NTNU
Kunnskap for en bedre verden

Fakultet for ingeniørvitenskap

TBA4251 - Programmering i geomatikk

Report

Author:
10006

Date: 10.01.2023

Table of Contents

List of Figures

1 Introduction

The objective of this project is to detect speed bumps in a point cloud of a road segment in Trondheim. Detected speed bumps can then be stored and visualized on a 2D map of the road. In practice mapping speed bumps can be useful for autonomous vehicles, city planning, and maintenance, among others. In this report I will first present the language and tools used, before showing an overview of the workflow and then the results. Afterwards I will go into detail about my code and algorithms and finally have a discussion.

2 Tools and languages

2.1 Python

I chose Python as the programming language for this project. It is the language I'm most comfortable with, and it supports libraries for handling LAS data. I also supports countless other libraries such as numpy, sklearn and scipy which I used for certain calculations.

2.2 Pylas

I used the Python library pylas for reading, processing, and writing las data. I found this to be user friendly, with easy methods to access and manipulate LAS data. However built in functions for processing data are quite limited, so the algorithms for processing point clouds had to be written from scratch. This is not a problem since the point of this project is to program myself.

2.3 CloudCompare

CloudCompare is a software for processing 3D data. I used this for visualization of the point clouds.

3 Workflow

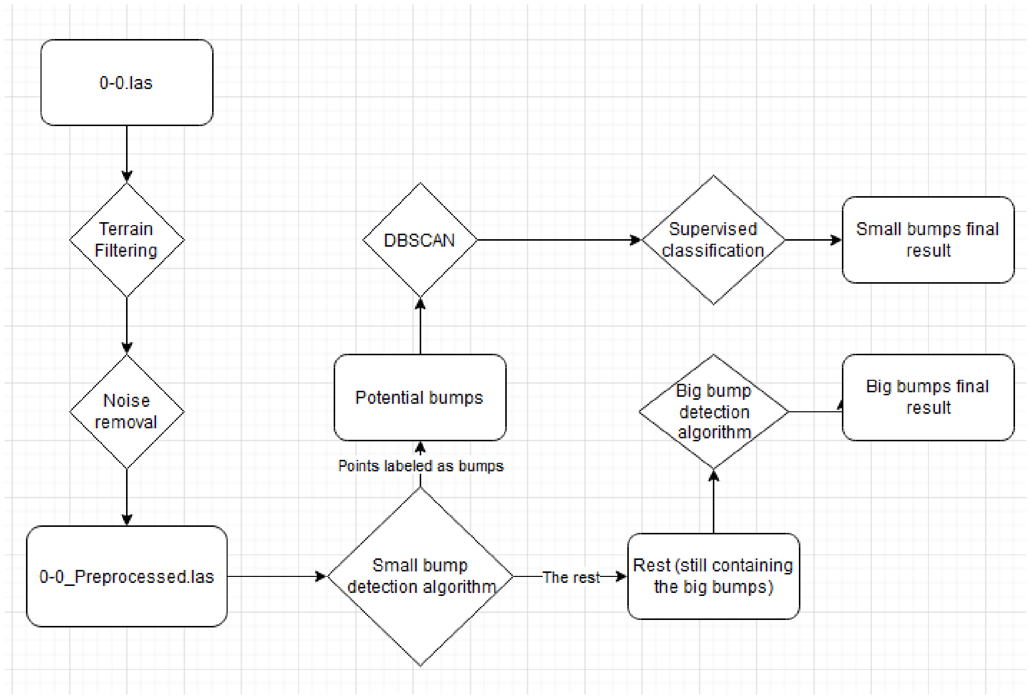

Figure 1: Workflow from the original point cloud to the final result.

This is the overview of what methods were used to get the final result. I will explain the specific implementations with code examples after presenting the results.

4 Results

4.1 Small bumps

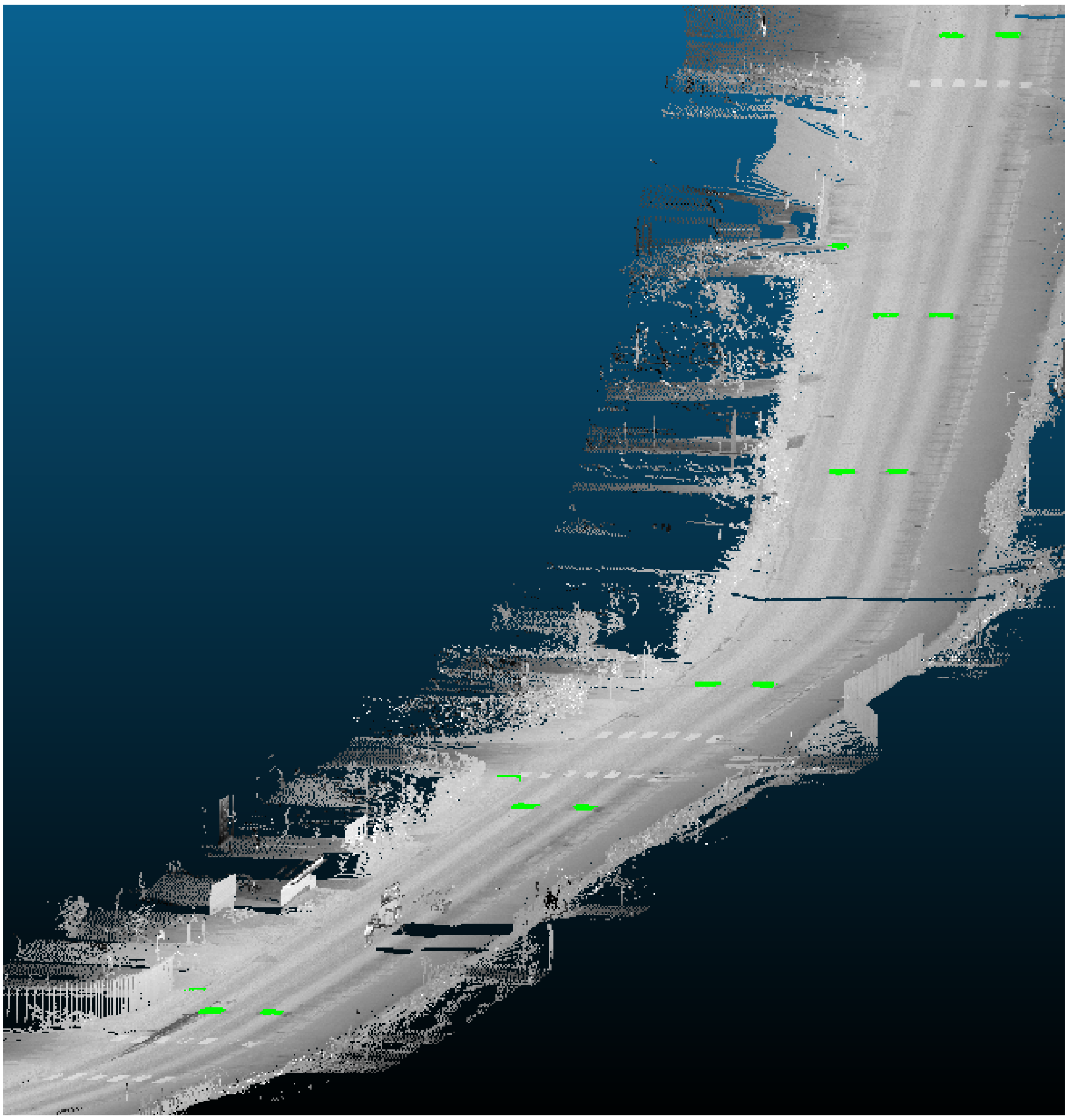

Figure 2: Overview of many small bumps detected. Good results, except some wrong detections of sidewalk as bumps.

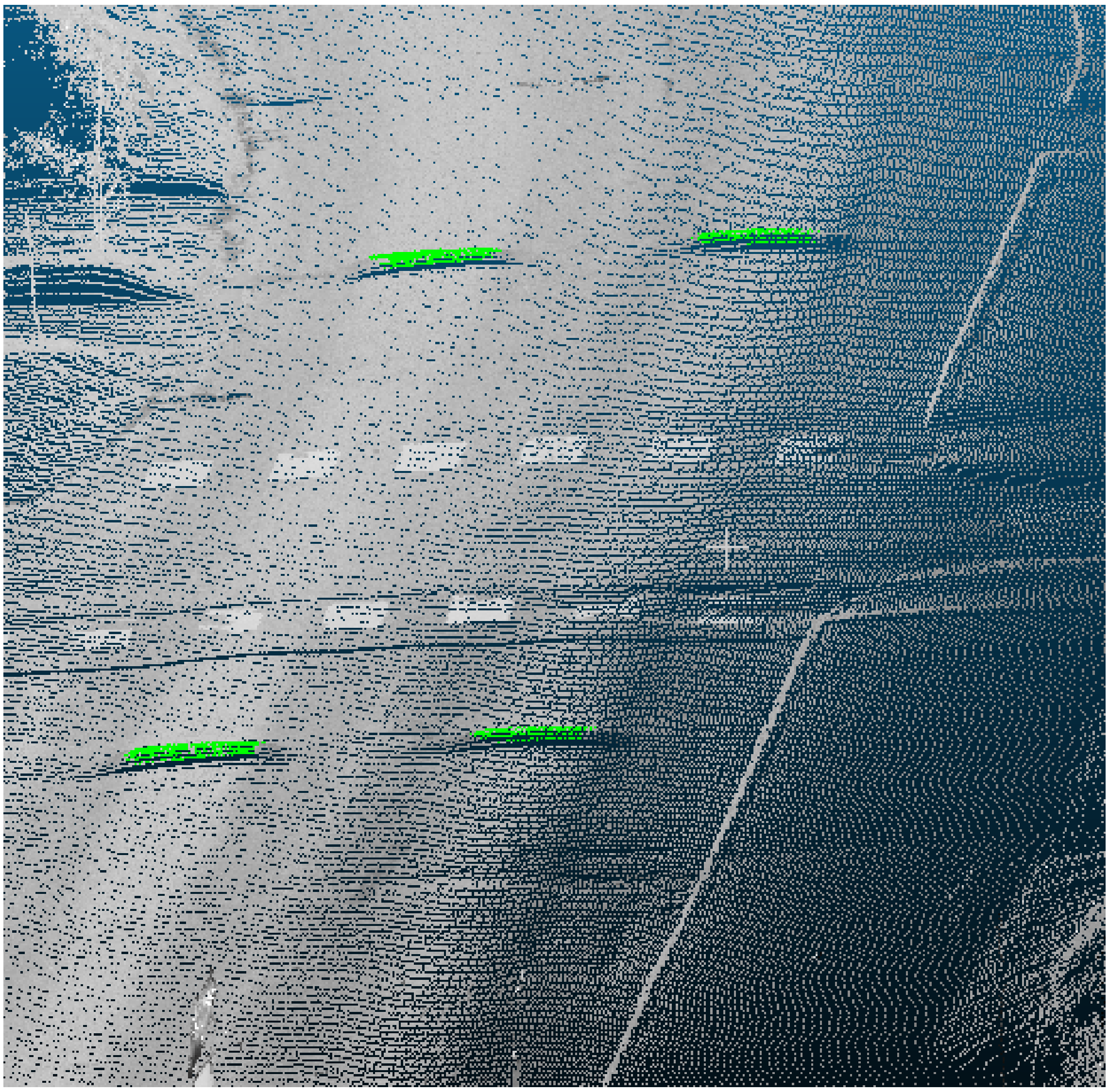

Figure 3: Closer look of two consecutive pairs.

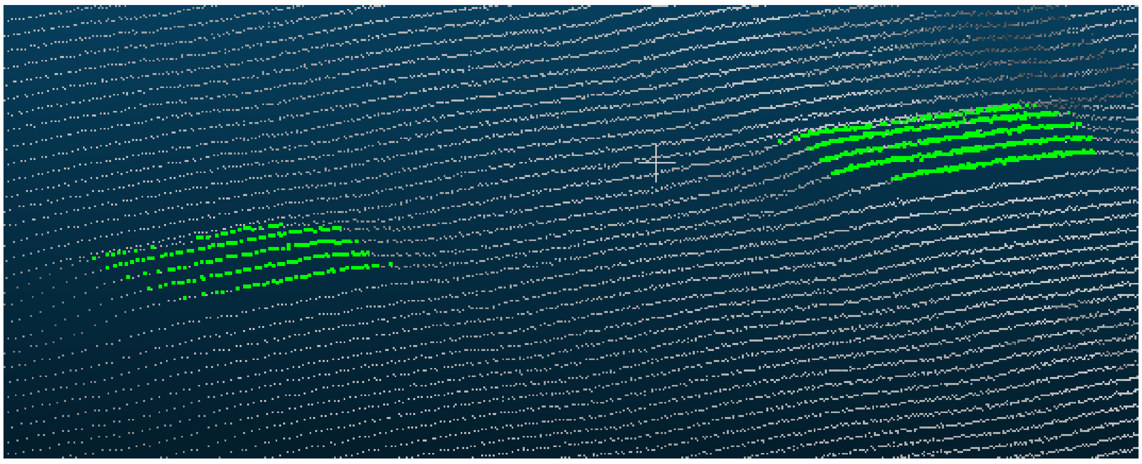

Figure 4: Close up of the best result. Most of the top points are detected.

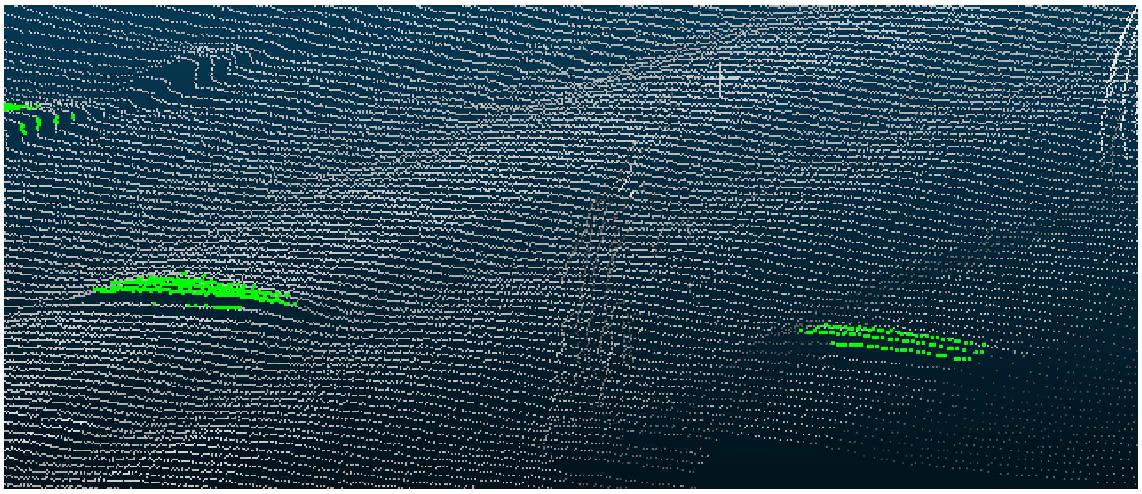

Figure 5: Close up of the worst correct result. Some points on top are not detected.

4.2 Large bumps

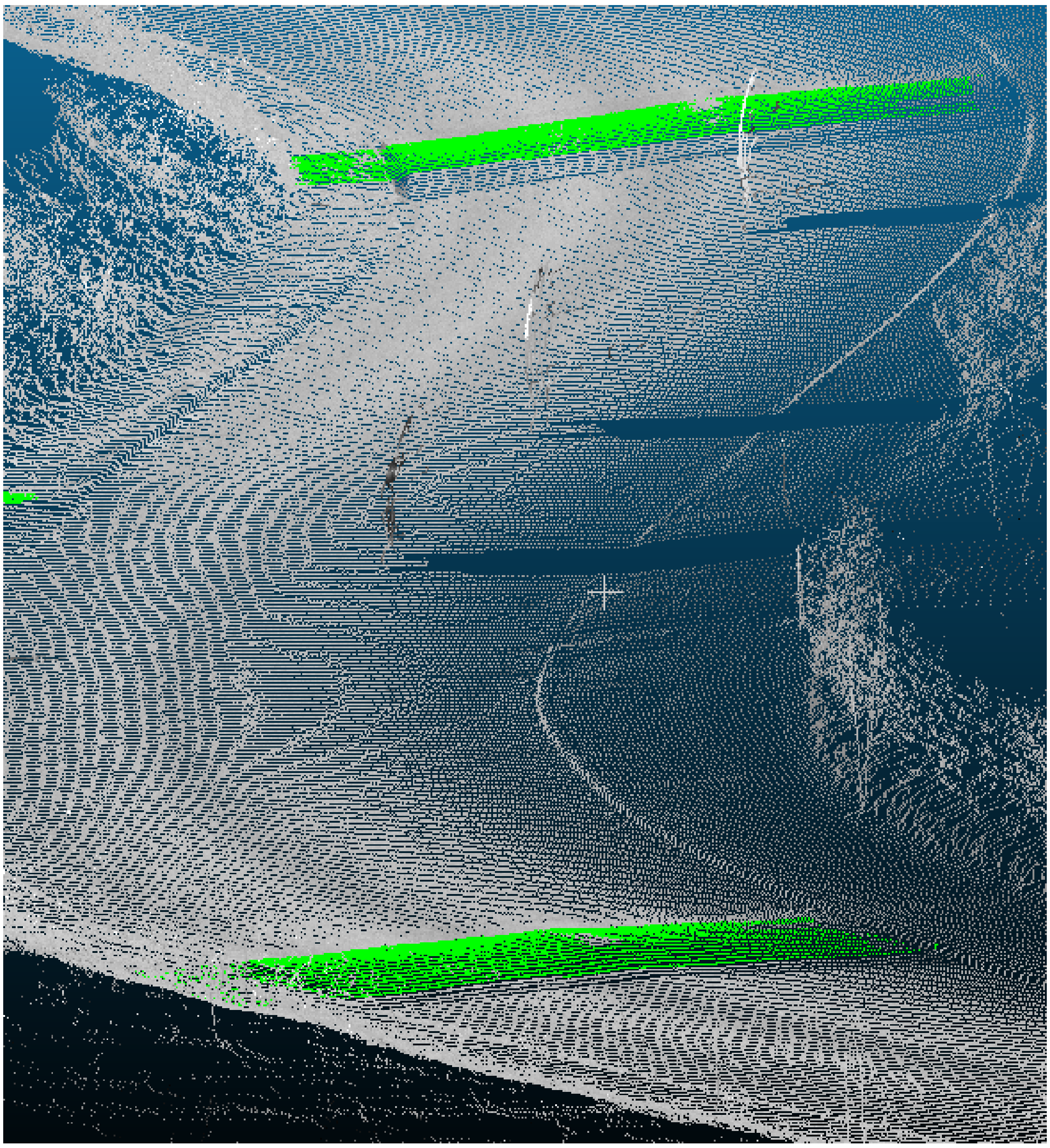

Figure 6: Result of two consecutive large bumps. Decent results.

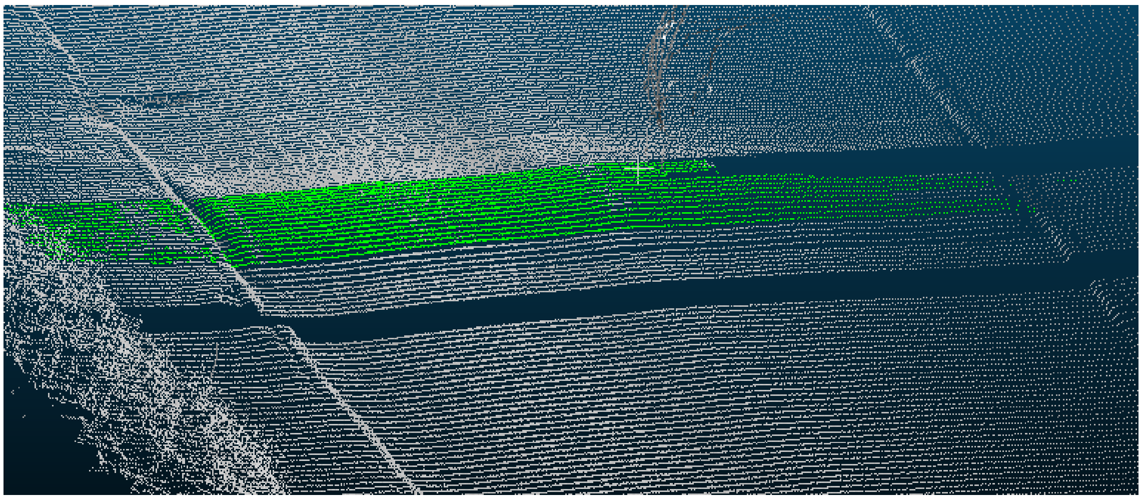

Figure 7: Close up. About half of the entire bump is classified. This was similar for all detected large bumps.

4.3 Wrong results

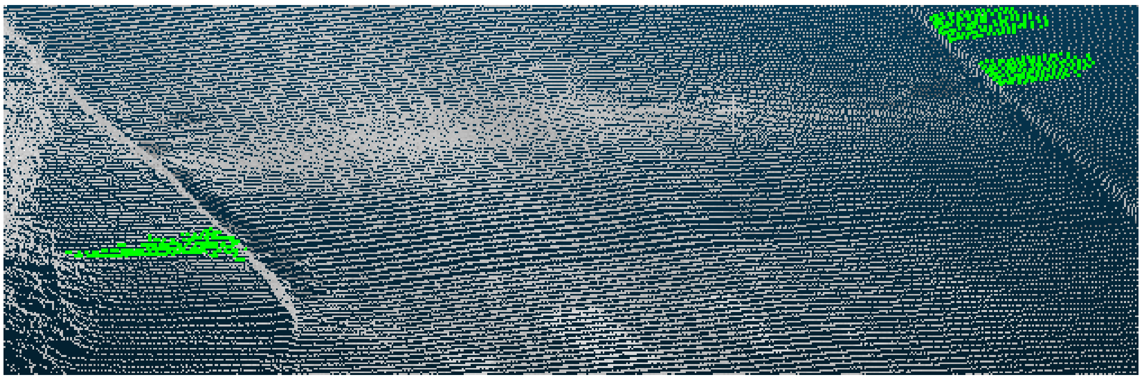

Figure 8: One of the large bumps which was completely missed.

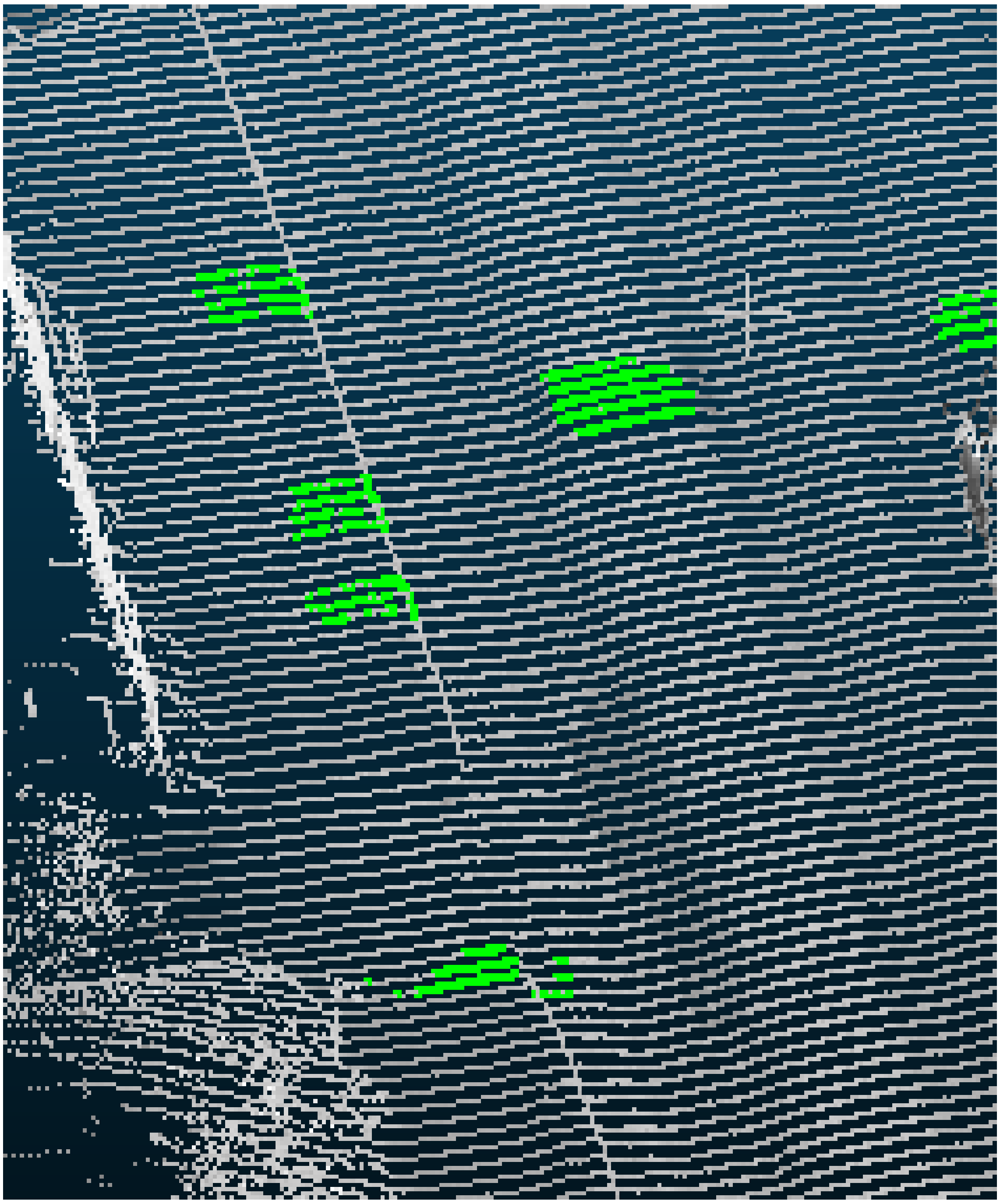

Figure 9: A few patches of sidewalk are wrongly classified as seen here.

4.4 Visualization in ArcGIS

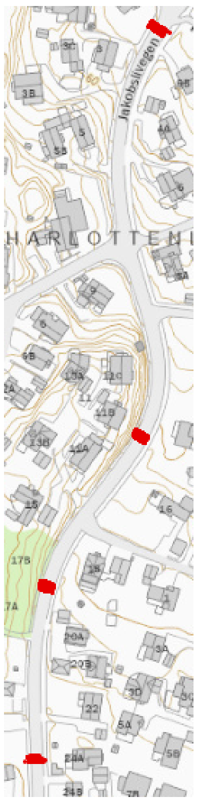
Figure 10: The four large bumps are clearly recognizable on the 2D map.

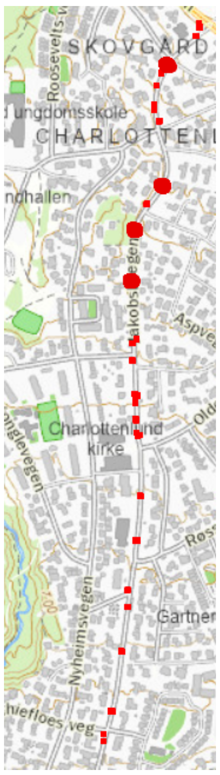
Figure 11: Wrong results in the small bumps ruins the result of the 2D visualization.

5 Evaluation

Overall my road segment contained 18 small bumps and 6 large bumps. I detected all of the small bumps but only 4 out of the 6 large bumps. Now lets look at precision and recall before discussing the results.

5.1 Precision and Recall

Precision and Recall are useful measures for understanding the quality of the results. These values can be calculated as follows:

$$Precision = \frac{TP}{TP + FP} \tag{1}$$

$$Recall = \frac{TP}{TP + FN} \tag{2}$$

I can calculate the precision accurately. TP +FP is simply the number of points in the result point cloud. Then I manually segment out the wrongly classified sidewalk in CloudCompare. The remaining points are true positives. I choose to separate the precision and recall of small bumps and large bumps because I used different algorithms for detecting them.

$$Precision_{small} = \frac{7440}{10963} = 0.68 \tag{3}$$

$$Precision_{large} = \frac{36104}{39996} = 0.90 \tag{4}$$

The recall can not be calculated accurately unless I were to manually label the bumps first. Such a manual labeling would be prone to human error, so I might as well do it approximately instead. For small bumps most of my results were similar to the ones shown in 4. From this I would argue an average recall of

$$Recall_{small} = 0.8 \tag{5}$$

for small bumps.

For the large bumps approximately half of the bump is detected, and with four out of six bumps detected we get:

$$Recall_{large} = \frac{4 * 0.5 + 2 * 0}{6} = 0.33 \tag{6}$$

5.2 Discussion

5.2.1 Small bumps

The results for the small bumps were good but not great. I'm happy with the recall but the precision is on the low end. One of the problems with a low precision is clearly shown in 11, where the 2D visualization of the results become meaningless if you cant trust it. Therefore, in hindsight, I would gladly sacrifice some recall for higher precision.

5.2.2 Large bumps

The results for the large bumps had the opposite problem. A very low recall but good precision. It could be argued that this is a better result than the small bumps, as 10 actually conveys meaningful information. Overall I'm content with this result.

6 Code

In this section I will explain my code in detail for each of the steps in the workflow, while showing code snippets. My approach for programming was to jump in head first and try to write my own algorithms.

6.1 Essential methods in pylas

First I would like to introduce some methods I used the most within pylas, which will make my code easier to understand.

- $las = pylas.read(inputfile)$: Loads las file into python variable.
- $las = pylas.write(outputfile)$: Writes new las file.
- $las = pylas.merge([pcd1, pcd2, ...]$: Merges a list of point clouds into a single point cloud.
- $las.points = las.points[condition]$: Filters the points, keeping the points matching the condition

6.2 Preprocessing

To make the detection of speed bumps easier I tried to isolate the road as much as possible. This was done by preprocessing the point cloud before working on it.

6.2.1 Terrain filtering

The goal with terrain filtering was to separate the ground points from non-ground points. To solve this I came up with the idea of dividing the point cloud into small enough chunks, so that the ground in the chunk would approximate a plane. Then I could use RANSAC to find this plane, and inliers will be ground points, and outliers are non-ground points. The RANSAC algorithm is pretty standard, and is based on my code from TBA4256. Below you can see the main part of the algorithm I used.

```
 # Here I segment the point cloud into chunks of 10000 points.
 # For each chunk I find the points that are inliers (ground points).
for i in range(0, all_point_count, 10000):
    temp.points = pcd.points[i:i+10000]

    # Parameters found by trial and error.
    rans = RANSAC(temp, 200, 0.09)
    best_interior_idx, rest = rans.RanSac_algthm()

    # save result
    newLas = pylas.create_from_header(temp.header)
    restLas = pylas.create_from_header(temp.header)
    newLas.points = temp.points[best_interior_idx]
    restLas.points = temp.points[rest]

    pcds.append(newLas)
    restpcds.append(restLas)
```

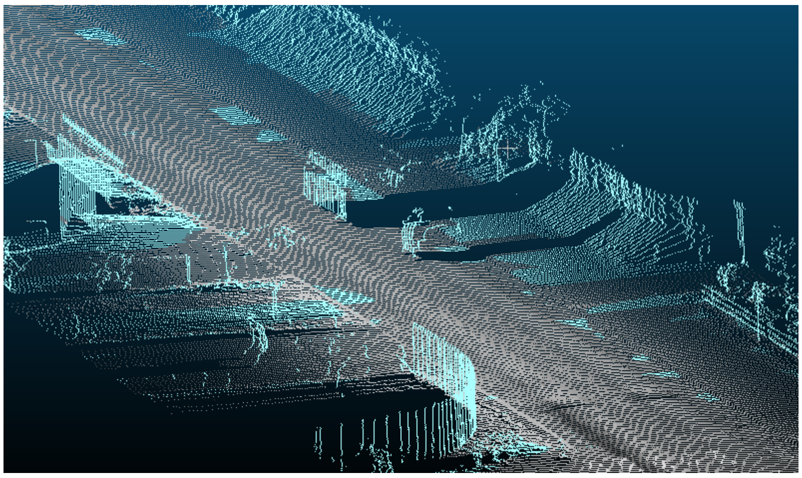

Figure 12: Example of points removed in terrain filtering(light blue). Cars, elevated ground and walls are all removed. The results were good over the entire point cloud.

6.2.2 Noise removal

The resulting ground points from terrain filtering were good but not perfect. Sidewalks and cross sections made the road segments wider than I hoped. To remove some of these unnecessary points I removed areas with low density and areas with high intensity. This is because the right side of the road is less dense, and the left side has higher intensity, to remove from both sides both need to be considered. Like earlier I processed small chunks at a time to speed up the processing time. I used the library scipy to make a KDTree to find neighbors of a point efficiently, then the number of neighbors represent the density. The code below shows how a chunk is processed.

```
# Here I remove noise by looking at how many neighbors
# each point has within a radius r.
# Additionally I remove points with an intensity over a certain value.

    # Extract the point cloud from the chunk
    pcd = np.column_stack((chunk_las.x, chunk_las.y, chunk_las.z))

    # Build a KD-tree data structure for the chunk
    kd_tree = KDTree(pcd)

    for i in range(0, chunk_point_count-1):
        # Specify the point for which to find neighbors
        point = np.array([pcd[i][0], pcd[i][1], pcd[i][2]])

        # Find the neighbors of the point within the radius r
        indices = kd_tree.query_ball_point(point, r=0.5)

        # Extract the coordinates of the neighbors
        neighbors = pcd[indices]
        # Most of the scalar fields are not present in 0-0.las
        # so I use gps_time as a placeholder attribute
        # to store relevant information
        # here i use it to mark points as noise.
        if len(neighbors)<60:
            las.gps_time[i+chunk] = 7
        elif las.intensity[chunk+i]>183:
            las.gps_time[i+chunk] = 7
```

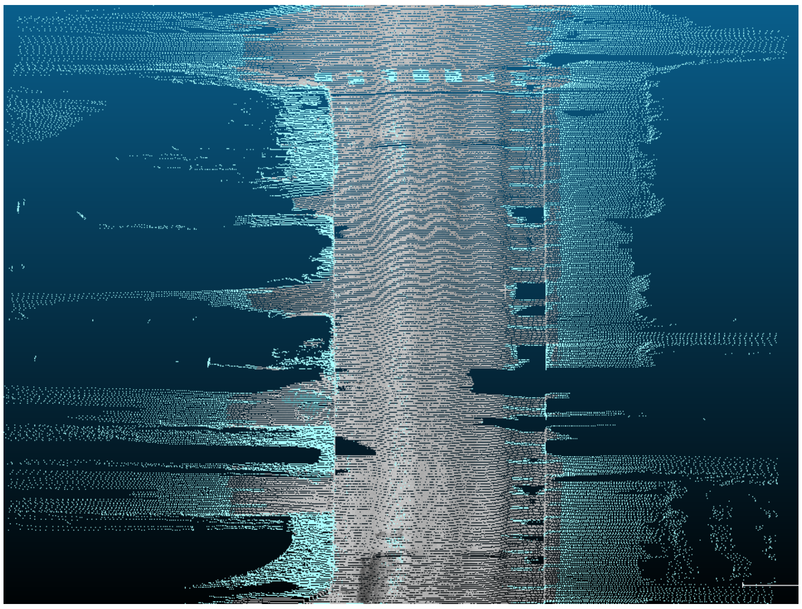

Figure 13: Noise (light blue). Now the road is isolated enough to begin bump detection.

6.3 Small bump detection

My idea for detecting small bumps was that points on the bump has a higher elevation than the mean elevation of its neighbors within a certain radius. This radius has to be big enough so that there are enough points beside the bump to drag down the mean elevation. A radius of 1 meter worked well. The problem with this approach, which you will see, is that sidewalks were extremely prone to being wrongly detected as a bump. The algorithm is identical to the one for noise removal, except the last part.

```
pcd = np.column_stack((chunk_las.x, chunk_las.y, chunk_las.z))
kd_tree = KDTree(pcd)

for i in range(0, chunk_point_count-1):
    point = np.array([pcd[i][0], pcd[i][1], pcd[i][2]])
    indices = kd_tree.query_ball_point(point, r=1.0)

    # Extract the z-coordinates of the neighbors
    z_neighbors = pcd[indices][:,2]

    meanZ = np.mean(z_neighbors)
    # If a point is significantly higher elevated than
    # its neighbors, it could be a speed bump.
    if chunk_las.z[i]>meanZ+0.025:
        las.gps_time[i+chunk]=8
```

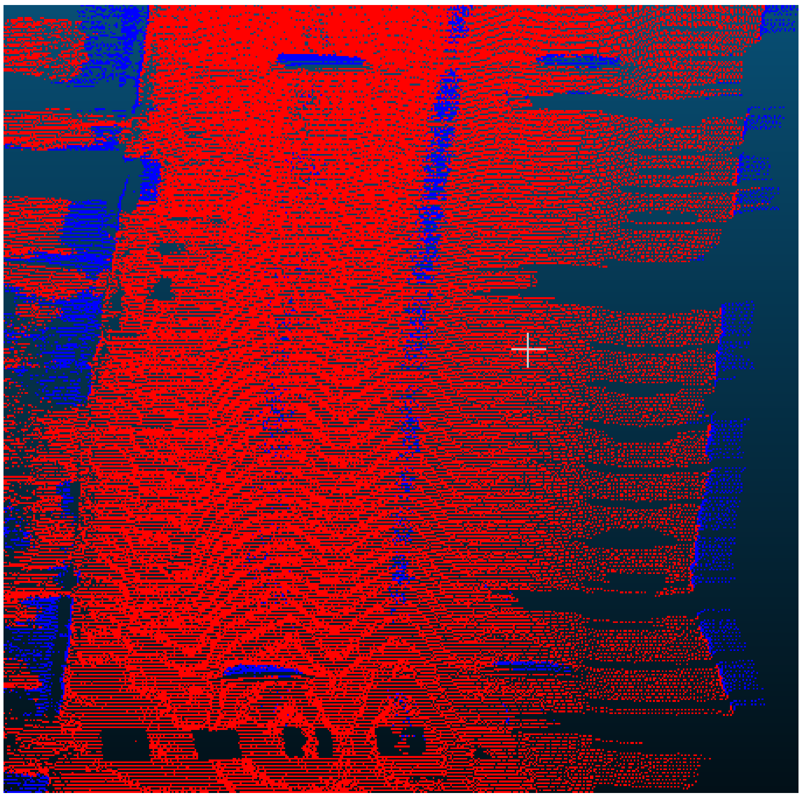

Figure 14: The blue points are potential bumps. The bumps are detected but so are sidewalks and the highest part of the road. Somehow the bumps need to be separated from the others.

6.4 Separating small bumps from sidewalk

A key observation here is that the bumps can easily be detected in a clustering algorithm since these points are highly concentrated. Furthermore the size of a bump cluster is predictable, so clusters above and below certain values can be removed. I used DBSCAN from the Python library sklearn as my clustering algorithm. The results from the initial clustering were good, but there were still many sidewalk edges present. I noticed that these edges were often linear, so I combined the clustering with the linearity feature found from PCA on each cluster. Clusters with high linearity were removed. This worked surprisingly well. The code is too big to show here, but I will write a rough pseudocode of the algorithm. The full code can be found in the zip file provided.

```
clustering = DBSCAN.fit(pcd)
for unique_label in clustering.labels:
    indices = np.where(clustering.labels==unique_label)
    cluster_pcd.points = pcd.points[indices]
    # Too big clusters are not bumps
    if cluster_pcd.header.point_count <800:
        eigvals = eig(covMatrix(cluster_pcd))[0]
        e1,e2 = [eigvals[0],eigvals[1]]
        linearity = (e1-e2)/e1
        # Too linear clusters are not bumps
        if linearity <0.8:
            # Will be part of the final result
            result.append(cluster_pcd)
```

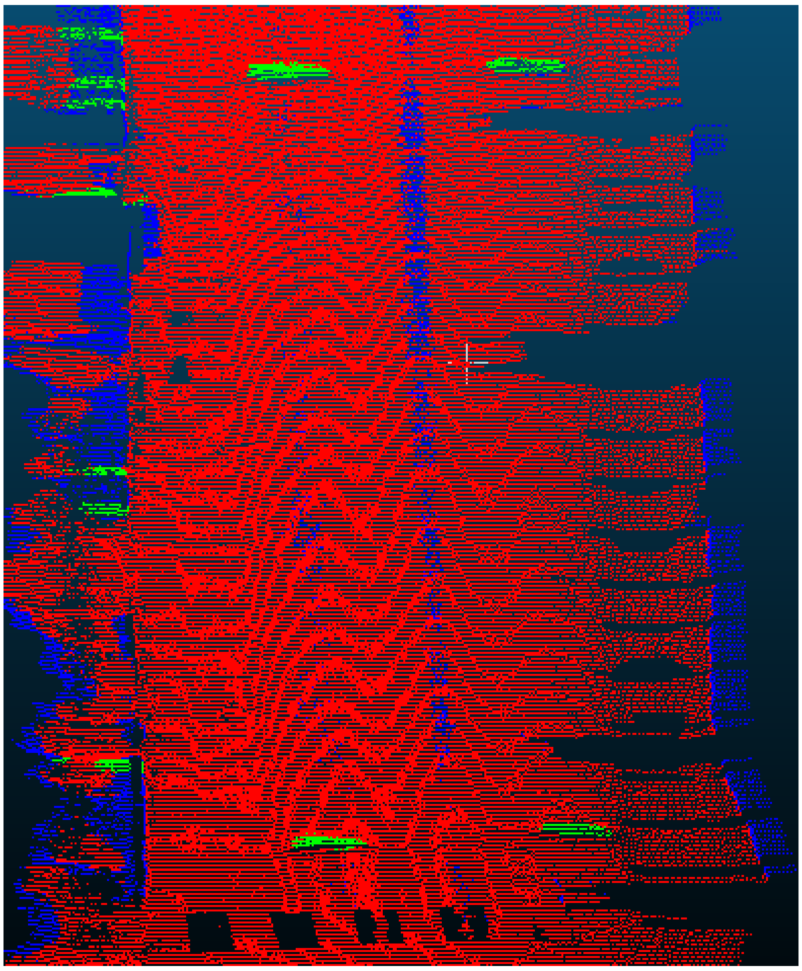

Figure 15: The green points are the surviving points after clustering. Now almost all wrong points from 14 are avoided.

This conclues the detection of small bumps.

6.5 Large bump detection

The large speed bumps differ from the smaller ones in both size and structure, so I decided to detect them separately. Since the bumps are so large I decided against using point neighborhoods, as the neighborhood radius would be too big to compute efficiently. Instead I analyzed entire chunks at a time. If a chunk is significantly higher elevated than itself extended in both directions it is labeled as a large speed bump. This didn't work great, as seen by the recall value, but it was the best I could come up with.

```
# I process the point cloud chunk by chunk.
for chunk in range(0, point_count-1, 10000):
    # Current chunk
    chunk_las.points = las.points[chunk:chunk+10000]
    # Slightly bigger chunk expanding the current one on both sides.
    surrounding.points = las.points[chunk-3000:chunk+10000+3000]
    chunk_point_count = chunk_las.header.point_count

    local_meanZ = np.mean(chunk_las.z)
    meanZ = np.mean(surrounding.z)
```

```
#If the current chunk is significantly higher elevated than the
# extended chunk we have detected a big speed bump.
if local_meanZ-meanZ>0.013:
    for i in range(0, chunk_point_count -1):
        # Mark as speed bump
        las.gps_time[i+chunk]=8
```

7 Discussion

In this section I want to discuss some of the factors that affected the results and the work process.

7.1 Parameters

Until now I have not discussed the parameters used in my code, and how I determined them. This is because all of them were set through trial and error. By continuously changing them and visualizing the results in CloudCompare I found which values worked the best. I recognize that this is a somewhat lazy approach, and it would be better to determine them on a more general basis. For instance I doubt the results would be great if I tried my code on a new road.

7.2 Point cloud challenges

7.2.1 Asymmetrical

One of the biggest challenges I encountered was the asymmetric nature of my point cloud. The density and the intensity of points decreased from the left side to the right side of the road. This can be seen in figure 6. This heavily affected the result of the preprocessing as the points on the right side were more easily removed. This effectively shifted the preprocessed point cloud towards the sidewalk on the left side, ultimately making my results worse.

7.2.2 Gaps

Another challenge I encountered was gaps of empty space in the road, as can be seen below.

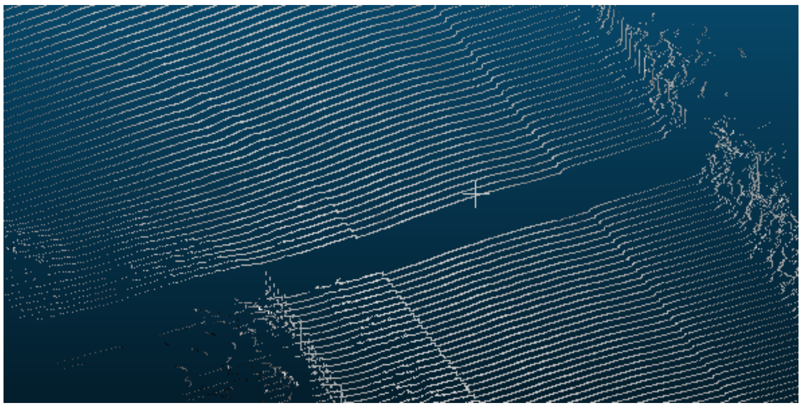

Figure 16: Gap with no points.

This interfered with my algorithm for detecting large bumps, because that algorithm is dependent on the presence of points on both sides of each chunk. Chunks close to these gaps was more likely to be wrongly labeled a large bump. To keep precision high I had to make sure these wrong labels did not happen, however at the cost of failing to detect two of the six large bumps.

8 Conclusion

Overall this has been a difficult, but interesting project. Although I'm not completely happy with my results I enjoyed all the problems that had to be solved along the way. My solution was far from elegant, but I did find it fulfilling to use all the tools in the toolbox to produce a result(RANSAC, segmentation, PCA, clustering, neighborhood). It was a good learning experience for processing point cloud data.