[Final project 5] Object Detection in Point Cloud: Lane Marking

June 5th, 2019

Zunran Guo, Feiyu Chen

Lane detection result

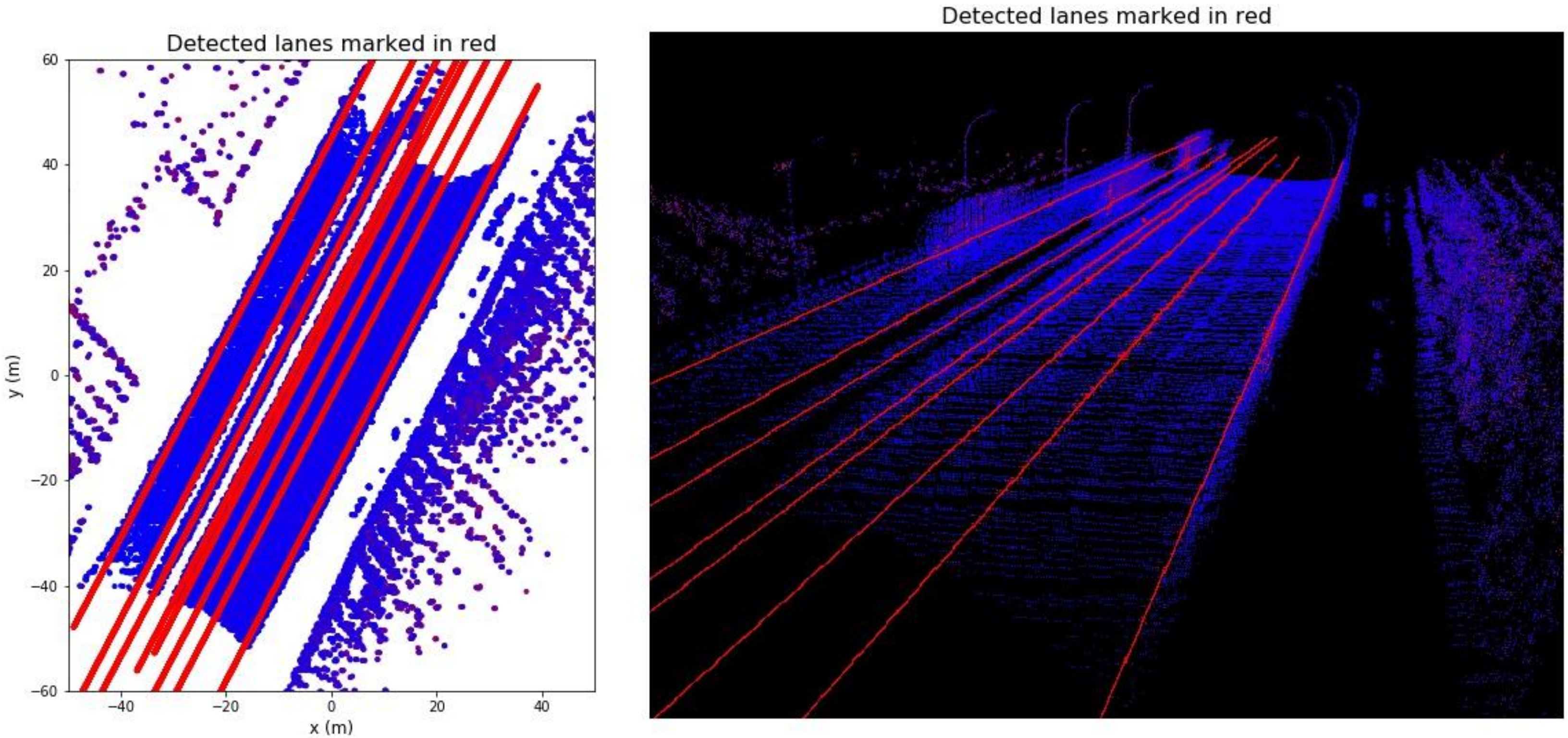


Figure: Eight lanes are detected from the point cloud data of a street.

Left: Point cloud projected on x-y plane. **Right:** 3D point cloud.

Lane detection

Detecting target objects from point cloud data is an essential task in HD Map processing and sensor-based autonomous driving.

Common types of objects on the road include: 1) road surface, 2) lane markings, 3) pavement area, 4) support facilities (such as guardrail and curb), 5) signs, 6) light poles,7) and other unrelated objects such as trees, pedestrians, and buildings.

In this project, we aimed at detecting the lanes automatically from point cloud data, which is crucial for the decision making of self-driving cars.

Method overview

To detect the lane, we summarize the attributes of the lane as follows:

1. The lane is in the road surface which is a thin and flat region in point cloud data.
2. Lane points have larger reflection intensity than other road points.
3. The shape of a lane is piece-wise straight.
4. Lanes are parrerel to each other

Based on the above observations, we designed the following procedures for detecting lanes from point cloud data:

1. Remove non-planar regions, and then detect the road surface.
2. Thresholding on point intensity to get possible lane points.
3. Estimate the direction of lanes.
4. Do clustering to get each lane, and estiamte their line equations.

Overview

The first step is to preprocess the point cloud data, which includes:

1. Change points’ coordinate from (latitude, longitude) to (x, y)
2. Downsample
3. Filter noise

Change coordinate: (lat, lon) to (x, y)

The raw point cloud data in “final_project_point_cloud.fuse” is stored in the following order for each row:

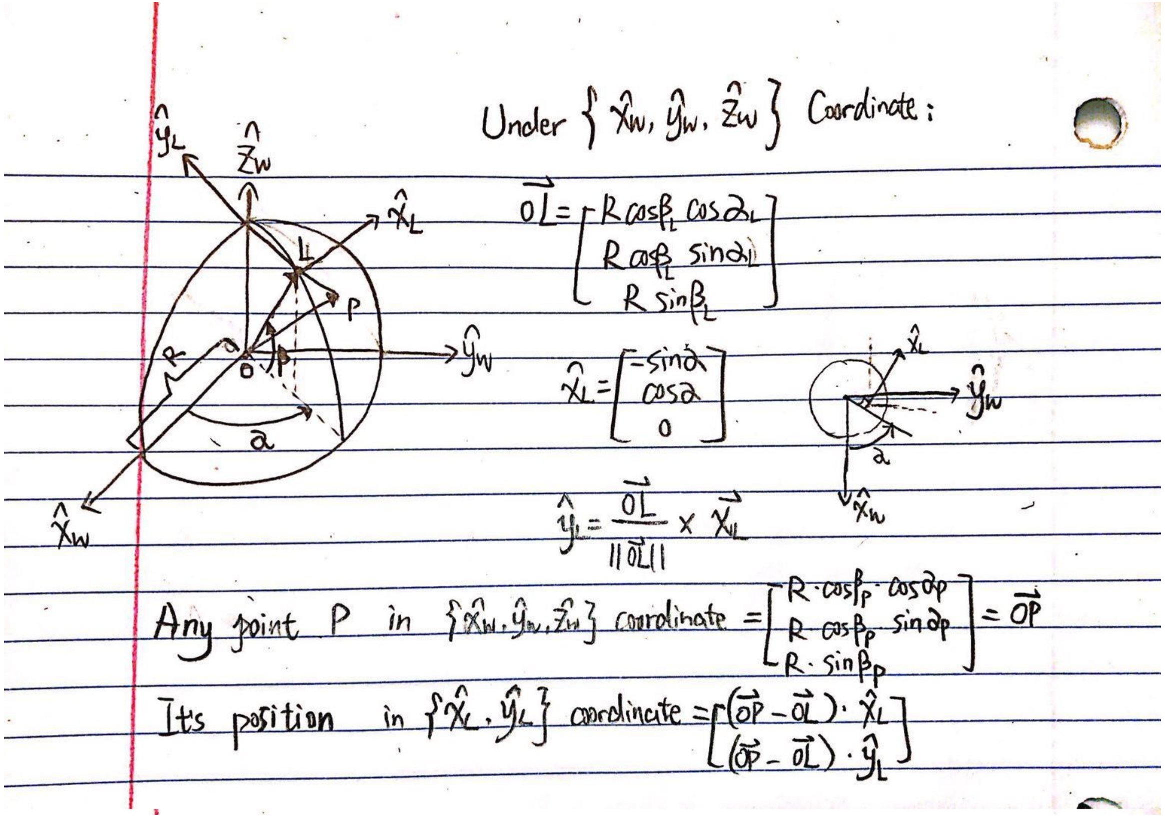
1. latitude (degree)
2. longitude (degree)
3. altitude (meter)
4. point reflexion intensity (0~100)

For more intuitive representation and processing, we transformed the points’ coordinate from (latitude, longitude) to a local Cartesian coordinate (x, y) established at the average position of all cloud points.
We re-used the code in Homework2 to accomplish this step. The formula is shown in next slide.

Change coordinate: (lat, lon) to (x, y)

Math formula to change coordinate from global (lat, lon) to a local Cartesian (x, y):

- Global coord is {O}
- Local coord is {L}
- The point we want to change coordinate is P


$\hat{y}_L$
$\hat{z}_W$
$\hat{x}_L$
L
P
$\hat{y}_W$
R
O
α
$\hat{x}_W$
$\hat{x}_L$
$\hat{y}_W$
α
$\hat{x}_W$

Under $\{\hat{x}_W, \hat{y}_W, \hat{z}_W\}$ Coordinate:

$$\vec{OL} = \begin{bmatrix} R\cos\beta_L \cos\alpha_L \\ R\cos\beta_L \sin\alpha_L \\ R\sin\beta_L \end{bmatrix}$$

$$\hat{x}_L = \begin{bmatrix} -\sin\alpha \\ \cos\alpha \\ 0 \end{bmatrix}$$

$$\hat{y}_L = \frac{\vec{OL}}{\|\vec{OL}\|} \times \vec{x}_L$$

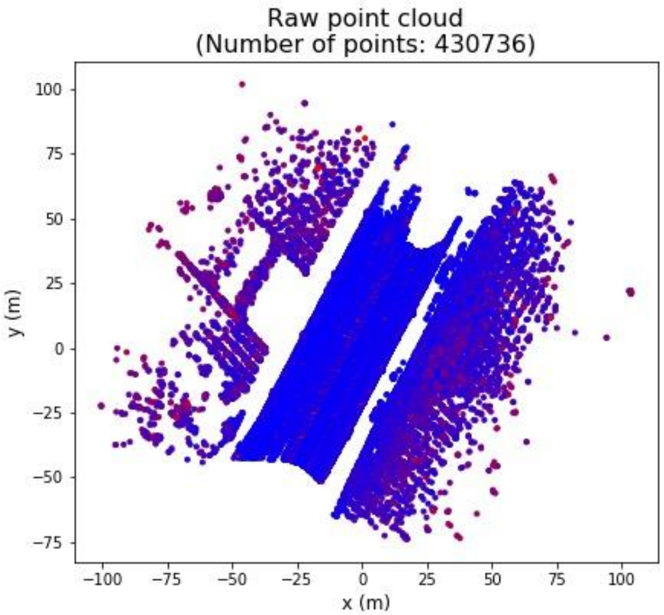
Any point P in $\{\hat{x}_W, \hat{y}_W, \hat{z}_W\}$ coordinate $= \begin{bmatrix} R\cdot\cos\beta_P\cdot\cos\alpha_P \\ R\cdot\cos\beta_P\cdot\sin\alpha_P \\ R\cdot\sin\beta_P \end{bmatrix} = \vec{OP}$

It's position in $\{\hat{x}_L, \hat{y}_L\}$ coordinate $= \begin{bmatrix} (\vec{OP}-\vec{OL})\cdot\hat{x}_L \\ (\vec{OP}-\vec{OL})\cdot\hat{y}_L \end{bmatrix}$

```
Code: lib_cloud_proc.py
def get_xy_from_latlon(lats, lons)
```

Preprocessing

Change coordinate: (lat, lon) to (x, y)



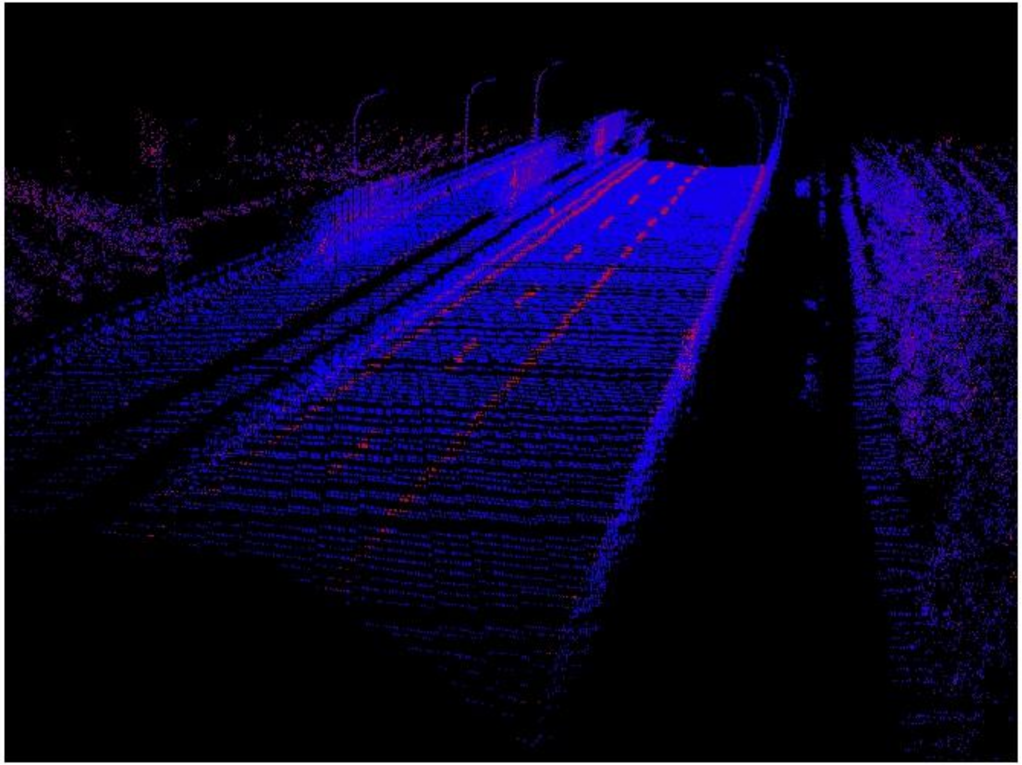
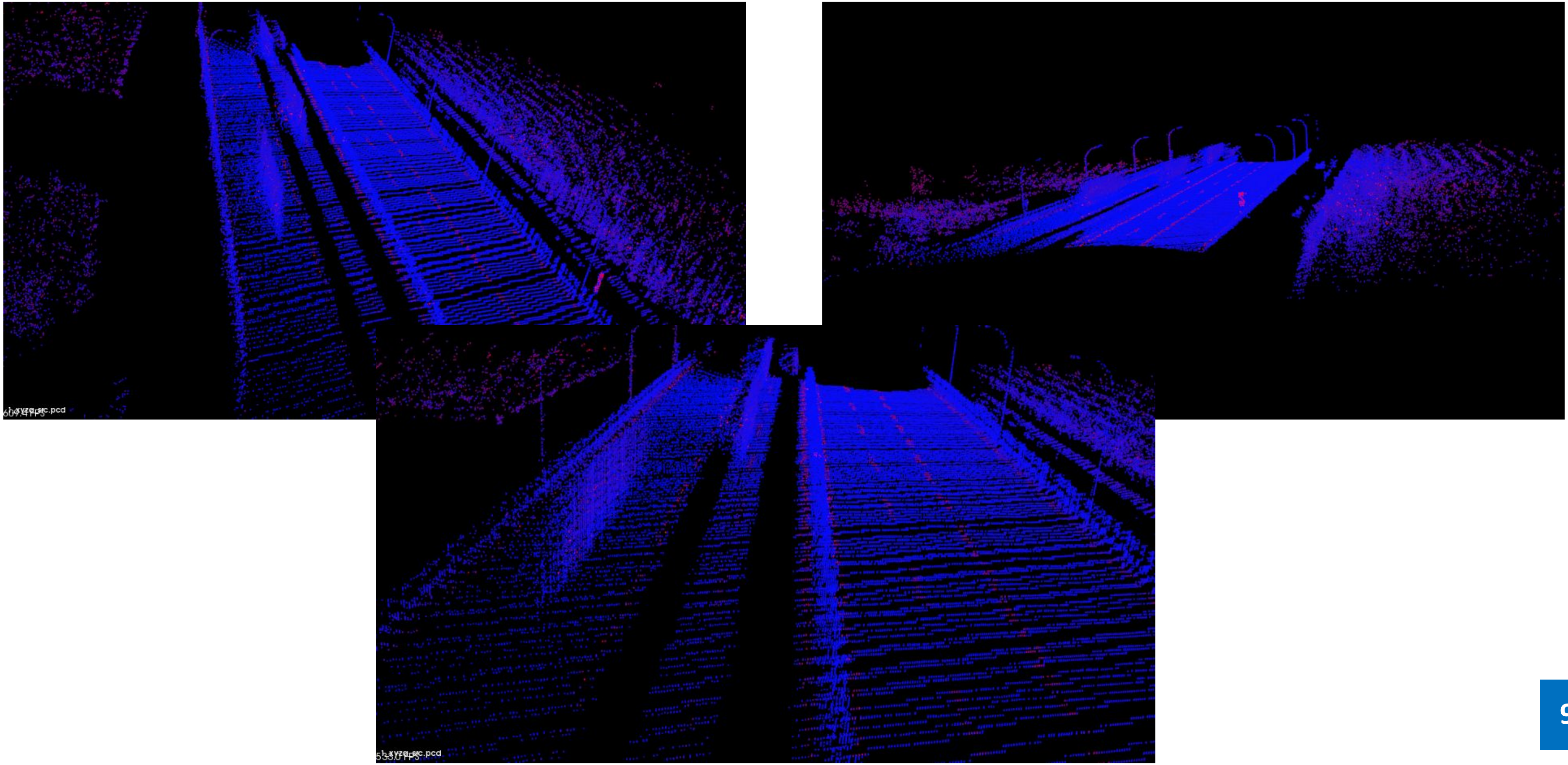


Figure: **Left:** Point cloud projected on x-y plane. **Right:** 3D point cloud. Points with higher intensity (i.e. more reflective) are more red.

Point cloud from different views



Downsampling

After changing coordinate, we downsampled the point cloud while taking the maximum of point intensities.

Steps:

For each voxel of size 0.1m in the world:

    Find all points {Pi} inside the voxel. Replace them with a single point Q.
    Q is placed at the center of the voxel.
    Intensity of Q is set as the max of intensities of {Pi}.

Reasons of downsampling:

1. Reduce points number to speed up processing.
2. Fix the density of point cloud for easier setting algorithms’ parameters.
3. Increase the intensity of lane points to make lane detection easier.

```
Code: lib_cloud_proc.py
def downsample(xyza, voxel_size=0.01, option='max_alpha')
```

Radius Outlier Removal

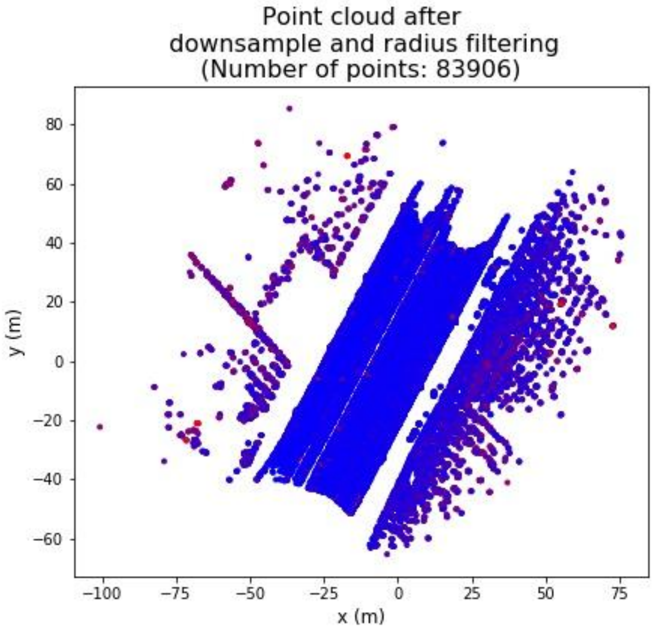
Points on road surface are adjacent to each other, while noise points are distance from neighboring points. Thus, we used a “radius outlier removal” algorithm to remove the noise points.

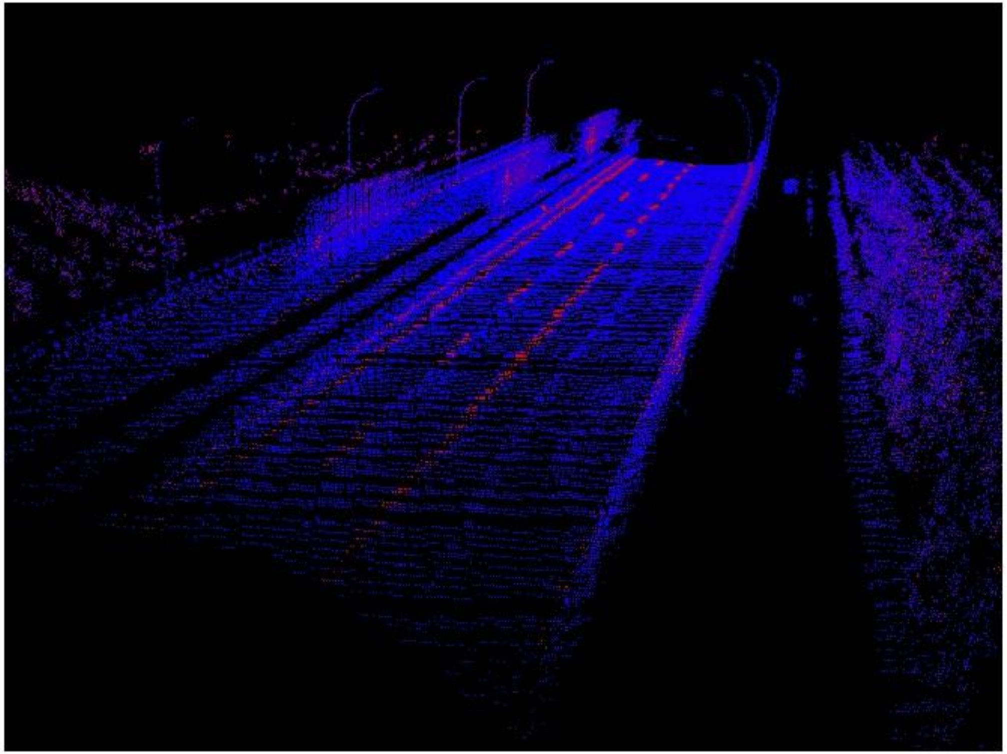
Steps: For a point, find all its neighbor points with a distance <= radius. If the number is smaller than nb_points, this point is considered as a noise point.

We use a function from the open-source library “open3d” to achieve this:

```
cloud, inliers_ind = open3d.radius_outlier_removal(
   cloud,
   nb_points=2,
   radius=0.5)
```

Downsample + Radius Outlier Removal





Result: Number of cloud points are reduced from 430k to 84k.
The noise points are also removed.

Remove non-planar regions

Observation: Lanes are on the road surface, which is planar.
Signs, trees, supporting facilities are non-planar.

In this step, we removed the regions that are non-planar, in order to narrow down the possible points of road surface and lanes.

Significance of this step:
If we didn’t take this step and directly fit a plane to the point cloud, the non-road points (mainly the points on two sides of the street) that are on the same plane of the road surface will also be included in plane detection result, which is undesired.

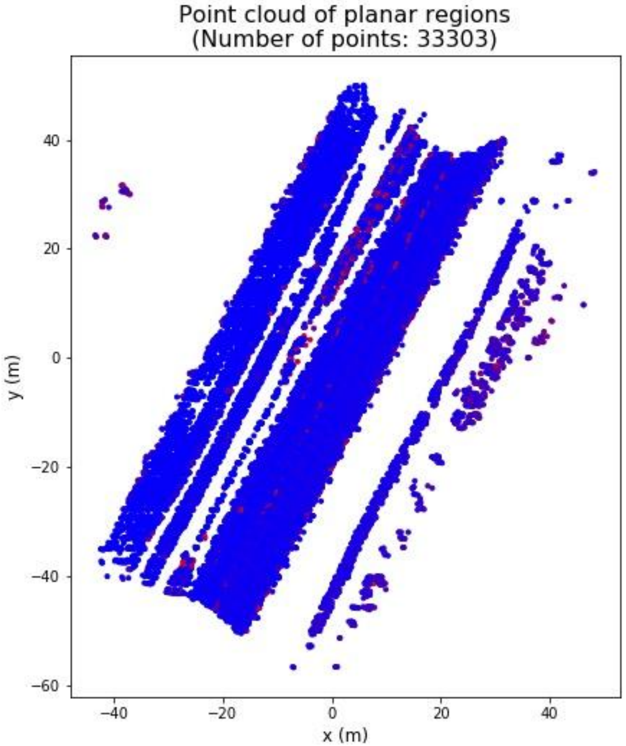
Remove non-planar regions

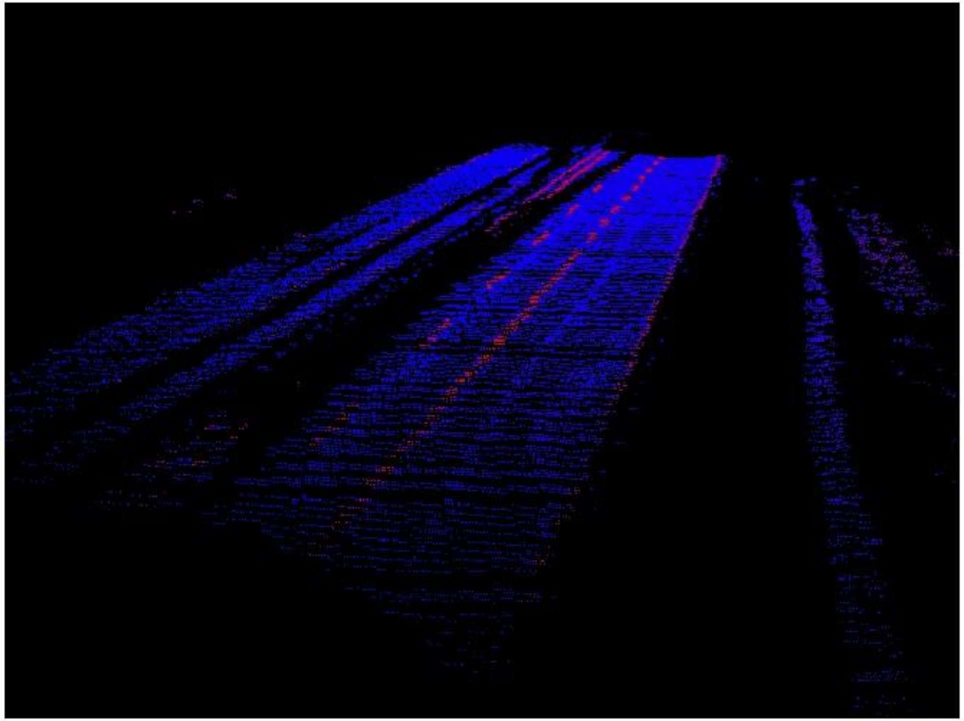
Steps of the algorithm:

For each point, find its nearest 10 neighbors (using kd-tree). If the difference of the max and min heights of these neighbors are larger than 0.1m, this point is considered in a “non-planar region”, and is removed.

```
Code: lib_cloud_proc.py
def find_plannar_points_by_kdtree(xyza, num_neighbor, max_height)
```

Remove non-planar regions





Result: Points on two sides of the street are removed.
The result points are mainly the street surface.

Detect road surface by RANSAC

Observation: Lanes are in the road surface, which is planar.
Goal: Detect the road surface, and remove the points not in this plane.
Method: Fit plane by PCA and RANSAC.

Method detials:
Fit plane by PCA: If we want to fit a plane to a set of points, we could use PCA to estimate a least principle axis of all points, which is the normal direction of the fitted plane.

RANSAC: This is an algorithm used for selecting: inliers (points of plane), and outliers (noise points). The basic steps are: (1) randomly sample a subset of points to fit a model. (2) Finetune the model. (3) If the model is better than previous ones, record it. (4) Repeat step (1) until some criteria is satisfied.

Detect road surface by RANSAC

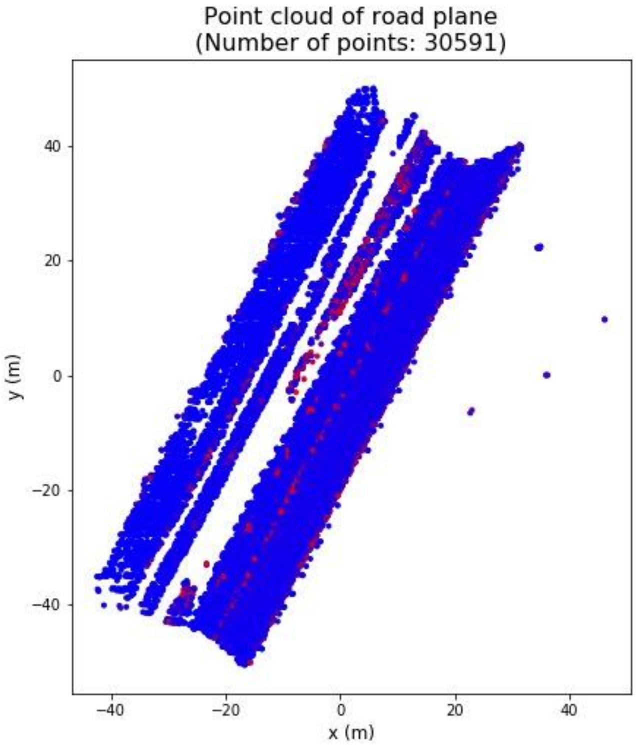
We modified the RANSAC algorithm from
https://scipy-cookbook.readthedocs.io/items/RANSAC.html
by (1) changing the criteria of a best model from “least error” to “most points”, (2) limiting the number of points for fitting model to avoid the memory error when calling SVD in PCA. (3) changing the strategy of selecting inliers.

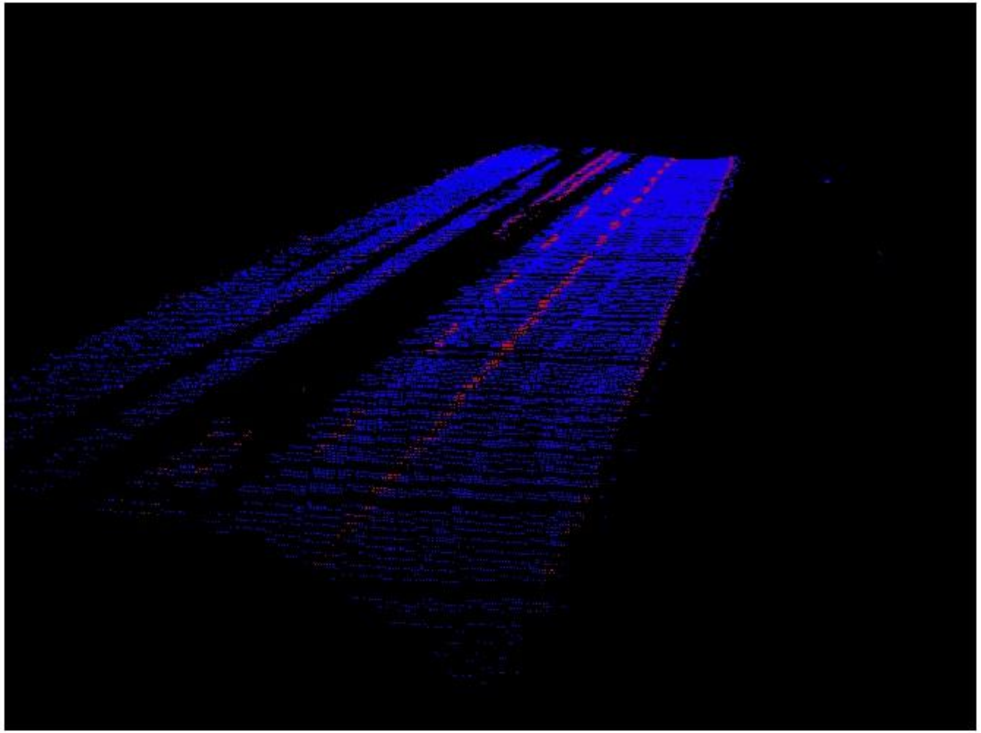
The main code for fitting a plane with parameters **w** is:

```
points_xyz = xyza_planar[:, 0:3]
plane_model = PlaneModel(feature_dimension=3)
w, inliers = ransac(
    points_xyz, plane_model,
    n_pts_base=3, n_pts_extra=50,
    max_iter=100, dist_thre=0.3,
)
```

Functions definition: see **lib_plane**.py and **lib_ransac**.py

Detect road surface by RANSAC



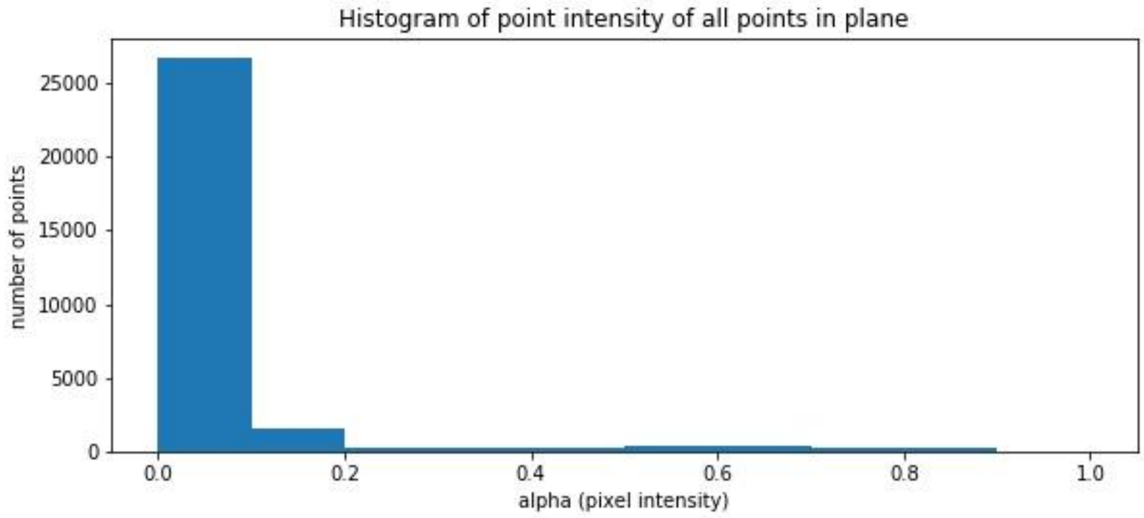


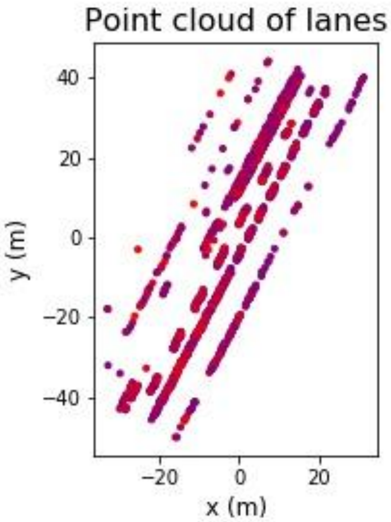
Result: The remaining points are mainly the road surface. Good!

Threshold on point intensity

The point reflexion intensity has a range of [0, 1].
After experiment, we found that the lane points have an intensity higher than 0.5.

Thus, we set a threshold of 0.5 to select possible lane points.
The result is shown below:



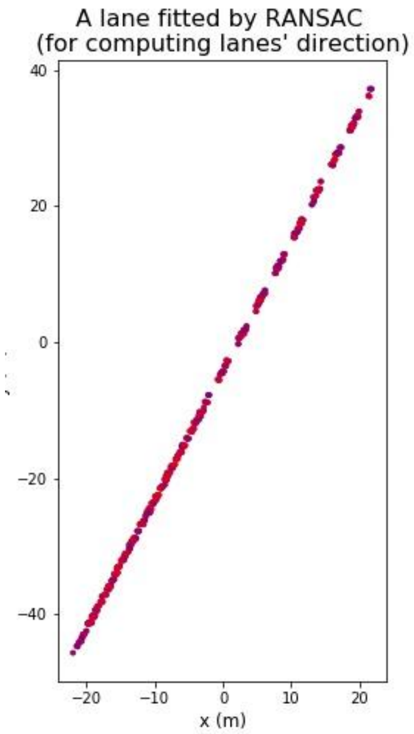


Use RANSAC to find lane direction

We again used PCA and RANSAC to fit a 2D line to the point cloud on (x, y) plane, in order to obtain the lane direction. (We assume all lanes are straight and parallel).

Code:

```
points_xy = xyza_lanes[:,0:2]
line_model = PlaneModel(feature_dimension=2)
w, inliers = ransac(
   points_xy, line_model,
   n_pts_base=2, n_pts_extra=10,
   max_iter=100, dist_thre=0.3,
)
print("Line fitting: {} --> {}".format(
   points_xy.shape[0], inliers.size ))
xyza_lane = xyza_lanes[inliers, :]
```



Step1: Feature reduction

In the previous step, we've detected one lane and its direction.
In this step, we want to obtain all lanes. What we do is:

"Project points to the normal of the lane direction, and do clustering"

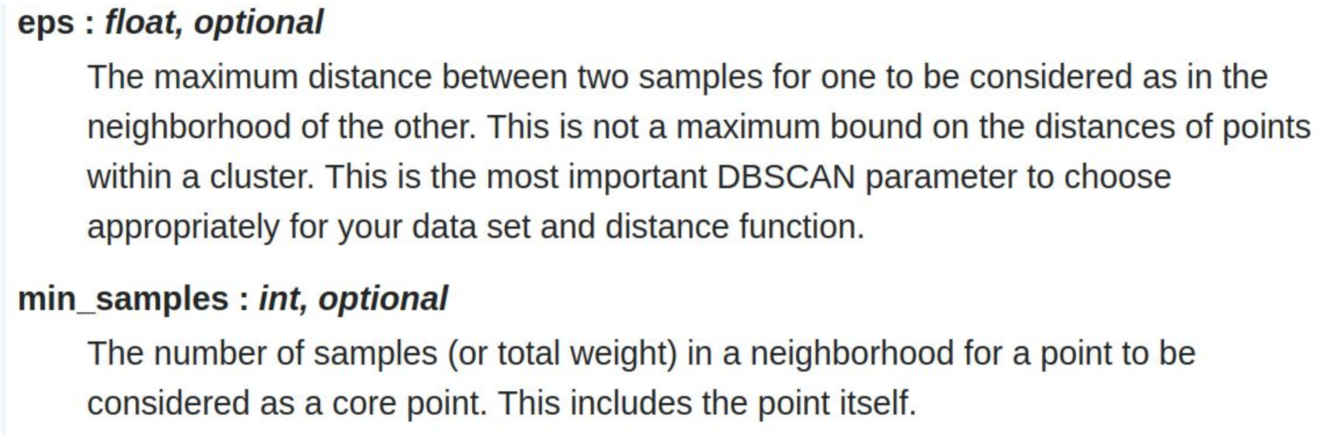
Since all lanes have a same direction, we could squash the points along this direction to ***reduce this redundant feature dimension***. It is done by projecting all points to the normal of the lane direction:

```
# get line's normal from line parameters w
norm_direction = w[1:] # w[0] + w[1]*x + w[2]*y = 0
print("Normal of lane: {}".format(norm_direction))

# project points
x = points_xy.dot(norm_direction)
y = np.zeros_like(x)
projections = np.column_stack((x, y))
```

Step2: Clustering

After projection, the points of different lanes can be easily separated from each other by using a clustering algorithm.

The **DBSCAN** algorithm is adopted (see this link) due to its good performance. It could do the clustering without manually specifying the number of clusters. Besides, it’s good at dealing with noisy data and clustered data samples. The 2 major parameters of this DBSCAN are:



eps : *float, optional*

The maximum distance between two samples for one to be considered as in the neighborhood of the other. This is not a maximum bound on the distances of points within a cluster. This is the most important DBSCAN parameter to choose appropriately for your data set and distance function.

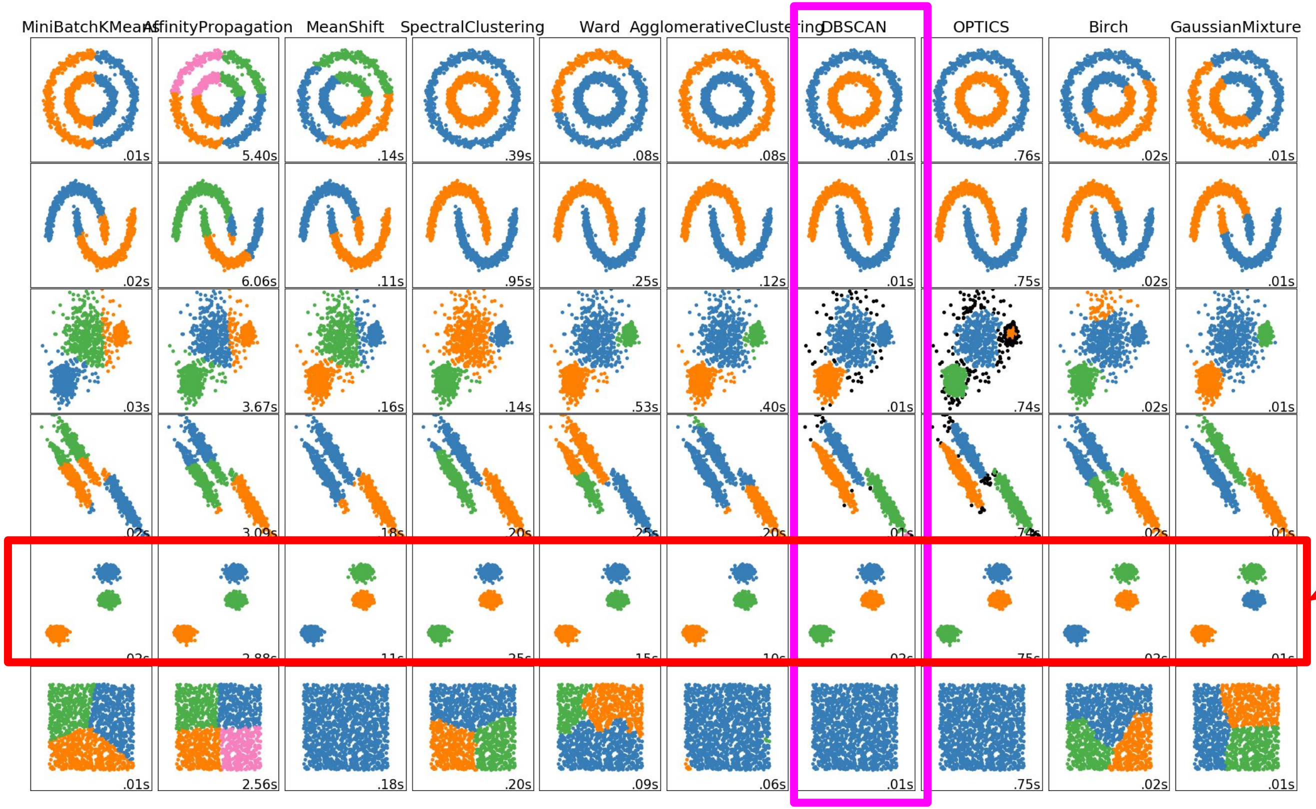
min_samples : *int, optional*

The number of samples (or total weight) in a neighborhood for a point to be considered as a core point. This includes the point itself.

Detect lanes by clustering

Step2: Clustering

Comparison of different clustering algorithm (see this link).



Data distribution in our task is most similar to this row. We can see that not only DBSCAN, but all other algorithms should also work well for our case.

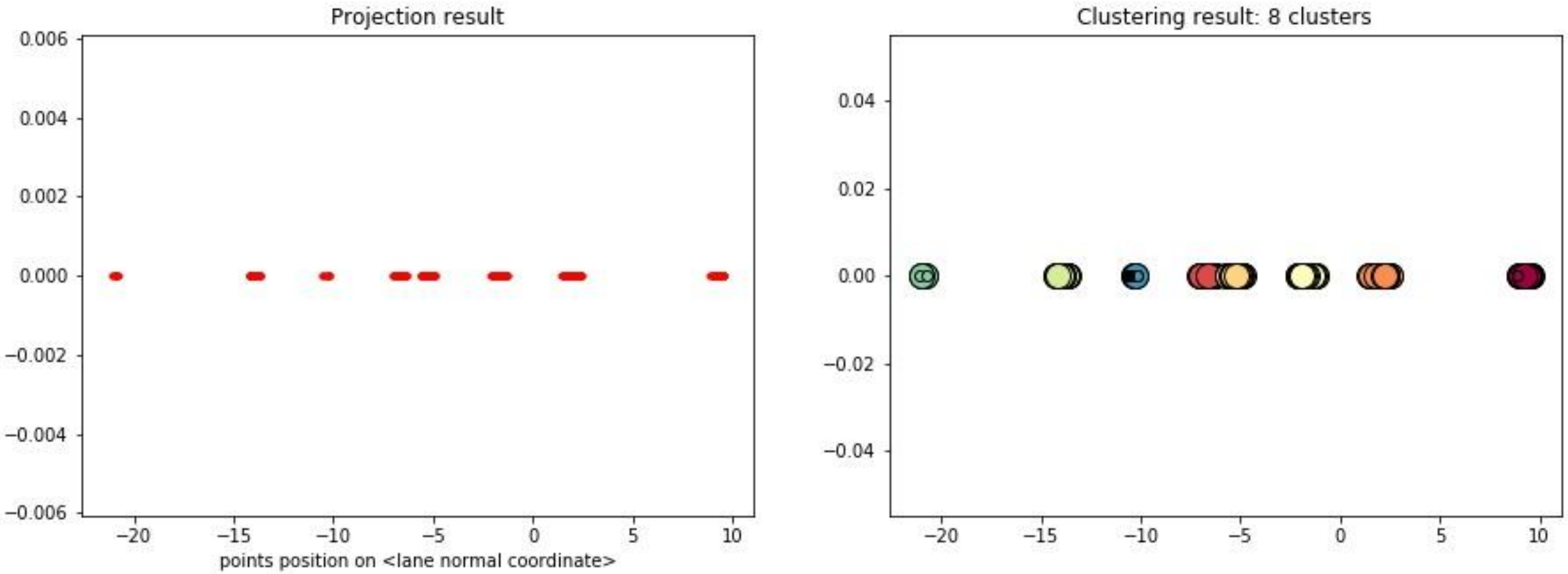
Code of DBSCAN clustering:

```
# settings
min_points_in_a_lane = 10
max_width_of_a_lane = 0.2 # (meters)

# do clustterring
cluster = Clusterer()
cluster.fit(projections,
   eps=max_width_of_a_lane,
   min_samples=min_points_in_a_lane)
```

Class definition: see **lib_clustering.py**

Detect lanes by clustering

Result



Result: After projection, the lanes points are easily separated by the DBSCAN clustering algorithm. Result: **8 lanes are detected**.

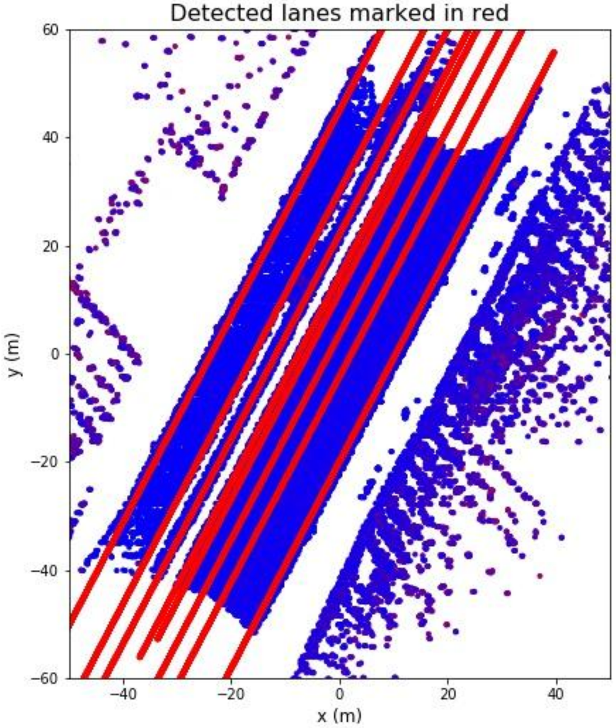
Final lane detection result

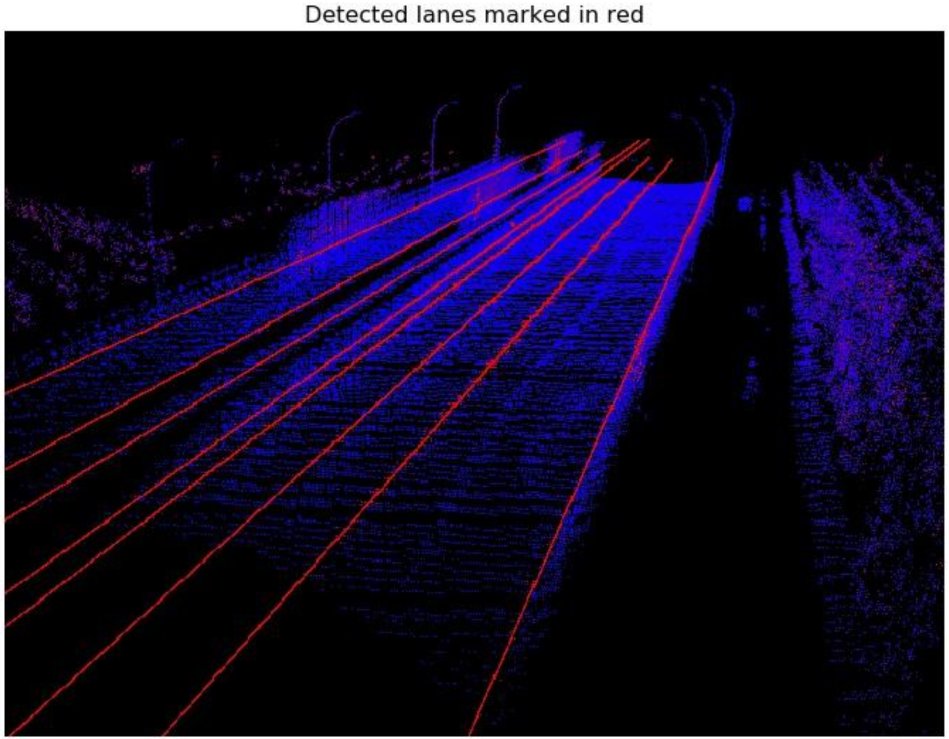
Finally, we again use PCA to fit a 3D line to each lane cluster:

```
1th line, 135 points.
Equation: (x-2.54)/(-0.46442) == (y+15.14)/(-0.88562) == (z-224.85)/(-0.00119)

2th line, 156 points.
Equation: (x-0.22)/(-0.46460) == (y-14.72)/(-0.88552) == (z-225.31)/(-0.00038)

3th line, 553 points.
Equation: (x+3.26)/(-0.46681) == (y+10.42)/(-0.88436) == (z-225.09)/(-0.00022)

4th line, 199 points.
Equation: (x-3.57)/(-0.46501) == (y-18.07)/(-0.88531) == (z-225.30)/(-0.00002)

5th line, 214 points.
Equation: (x+6.22)/(0.46647) == (y+8.22)/(0.88454) == (z-225.20)/(0.00012)

6th line, 49 points.
Equation: (x+15.62)/(0.46467) == (y-0.28)/(0.88548) == (z-225.11)/(0.00081)

7th line, 11 points.
Equation: (x+13.56)/(-0.46543) == (y-19.15)/(-0.88508) == (z-225.01)/(-0.00041)

8th line, 11 points.
Equation: (x+13.81)/(-0.46783) == (y+4.07)/(-0.88382) == (z-225.19)/(-0.00048)
```

Final lane detection result

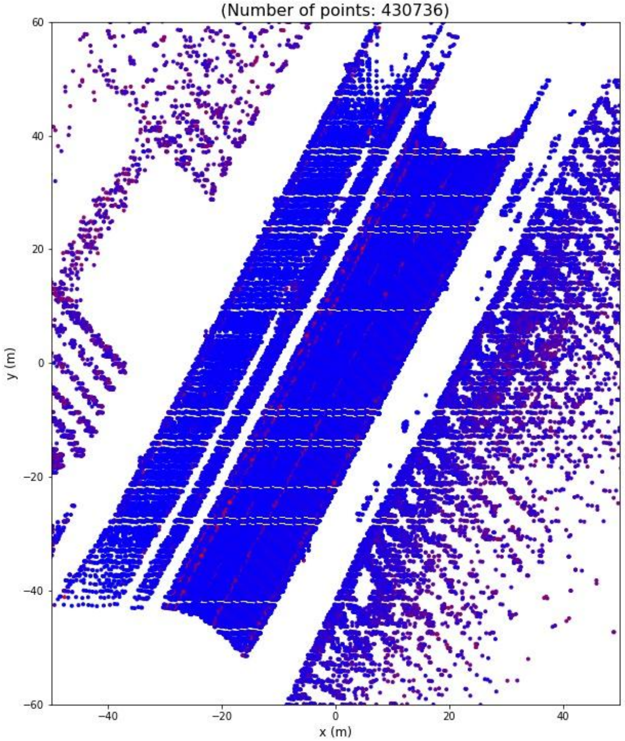
The detected 8 lanes are shown in red below:

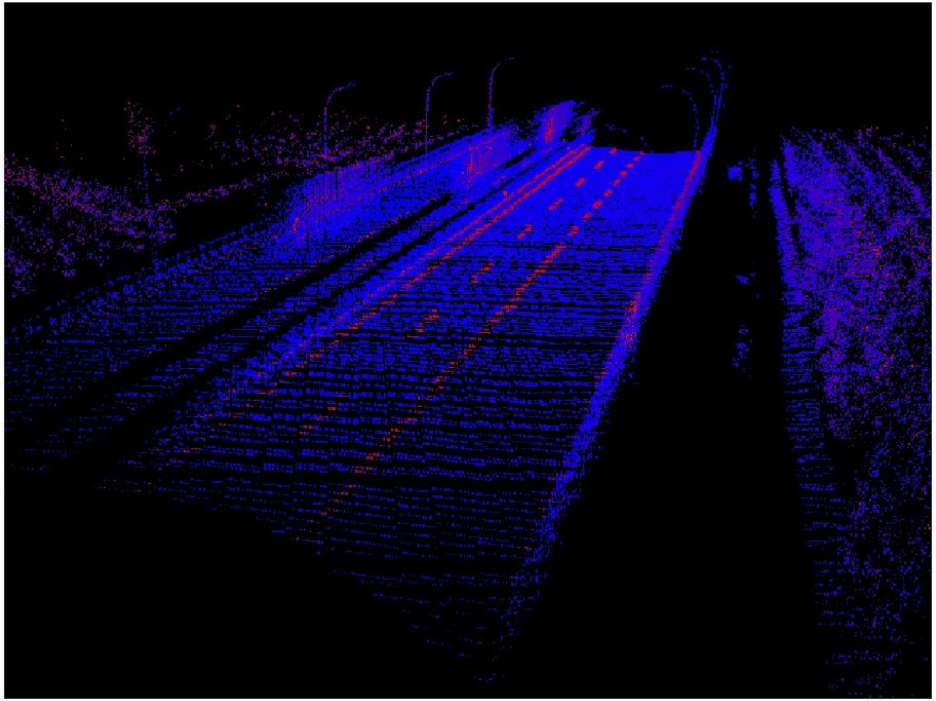




Raw data for comparison

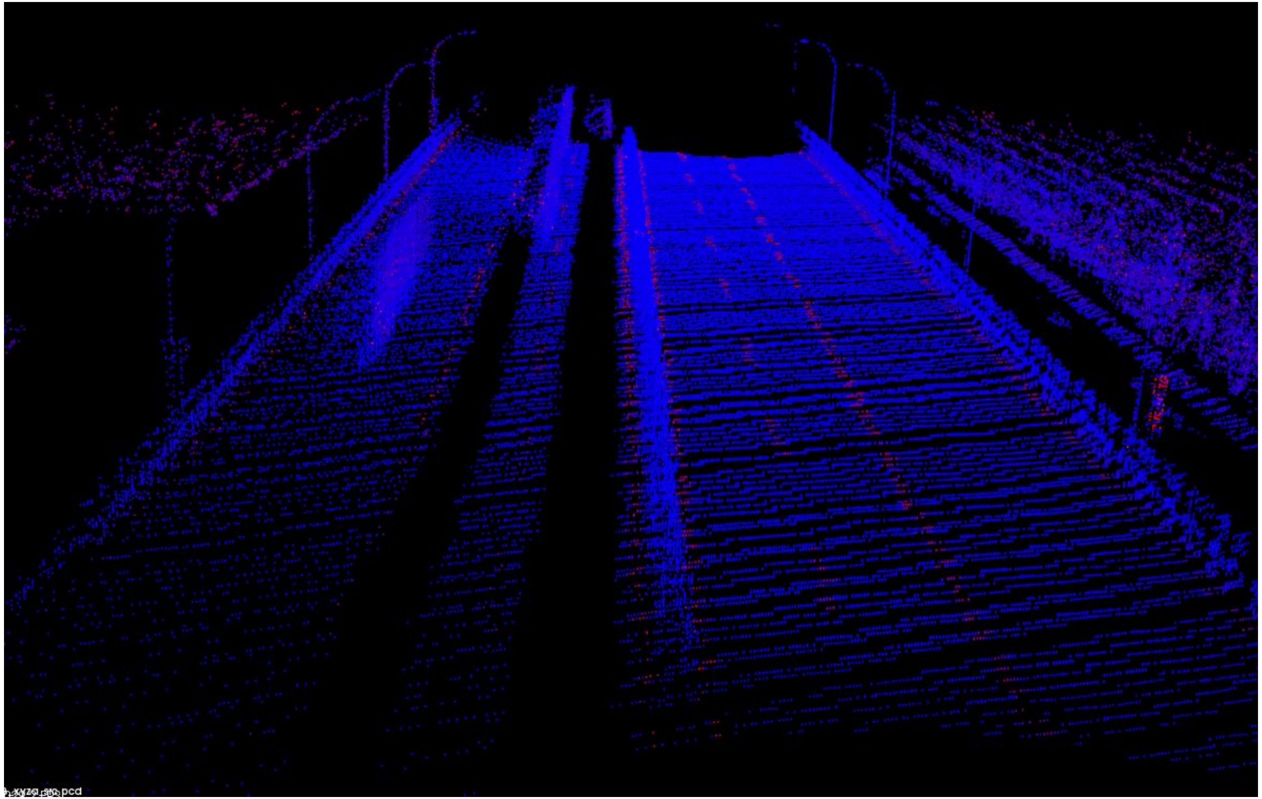
The raw point cloud is shown below for comparison:





Raw data for comparison

The raw point cloud is shown below for comparison:



Time cost

Procedure	Time cost (s)
Run through the whole main program “main.ipynb”	123.4
Plot a figure of 2D/3D point cloud with 430k points by using matplotlib	2.3
Remove non-planal regions by using kd-tree and querying 10 neighbors	59.4
Fit a plane to find the road surface by using RANSAC with 100 iteration	6.9
Other algorithms	<1

Conclusion

In this project, we successfully detected all eight lanes from the given point cloud data.

Our method is built upon the assumption/observation that:

1. Road surface is a thin planar region in point cloud, while other objects are not.
2. Lanes are in the road surface, and have larger reflexion than other road points. Besides, lanes are straight and parallel to each other.

Based on this observations, we designed the following algorithm to detect lanes:

1. Remove the non-planar regions, and detect the road surface from the remaining.
2. Threshold on reflexion intensity to get possible lane points.
3. Estimate the lane direction. Use clustering algorithm to separate each lane.
4. Finally, we estimate the line equation of each lane.

This framework performs well on the given point cloud data as it detects all the lanes in the point cloud, which proves the feasibility and accuracies of the proposed method.

Limitations and future work

1. Larger dataset

The proposed algorithm was tested only on a single point cloud, which might not work well for other real world scenarios. Larger dataset is needed in order to design more generalized and more robust method for lane detection.

2. Dealing with curved lanes

Lanes might be straight or curved. Instead of a linear model, future work should employ the quadratic function to fit the curve of each lane, and then do the feature reduction on the corresponding manifold.

3. Time cost

The current program is slow, and should be improved in the future work. (1) One way is to use *different downsampling rate* in the *detecting-lane phase* and *fitting-lane phase*. (2) Another way is to utilize the prior knowledge of where the lanes might be, in order to reduce the search region of the lanes. (3) Besides, it is promising to use the deep learning frameworks to process point cloud data, which might reduce the time cost and increase the accuracy of lane detection.

Thank you!