedure is the same as the current system. The ZED stereo camera is used to capture a point cloud of the current scene into GPU memory. A pass through filter runs to eliminate irrelevant points that are above the rover height or very far away. RANSAC plane segmentation is then used to identify the ground plane. Points that fall within a threshold of the ground plane will be removed to simplify the scene. From here, Euclidean cluster extraction allows us to identify the discrete obstacles in the scene (which are no longer connected via the ground points). Once points are properly classified with their respective obstacles, the minimum and maximum coordinates of each obstacle are identified as “interest points” that define a bounding box for that obstacle. Knowledge of the location of these bounding boxes is used to determine a clear path through the environment. Information about the identified clear path is sent over LCM.

Point clouds processing often inherently lends itself to parallel computing since it is desirable to run the same processes on every point in the cloud. GPUs have specialized hardware to enable massively parallel computation. The combination of these two factors makes CUDA a logical platform for our obstacle detection system. Iterating sequentially over a large number of points in the CPU is slow, whereas running filters and clustering algorithms in parallel on the GPU can be dramatically faster.

For example, consider the case of the RANSAC plane segmentation. Originally, this algorithm had to loop over every single point in the point cloud (e.g. 14000) to determine if that point fell within the randomly chosen ground plane, and it had to repeat this for many randomly chosen ground planes (e.g 400). This required over 5 million iterations to run this single step of the pipeline. On the other hand, the GPU can check the points for each randomly chosen plane in parallel, reducing the number of iterative steps by two orders of magnitude. In practice, the GPU Ransac implementation is nearly 2.5 times faster than its CPU equivalent. Euclidean Cluster Extraction and pass through filtering have similar sequential logic that can be parallelized.

An added benefit of this system is that it provides tremendous flexibility and control since the pipeline is custom written for our rover. This allows us to remove some of the overhead associated with the general purpose PCL library functions as well as process and return data in a format of our choosing. Finally, switching to this pipeline could increase the speed of all software running on the TX2 in general, including subsystems other than perception. This is because all processes are currently sharing CPU resources, forcing the operating system to cycle resources and processor time between them. Our PCL obstacle detection is a computationally intensive program, meaning some significant CPU cycle time is currently spent on it. The GPU resources are completely unused as of now, and by utilizing them, it may free up more CPU cycles for other software.

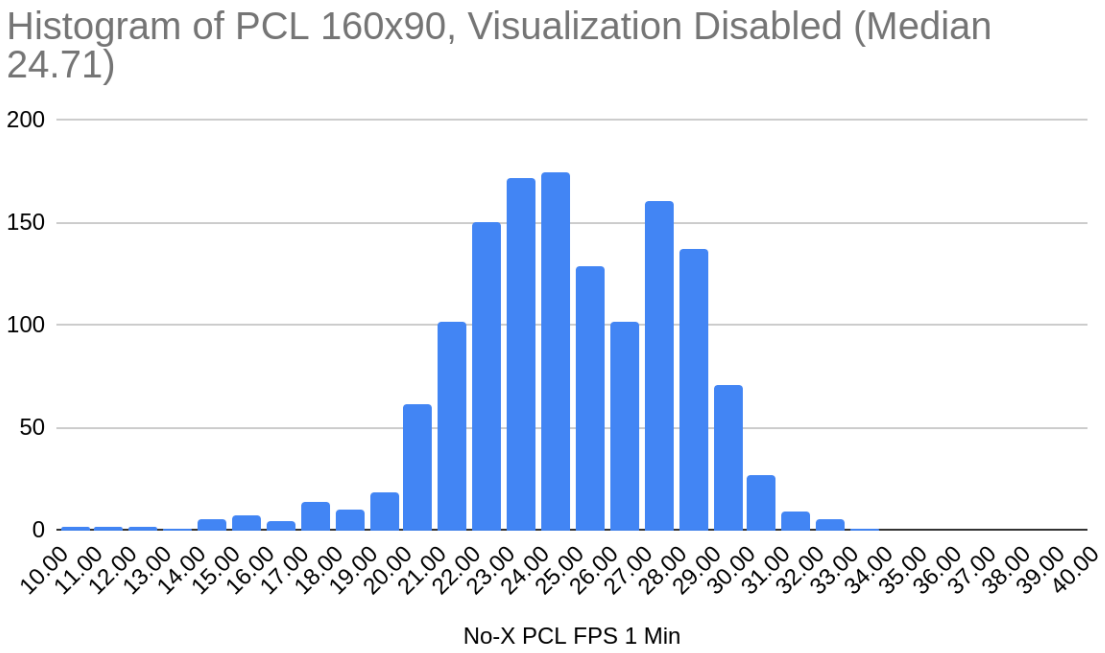
Perception's communications with other packages will not change and the hardware and location of the hardware that perception uses will not change. These changes are only in perception software.

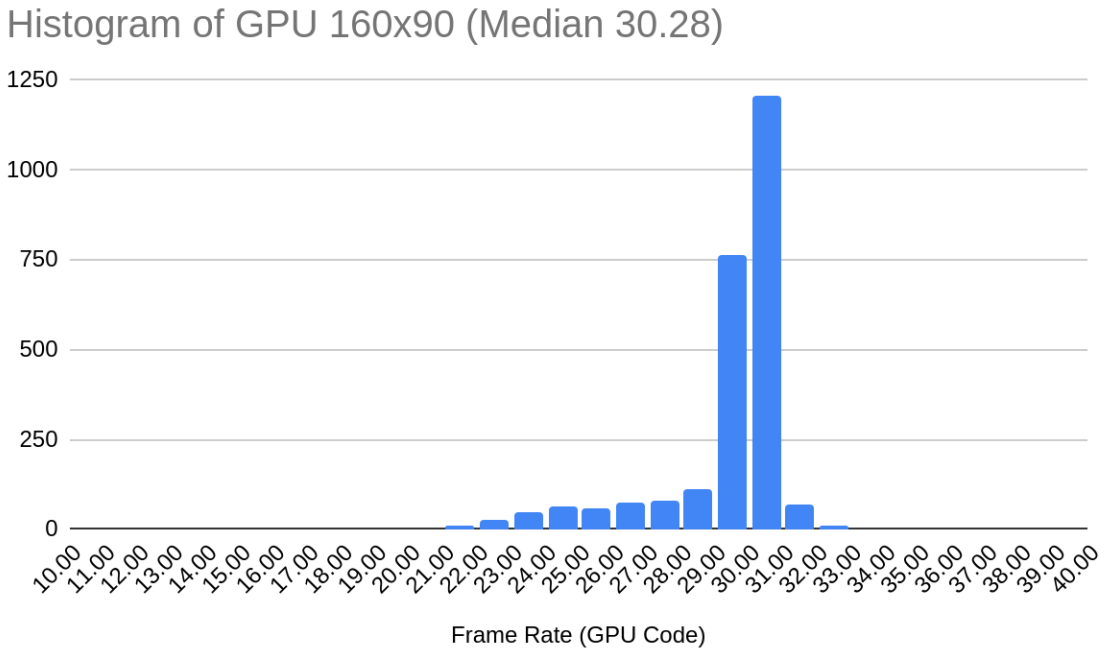


Data Analysis:

A true "one-to-one" comparison between the CUDA and PCL pipelines remains difficult for two major reasons. First, the PCL pipeline makes use of a Voxel Filter which downsamples the point cloud from beyond the input resolution by converting to a voxel representation of data that is utilized by PCL's clustering algorithm. This step is currently not present in our implementation, which means for the same input resolution, our GPU implementation must process considerably more points. Second, the two pipelines have entirely different visualization and drawing systems. To get a comparison of "competition mode" performance, the visualization should be disbled on both. However, at this time, we have not written the functionality to disable visualization into the CUDA system as it is still early in development.

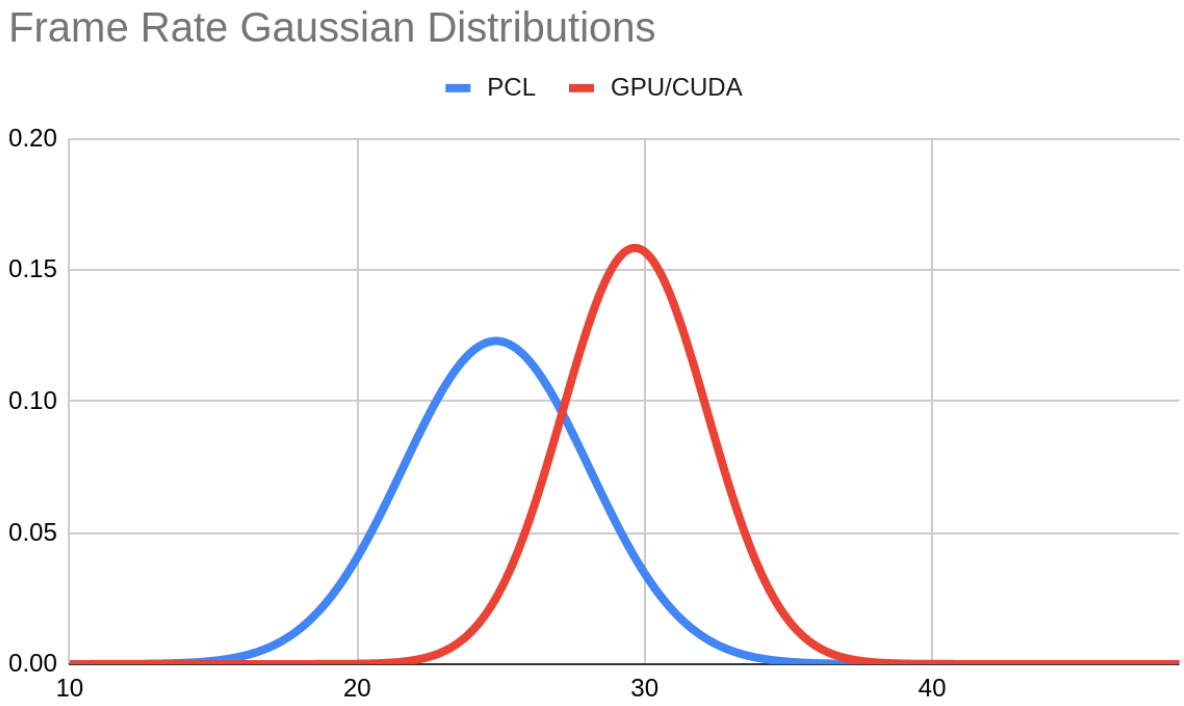
In this analysis, we will look at data under conditions that heavily favor PCL. PCL is set to use the same input resolution as the CUDA pipeline, the Voxel Filter remains in place, all visualization/debug output is disabled for PCL and remains enabled on the CUDA pipeline. The clouds PCL processes are reduced to about ~2800 points after filtering, and the clouds CUDA processes are still about ~6000 points after filtering .





These two plots demonstrate the frame rates of each pipeline with an input resolution of 160x90 detecting obstacles of similar size in a similar static environment. Note, the vertical axis is autoscaled to normalize the number of samples collected, but the horizontal axis has been set to [20, 40] with a fixed bin size of 1 so that the histograms align. In PCL's optimal configuration, we achieve an average performance of approximately 25 frames per second on the competition mode with no visualization. This means that the CUDA pipeline's average performance at about 30 frames per second with visualization enabled and no VoxelFilter is still about 22% faster than PCL. The CUDA software is processing about double the amount of points that PCL is in this test.

Another interesting observation is that the PCL pipeline's performance was less consistent, and experienced frame rate drops below 20 frames per second. Assuming frame rate to be a Gaussian random variable, we can consider the same data above by analyzing their bell curves on the same axis:

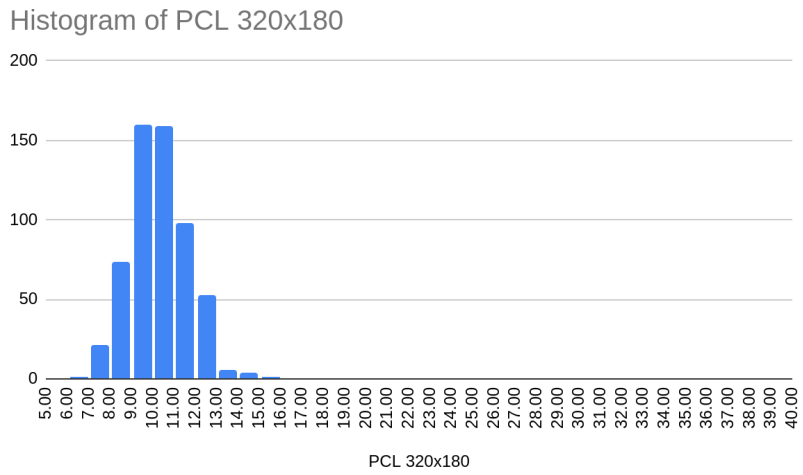


The standard deviations are 3.2 and 2.5 for the PCL and CUDA pipelines respectively. The minimums were 10.79 and 20.12 respectively. Overall, the worst-case performance of the CUDA pipeline is about twice as fast as the worst case performance of the PCL pipeline.

Future Analysis

The following remarks illustrate some scenarios that may be considered in the future.

- The PCL code has historically been observed running between 5-10 frames per second when the Voxel Filter has been disabled. In this case, the number of points processed was equivalent to the GPU implementation (which averaged about 30 frames per second). However, at this time, no controlled experiment has been performed to allow for a rigorous analysis of speeds without the Voxel Filter. We do have datasets that demonstrate that the Voxel Filter results in a reduction of a factor of about 2x the input cloud dimension. By reducing the resolution inputted to the GPU code to 113x63, we can simulate the effect of a Voxel Filter. Alternatively, a Voxel Filter could be implemented.
- In all tests (PCL and GPU), only perception obstacle-detection code was run for benchmarking. If additional rover software is run in the background, the performance gap between PCL and GPU implementations is expected to grow because the PCL implementation must share resources with other rover processes, while the CUDA implementation has the complete bandwidth of the GPU and should not be affected by background CPU processes.
- Writing a “silent” GPU implementation with no visualization will likely increase the performance further
- If it is desirable to have more detailed environment data, the speed gains from the GPU implementation may enable processing on larger input clouds in a timely fashion. Testing at 640x420 may be of interest. Due to the nature of parallelization, it is hard to determine on paper exactly how much runtime will scale with size, but it is likely to be better than the sequential algorithms if GPU occupancy is low



At this resolution, PCL processes about 7000 points post-voxel. 160x90 is 14400.

Dependencies

- Zed SDK
- CUDA 10.0
- OpenGL (3+)

Requirements

Requirement ID	Requirement Description	Notes
AT.1.C.i	Perception system shall detect objects that are greater than 45 cm in height from a distance of 6 m	This system will permit us to recognize the size or distinct objects and allow us to fulfill this requirement
AT.1.C.iii	Perception system shall detect objects that are greater than 24 cm in height from a distance of 2 m	This system will permit us to recognize the size or distinct objects and allow us to fulfill this requirement
AT.1.C.vi	Perception system shall detect ledges exceeding 15 cm vertical drop from a distance of 6 m	This will be challenging, but point clouds should provide the information necessary to make drop detection possible

Risk Analysis

Risk	Likelihood	Severity	Mitigation Plan?
The implementations are unstable or fail on edge cases	2	5	Put in much more development-time and testing to catch these cases and make code competition ready

New additions to the pipeline require us to write that functionality from scratch, rather than ready-to-go PCL modules	5	3	Training more perception programmers in CUDA will allow us to develop new features faster
Other perception programmers who are not familiar with CUDA will be less able to debug/develop obstacle code	3	1	CUDA Code is very similar to C++ with a few exceptions. Perception programmers can be trained in basic CUDA.

Alternatives

PCL CPU (Current Pipeline): The current PCL pipeline is undoubtedly highly effective for obstacle detection despite its speed limitations. At a maximum after all optimization, frame rates can peak no larger than 25 frames per second for PCL CPU with no other processes running on the TX2.

PCL GPU/CUDA: PCL does have some GPU algorithms which we explored. However, it currently has extremely limited support for GPU/CUDA optimized implementations of its algorithms. As of this time, many of these do not produce correct results and even run slower than the CPU implementations (GPU Euclidean Clusters). Some are incomplete (pass through filter), and do not ship with any release builds of PCL.

Drawbacks

The drawbacks of this system are given in the risk analysis. To recap, our algorithms are likely to be less robust than the older, more-developed PCL counterparts and will require some work. Since the entire pipeline is custom written, any new additions will require us to implement them from scratch. Finally, we will have to train perception team members in basic CUDA so they can debug and develop the obstacle detection code.

Development Plan

Steps:

- Implementation
    - Perform various filterings to decrease point cloud size
    - Use RANSAC Plane Segmentation (see glossary) to detect and remove ground
    - Use Euclidean clustering (see glossary) via building a graph describing points and their fixed-radius neighbors, then propagating cluster labels via connected component labeling
    - Iterate over all identified clusters, eliminate those that are too small, and determine extrema on each axis for a bounding box
    - Create a function to see if the provided path of the rover is clear
    - Create a function that uses trigonometry to find a new clear path
        - In the case that the current path is blocked
    - Add timing functions to all parts of the function
    - Create 3D visualizer to depict results of point cloud function
- Testing
    - Outdoor environmental testing will be performed in a mock obstacle course
    - Different processes performed within the function will be adjusted and tested
    - All adjustments will be tested in the same mock obstacle course to assess performance and accuracy and find a balance between the two
- Jarvis
    - The current proof of concept compiles with CMake. In order to integrate it with the existing rover code, the CUDA software must be built with Jarvis, which has not yet been attempted.

Timeline:

- Implementation - complete by 2021-04-30
- Testing - complete by 2021-08-30

Code Reviews:

- Project Review 1:
    - Objective: Assess whether the GPU implementation has potential to fulfill the relevant requirements and do so at a meaningful performance increase compared to its CPU counterpart
    - Tentative Date: 2020-01-30

- Project Review 2:
    - Objective: Review testing and success of project, full pull request review
    - Tentative Date: End of summer

Developers:

- Ashwin Gupta, Matt Martin

Resources

No additional hardware is required.

Glossary

A small table explaining terms that might not be clear

Term	**Definition**
C++	Another one of the software team's languages. Compiled language that generally runs faster than python. Used in the majority of CS classes at Michigan.
CUDA	CUDA is a computing platform that allows programmers to run code on GPUs and utilize their multi-threaded capability.
GPU (graphics processing unit)	GPU's are a special type of processor that have specialized hardware to make parallel computation very efficient. They were originally invented for computer graphics (computing each pixel simultaneously), but have since been used for more general purpose use through CUDA.
CPU (central processing unit)	CPU's are the traditional "processors" that are commonly found in computers, mobile devices, and tablets. These processors are very general purpose and are good at running instructions quickly in order (serial), but are not as good at running many things at the same time (parallel).

Point Cloud	A collection of points that contain x,y,z values, and when these points are mapped in a coordinate system, provide a 3D representation of the captured environment Example: https://tinyurl.com/yyqrn66y
Stereo Camera	A camera which is able to assign an x,y,z value to every pixel in each frame it captures.
Point Cloud Library (PCL)	The point cloud library is a large scale open project library that makes creating and processing point clouds extremely easy. The library provides a large breadth of functionality including containers, filters, visualizers, clustering algorithms, plane finding algorithms, shape recognition algorithms, sorting algorithms, and more, designed specifically for operating on point clouds. For more information on the capabilities of the point cloud library go to https://pointclouds.org/.
Euclidean Cluster Extraction	This algorithm is used to identify “clusters” of points. A point is considered to be within a cluster if it is at most some pre-defined radius away from the cluster (the search to check this condition is called fixed radius nearest neighbors). It functions by creating a graph of points and their fixed radius nearest neighbors, and then propagating labels throughout the graph using connected component labeling (CCL).
Ransac Plane Segmentation	This algorithm picks three random points in the point cloud and counts how many points lie on or, within a tolerance determined by the programmer, near to the plane created by these three points. These points are considered inliers. It then repeats this process of picking random points and determining inliers for some pre-defined number of iterations. The iteration with the most inliers and that meets the angle specifications is considered the optimal ground plane.
Voxel Filter	This is a filter that is only present in the PCL Pipeline. This filter significantly reduces the dimensions of input clouds to allow PCL to process less points.
LCM (Lightweight Communications and Marshalling)	The LCM is the resulting message generated after a program runs. This message is communicated to various other programs running on the rover. In this case the LCM is a number and is used by the auton program to direct the rover down the clearest path.