Introduction

I'd like to start my report by saying that I have no prior background or experience in "textbook" Data Science, so I was quite shocked when I first received the task, as I was expecting something more focused on programming itself. My relationship with math is also more related with statistics, and, while I'm not a complete stranger to analyzing data, I had never worked before with software that specializes on the matter. So I had to do a lot of research, methodology included. The theme of lane detection is also new to me, so it's safe to say, the whole process of completing this task is filled with uncertainty, but I'm proud of my end result during this time. I just wanted to get this out of the way because there can be some obvious preconceptions that I may be missing. Still, I was quite interested and comfortable with the topics and technology I came across, and I felt good working with them.

I completed the project with just C++, as it is the language I'm used to working with. It also made me realize how python is the main language when working with data science, as it was very easy to find python resources, while C++ felt like an afterthought. I also used the open3d library that was suggested for visualization and operations. I had no experience with any, so I stuck with it.

Methodology

My first thought was how to store and read the data correctly from the binary files, so I created a Point class that would represent each point from the cloud point. These points have 5 attributes: x, y, z coordinates, intensity and lidar beam. From those, I focused on the x and y coordinates and intensity. I was afraid to make any assumptions or interpretations with the z coordinate, as I don't have a very clear idea of what the process of collecting data looks like. (Does z = 0 represent a ground point?). The exercise header was also focused on x and y. The lidar beam information also didn't seem of much use.
After that I created a Pointcloud class, which is the main class of the program, that wraps around open3d's pointcloud and that includes our vector of Points. It also contains all the data manipulation operations necessary for the completion of the task. Let's start with filters.

Filters

The pointcloud contains a lot of points that are not relevant for the task, so it is necessary to remove them and to capture only the points that represent the car lanes. I created a total of 7 filters. I will show a demonstration of the original pointcloud with its added filters to better visualize their effectiveness until we reach the final result. Not all of them will have significant visual differences. Please note that I am showing them in progression with prior mentioned filters already applied. I ordered the filters by convenience on completing the task, but if optimization is truly an important factor, we should apply first the filters that are less computationally expensive and that remove the most points. For each cloudpoint I describe the parameters I used for filtering in a file with the same name as the cloudpoint binary one.

Intensity filter
As the exercise describes, this value represents the brightness of a certain point. Car lanes are painted in white and reflect the light well. The filter simply eliminates each point under a certain intensity value. To get a good idea about the distribution of intensity values between points, I created a function that store their values separated by them, and another that returns the average intensity.

Rotation filter
Sometimes the point cloud would not be perfectly aligned, and it would be necessary to adjust its rotation for better visualization and data management. Quick reminder that this rotation needs to later be undone so that the points would have their original coordinates for the lane fitting. Rotation applied on the z axis.

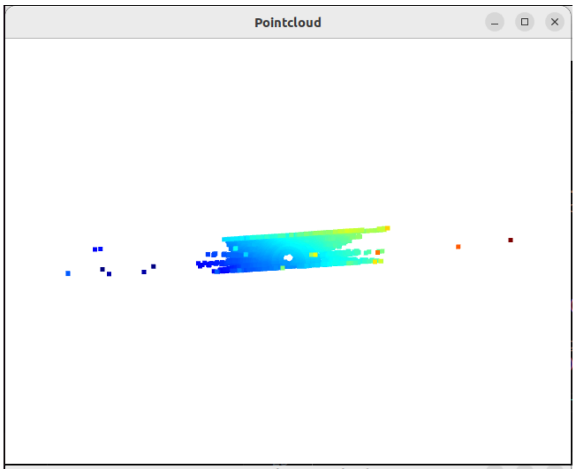
Distance filter
This filter delimits the points to a zone that is between -x and x values and -y and y values. It just designates the space we're working with. This is where I was most unsure about data interpretation. The values seem to be in meters, so I tried to use values that would correspond to the size of a real road. Sometimes I could figure out several lines around the car (center (0,0)), but it would not be completely clear to me which ones truly represent the road lanes.

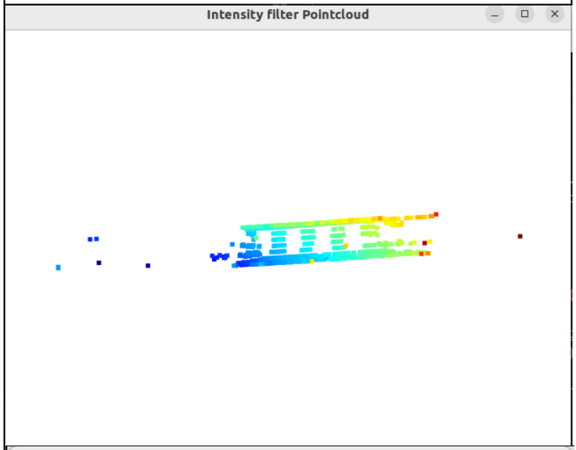
Slice filter
The order of this filter matters a ton, since it only makes sense to apply it after applying both the rotation and distance filters. This filters "slices" points horizontally above and under certain y values (same as distance filter), but also in between two inner y values. It basically announces a "thickness" that the lanes are allowed to have and removes all points outside this thickness.

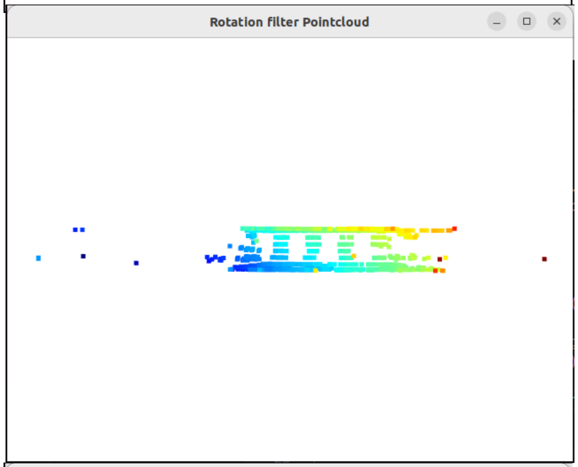
Clean filter
This filter takes advantage of the already incorporated functions in open3d, RemoveDuplicatedPoints and RemoveNonFinitePoints. Curiously enough, RemoveNonFinitePoints would always almost half the number of points existing, but I never grasped the true reason why.

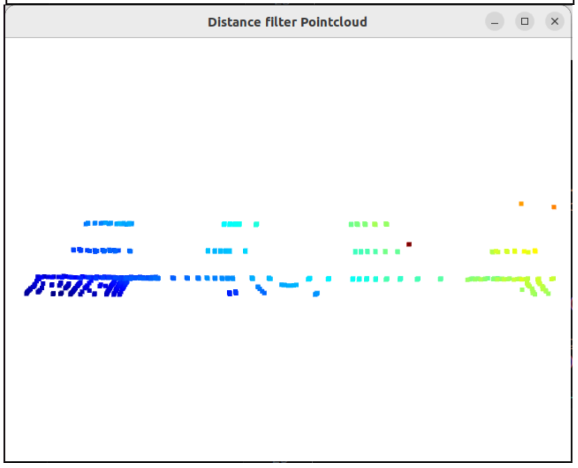
ClusterDBSCAN filter
This filter applies the DBSCAN clustering algorithm that is also already present in open3d. This algorithm detects a cluster of points, if a minimal amount of points are in a distance of each other. Sadly, I couldn't take good use of it, even adjusting its parameters. I think the main reason is that the amount of points at this stage already fluctuated around 100-500. It never proofed truly useful to me, but I decided to keep it anyway, as I didn't consider it to harm the results.

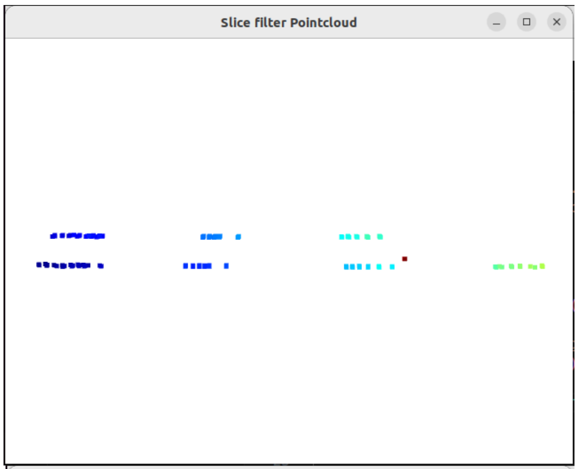
VoxelDownsample filter
This filter applies a voxel grid system to the pointcloud and aggregates all points inside a voxel. Much like the prior filter, it was already present in the open3d library, and I didn't find it useful in most of the cases.

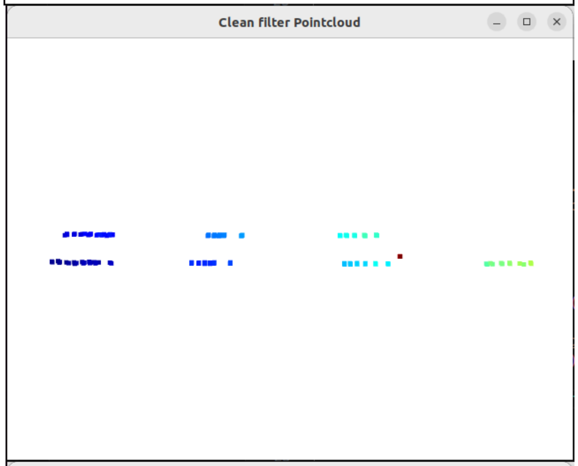

Pointcloud

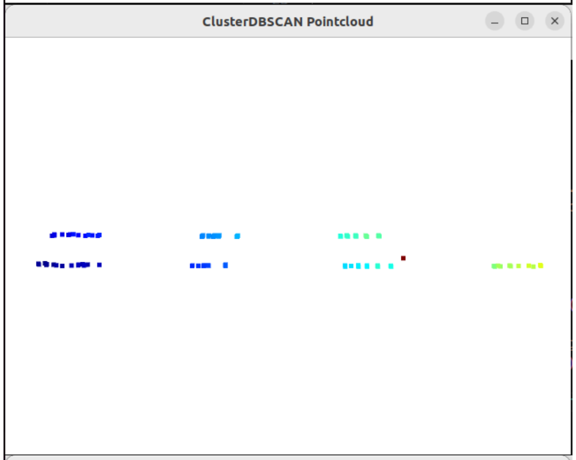

Intensity filter Pointcloud

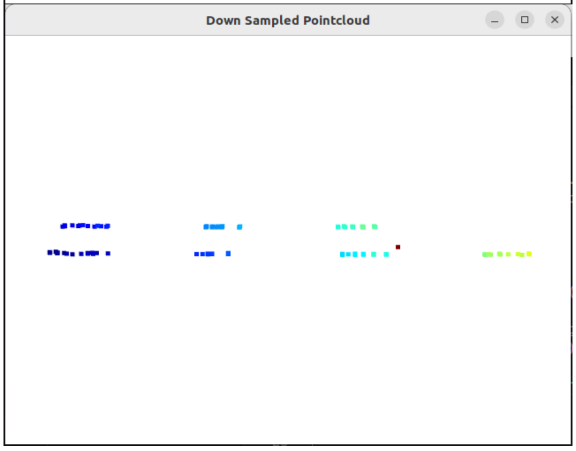

Rotation filter Pointcloud

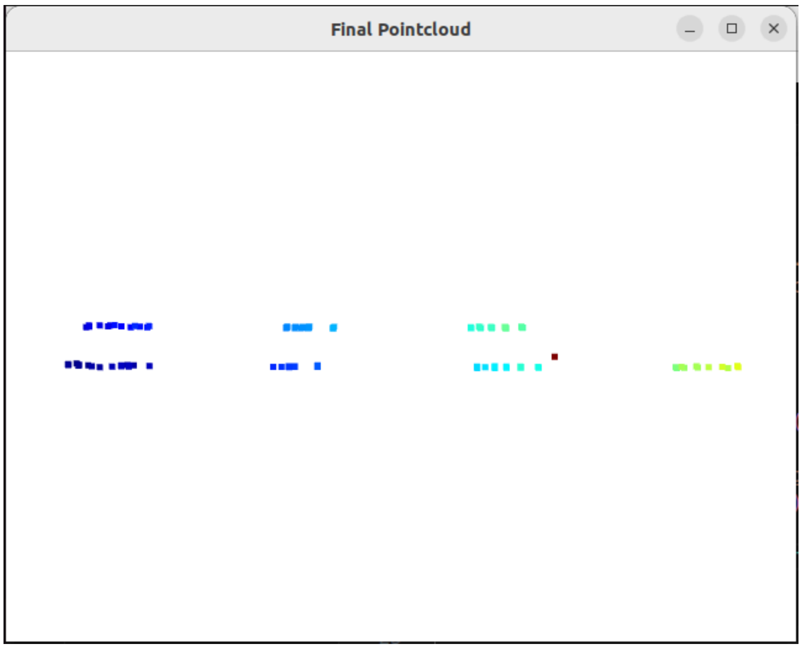

Distance filter Pointcloud


Slice filter Pointcloud


Clean filter Pointcloud

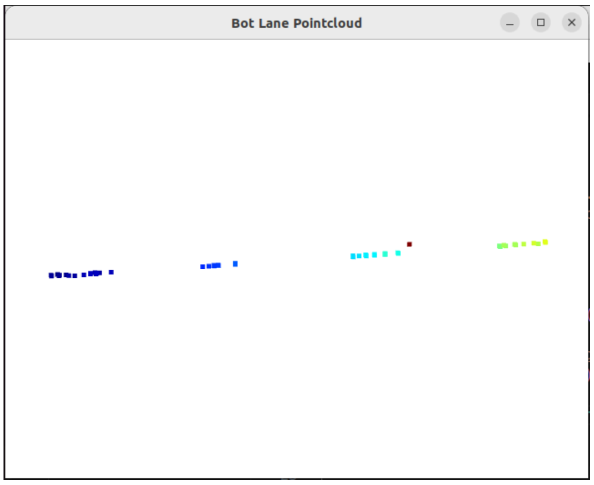

ClusterDBSCAN Pointcloud

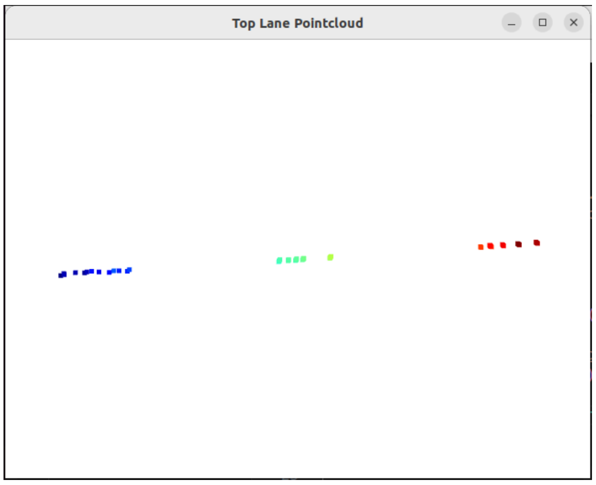

Down Sampled Pointcloud



Lane separation

After applying all filters, I know need to separate the points regarding one lane from another. I tried to accomplish this using a sorting algorithm. Given a proximity value on the y coordinate, it will detect all points within that proximity as belonging to the same lane. Beyond that, I need to add a point to separate between lanes. I try to place it as the starting point for the top lane. Example point (-100, 2, -100). The x coordinate just needs to be a value with x below the first point from the top lane. The y coordinate, a value close to the top lane within the proximity value used for the sorting algorithm. The z coordinate, a unique value that for sure no other point will have. Before separating the lanes, we need to undo the rotation applied in the filters. Since the rotation applied was on the z axis, the points' z coordinates won't change, unlike their x and y coordinates. With this, we can identify the separator point by its unique z value, and create two different vectors. One for the bot lane that ends on the separator point, and another for the top lane that starts on the separator point.





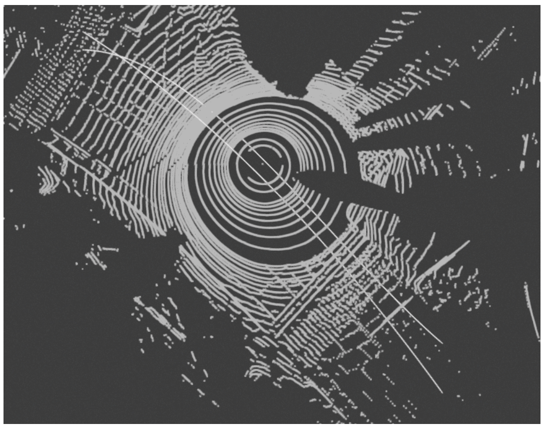
There are problems with this approach, however. If the lanes curve to a degree where the y coordinate from points from different lanes intersect, the sorting will not act as desired, and that could be a potential future problem. After confirming that this approach worked with all case scenarios provided, I kept it, but I understand the need to adapt it if the exercise were to extend outside its original scope.

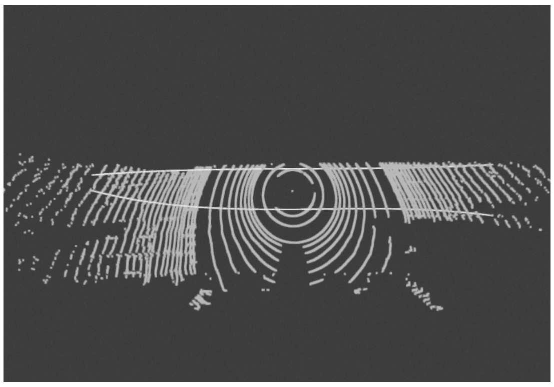
Polynomial fitting

While researching the topic, I couldn't find many existing C++ libraries that already added the functionality of a polynomial line regression, but I did find an article by Rahul Bhadani with its implementation using the Least Squares method. After understanding what I was coming up against, I didn't feel confident enough to create an implementation of my own, and decided that it was best to use already existing material, than to reinvent the wheel. The method involves trying to find a line that minimizes the squared distance between each point and the line. Using an equation for the sum of squared residuals, its minimal value is present when the slope is 0 which can be obtained by taking the derivative. We apply this implementation to our different lanes and obtain a third degree polynomial fitting.

Conclusion

The algorithm seemed to provide successful results in most cases, but there were a couple of results where the lines start to deviate unrealistically towards their end. I believe this mostly a result of my interpretation of the lanes in the point cloud, and not filtering correctly, which were the aspects I felt the least confident in. Beyond those scenarios, I feel satisfied with the other results.





References

Least-square Polynomial Fitting using C++ Eigen Package by Rahul Bhadani

https://towardsdatascience.com/least-square-polynomial-fitting-using-c-eigen-package-c0673728bd01

https://math.stackexchange.com/questions/7125/polynomial-fitting-how-to-fit-and-what-is-polynomial-fitting

https://www.mathworks.com/help/lidar/ug/lane-detection-in-3d-lidar-point-cloud.html

https://www.faro.com/en/Resource-Library/Article/Point-Clouds-for-Beginners

https://interestingengineering.com/science/what-is-lidar-technology-and-what-are-its-main-applications