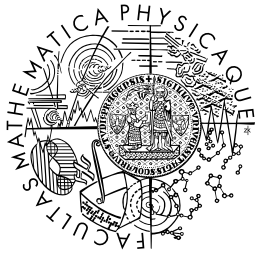

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Yana Hrynevich

Lane Detection Using LIDAR Data

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the bachelor thesis: Mgr. Martin Pilát, Ph.D.
Study programme: Computer Science

Prague 2024

I declare that I carried out this bachelor thesis on my own, and only with the cited sources, literature and other professional sources. I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

I dedicate this thesis to everyone who supported and assisted me during its creation. I would like to express my deep gratitude to my supervisor, Mgr. Martin Pilát, Ph.D., and my consultant, Tomáš Svoboda, for their guidance and invaluable advice. I also wish to acknowledge Valeo for providing the idea and data for the initial research. I am especially thankful to Yegor S. for motivating me and supporting during challenging times. Finally, I dedicate this work to my friends and parents for always being by my side.

Title: Lane Detection Using LIDAR Data

Author: Yana Hrynevich

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: Mgr. Martin Pilát, Ph.D., Department of Theoretical Computer Science and Mathematical Logic

Abstract: Lane detection plays a critical role in autonomous vehicle navigation. While traditional approaches utilize camera data, they often suffer from significant image distortion issues. In contrast, recent developments have introduced techniques based on LIDAR (Light Detection and Ranging) data, which are unaffected by these limitations. The primary goal of this study is to investigate and develop LIDAR-based lane detection techniques, with a particular focus on machine learning algorithms. It will propose a novel approach and compare different variations of this approach to evaluate their performance and potential advantages. Through this comparative analysis, the study seeks to contribute to the development of autonomous vehicle navigation by offering a more robust and accurate lane detection solution that can significantly reduce navigation errors and improve overall vehicle safety.

Keywords: LIDAR, Lane Detection, Autonomous Vehicle Navigation, Machine Learning

Contents

Introduction

The development of autonomous vehicles represents a significant technological advancement with the potential to transform transportation systems worldwide. A critical component of autonomous vehicle functionality is accurate and reliable lane detection, which ensures safe and efficient navigation. Recent advancements in sensor technology, particularly Light Detection and Ranging (LIDAR), have greatly improved lane detection capabilities. LIDAR sensors provide high-resolution, three-dimensional representations of the environment, making them ideal for detecting and classifying lanes.

Despite the potential of LIDAR technology, there are challenges associated with lane detection, particularly in the context of human labeling. Creating large labeled datasets for training supervised machine learning models is time-consuming and prone to errors. This motivates the need for an unsupervised, fast, and effective lane detection method that does not rely on extensive human-labeled data.

This thesis aims to investigate and develop LIDAR-based lane detection techniques, focusing on an unsupervised approach that facilitates three-dimensional segmentation of point clouds. The primary objective is to create a method where each point in the point cloud is labeled as lane or non-lane, with an additional classification of lane types. This will result in the creation of a comprehensive 3D map of the driving environment, which is crucial for autonomous navigation.

To achieve this, the study will combine several well-known machine learning techniques such as Principal Component Analysis (PCA), Random Sample Consensus (RANSAC), and Density-Based Spatial Clustering of Applications with Noise (DBSCAN) clustering. These techniques will be integrated into custom-developed algorithms designed to filter non-lane points from the point clouds, connect remaining lane parts into unified lane entities and perform semantic segmentation.

PCA and RANSAC will play a pivotal role in this custom algorithm. PCA will be used to find the principal directions of the data, making it easier to identify lane points. RANSAC will be utilized for robust fitting of models to the data, effectively distinguishing lane points from noise and outliers. These methods, combined with DBSCAN clustering, will allow for the effective grouping of lane points and the accurate classification of lane types. Additionally, PCA and RANSAC-based techniques will be compared to evaluate their performance and effectiveness in the context of lane detection, ensuring that the most reliable and efficient methods are utilized.

The main goal of this research is to create a new method for detecting lanes using LIDAR data that is accurate and reliable, without needing large amounts of labeled data. By combining well-known machine learning techniques into a single system, this study aims to improve autonomous vehicles navigation and to make autonomous driving safer and more efficient.

1 Explanation of LIDAR Data

This chapter provides an overview of LIDAR technology and its application in lane detection. We start by explaining the working principles of LIDAR, including laser pulse emission and time-of-flight distance measurement. Next, we discuss the creation and importance of LIDAR point clouds for spatial data representation. We then explore the use of LIDAR in lane detection, highlighting key methods and advancements that improve lane-keeping accuracy. Finally, we introduce the Agroverse 2 dataset, which offers extensive LIDAR data and annotations essential for training and evaluating lane detection models in autonomous driving research.

1.1 Working Principle of LIDAR

LIDAR (Light Detection and Ranging) is a remote sensing technology that uses laser light to measure distances and create detailed, high-resolution maps of the Earth's surface and other environments. The basic principle behind LIDAR involves emitting laser pulses towards a target and measuring the time it takes for the pulses to reflect back to the sensor. This time-of-flight measurement allows for precise calculation of distances, enabling the creation of accurate three-dimensional representations of objects and landscapes.

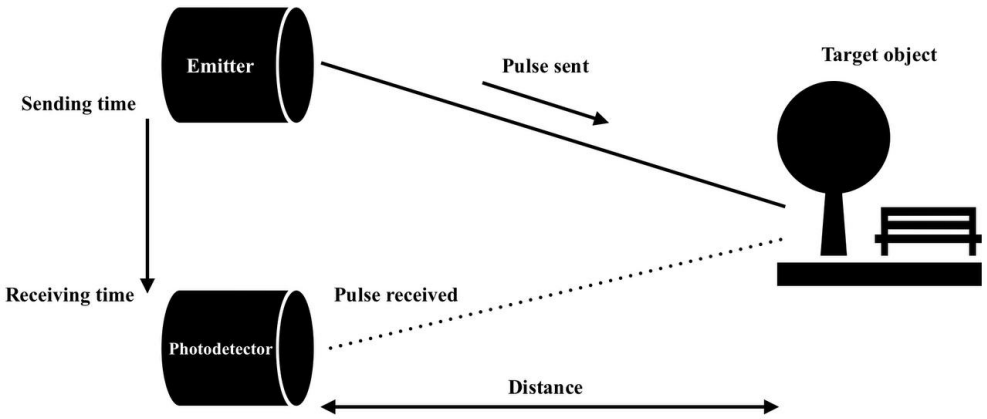

Figure 1.1 Working principle of LIDAR. [1]

LIDAR systems begin by emitting laser pulses towards a target area. The laser source, typically a solid-state or fiber laser, generates rapid bursts of light at specific wavelengths, often in the near-infrared spectrum. These pulses are directed through a scanner, which can rotate or oscillate to cover a wide area, ensuring comprehensive spatial coverage.

When the laser pulses hit an object, such as a building, tree, or the ground, they reflect back towards the LIDAR sensor. The photodetector in the sensor array captures these reflected signals. The crucial component here is the timing system, which precisely measures the time interval between the emission of each pulse and

its return. This time interval, known as the time-of-flight, is directly proportional to the distance between the sensor and the target.

The time-of-flight measurement is fundamental to LIDAR's ability to calculate distances. By knowing the speed of light and the time it takes for the laser pulse to travel to the target and back, the LIDAR system can compute the distance using the formula:

$$D = \frac{c \times t}{2}$$

where D is the distance to the object, c is the Speed of Light, and t is the time it takes for the laser pulse to travel from the source to the object and back.

The division by two accounts for the round-trip journey of the pulse. This process is repeated for millions of pulses per second, allowing the LIDAR system to capture a dense array of distance measurements.

Once the raw distance data is collected, sophisticated signal processing algorithms are employed to filter out noise and enhance the accuracy of the measurements. These algorithms handle issues such as multi-path reflections, where a single pulse might bounce off multiple surfaces before returning to the sensor, and atmospheric interferences that can affect the time-of-flight calculations.

1.2 LIDAR Point Cloud

The processed distance measurements are compiled into a point cloud, a collection of spatial data points representing the scanned environment. Each point in the cloud has coordinates (x, y, z) corresponding to a specific location in three-dimensional space. Additional attributes, such as intensity values, can also be recorded, providing information on the reflectivity of the target surfaces.

	x	y	z	intensity	laser_number	offset_ns
0	-1.291016	2.992188	-0.229370	24	31	3318000
1	-25.921875	25.171875	0.992188	5	14	3318000
2	-15.500000	18.937500	0.901855	34	16	3320303
3	-3.140625	4.593750	-0.163696	12	30	3320303
4	-4.445312	6.535156	-0.109802	14	29	3322607
...	...	...	...	...	...	...
98231	18.312500	-38.187500	3.279297	26	50	106985185
98232	23.109375	-34.437500	3.003906	20	49	106987490
98233	4.941406	-5.777344	-0.162720	12	32	106987490
98234	6.640625	-8.257812	-0.157593	6	33	106989794
98235	20.015625	-37.062500	2.550781	12	47	106989794

[98236 rows x 6 columns]

Figure 1.2 Format of Agroverse 2 (1.4) Point Cloud.

1.3 Applications of LIDAR Technology in Lane Detection

LIDAR technology helps in precise lane keeping by continuously scanning the road surface and detecting lane markings, even in challenging conditions like poor lighting or adverse weather. The high-resolution data allows the vehicle to stay centered in its lane and execute smooth lane changes when necessary. Recent advancements have introduced methods like the Scatter Hough algorithm [2], which improves the detection of lanes through a neighbor voting system, adapting to both straight and curved lane patterns even in noisy environments.

In addition to traditional lane keeping, LIDAR technology plays a crucial role in overall navigation by providing detailed, lane-level digital maps that guide vehicles along planned routes. This capability is crucial for navigating complex urban environments and intersections, roundabouts, and other challenging road features. For instance, a real-time lane detection system described by Jiyoung Jung and Sung-Ho Bae [3] uses an expectation-maximization method to update 3D lane parameters dynamically, enhancing the ability to generate accurate lane-level maps.

Furthermore, the LiLaDet framework [4] represents a breakthrough in LIDAR-based 3D lane detection by utilizing a dual-pathway processing approach, which merges bird's-eye view (BEV) and voxel-based spatial features to increase detection accuracy across diverse urban landscapes. Another innovative approach, the row-wise LIDAR Lane Detection Network [5], leverages a two-stage network that refines lane detection through correlation of lane features, demonstrating enhanced detection capabilities in scenarios with occlusions and complex merging lanes.

By integrating these LIDAR data with GPS and other sensor inputs, autonomous vehicles can achieve highly accurate localization and sophisticated path planning, ensuring safer and more efficient navigation across a variety of driving conditions. These advancements underscore the dynamic potential of LIDAR technology in transforming autonomous driving technologies.

1.4 Agroverse 2 Dataset

In this research, we will utilize the Agroverse 2 Sensor dataset [6], which is is a collection of open-source autonomous driving data and high-definition (HD) maps from six U.S. cities: Austin, Detroit, Miami, Pittsburgh, Palo Alto, and Washington, D.C. This release builds upon the initial launch of Argoverse, which was among the first data releases of its kind to include HD maps for machine learning and computer vision research. The dataset includes recorded logs of sensor data, or "scenarios," across different seasons, weather conditions, and times of day, making it a versatile resource for training and evaluating machine learning

models in the context of autonomous driving.

The data is collected using a fleet of identical Ford Fusion Hybrids, fully integrated with Argo AI self-driving technology. The dataset includes data from two LIDAR sensors, seven ring cameras, and two front-facing stereo cameras.

Each vehicle log is approximately 15 seconds in duration and 1 GB in size, including 150 LIDAR sweeps on average, and 300 images from each of the 9 cameras (2700 images per log).

LIDAR sweeps are collected at 10 Hz, along with 20 fps imagery from 7 ring cameras positioned to provide a fully panoramic field of view, and 20 fps imagery from 2 stereo cameras. In addition, camera intrinsics, extrinsics and 6-DOF ego-vehicle (the vehicle that contains LIDAR sensors) pose in a global coordinate system are provided. LIDAR returns are captured by two 32-beam LIDARs, spinning at 10 Hz in the same direction, but separated in orientation by 180°. The cameras trigger in-sync with both LIDARs, leading to a 20 Hz frame-rate. The nine global shutter cameras are synchronized to the LIDAR to have their exposure centered on the LIDAR sweeping through their fields of view. On average, LIDAR sensors produce a point cloud with 107,000 points at 10 Hz.

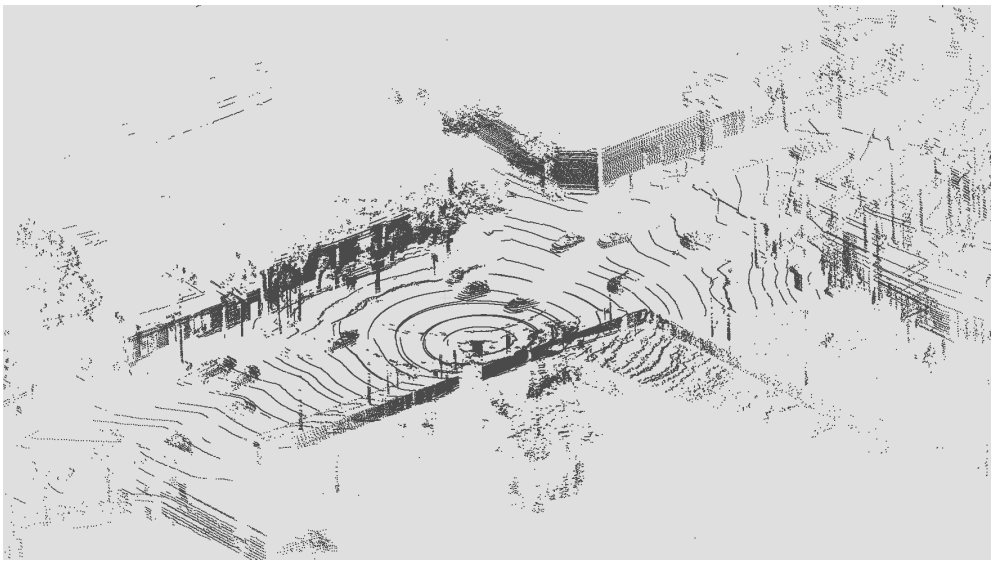

Figure 1.3 Agroverse 2 LIDAR sweep captured by one nanosecond timestamp.

All returns are aggregated from the two stacked 32-beam sensors into a single sweep. These sensors each have different, overlapping fields-of-view. Both LIDARs have their own reference frame. All LIDAR returns are provided in the egovehicle reference frame, not the individual LIDAR reference frame.

To support the Sensor datasets, Agroverse 2 maps include real-valued ground height at thirty centimeter resolution. With these map attributes, it is easy to filter out ground LIDAR returns (e.g. for object detection) or to keep only ground LIDAR returns (e.g. for building a bird's-eye view rendering).

Detailed annotations of lane markings are also provided, allowing for the supervised

training of lane detection models and the assessment of their performance against ground truth data.

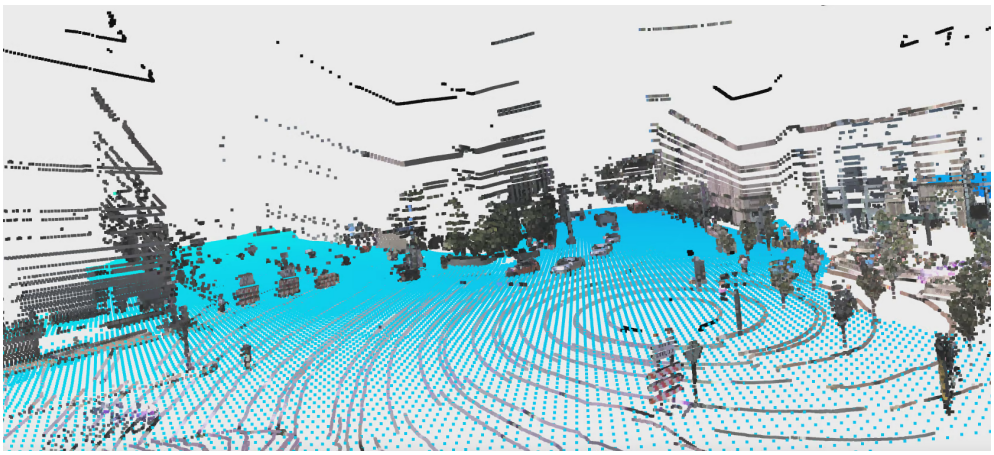

Figure 1.4 Ground height samples for an Argoverse 2 scenario. Ground height samples are visualized as blue points. LIDAR samples are shown with color projected from ring camera imagery. [6]

Tabular data (annotations, lidar sweeps, poses, calibration) are provided as Apache Feather Files with the file extension .feather. Examples are provided in the figures 1.2 and 2.1.

Dataset structure is shown in the figure 1.5.

```
av2
└── sensor
    ├── train
    │   └── LyIXwbWeHWPHYUZjD1JPdXcvvtYumCWG
    │       └── sensors
    │           ├── cameras
    │           └── lidar
    │               └── 15970913559644000.feather
    │                   .
    │                   .
    │                   .
    │       ├── calibration
    │       │   ├── egovehicle_SE3_sensor.feather
    │       │   └── intrinsics.feather
    │       ├── map
    │       │   ├── LyIXwbWeHWPHYUZjD1JPdXcvvtYumCWG__Summer___img_Sim2_city.json
    │       │   ├── LyIXwbWeHWPHYUZjD1JPdXcvvtYumCWG__Summer_ground_height_surface____PIT.npy
    │       │   └── log_map_archive_LyIXwbWeHWPHYUZjD1JPdXcvvtYumCWG__Summer____PIT_city_77257.json
    │       ├── city_SE3_egovehicle.feather
    │       └── annotations.feather
    ├── val
    └── test
```

Figure 1.5 Structure of Agroverse 2 Sensor dataset.

The following table provides the abbreviations for each Log ID, which corresponds to a dataset directory containing LIDAR sweeps.

PC	Log ID	Number of Timesteps
PC1	0f0cdd79-bc6c-35cd-9d99-7ae2fc7e165c	196
PC2	0c6e62d7-bdfa-3061-8d3d-03b13aa21f68	156
PC3	0fdbd56a-1ff7-3624-81f9-03cd68fd5616	157
PC4	11a84740-18a3-3798-91c5-21dc9c765350	157

Table 1.1 Information about Dataset Logs

2 Transformation and Processing of LIDAR Point Clouds

This chapter explores the transformation and processing of LIDAR point clouds to create detailed and accurate 3D maps essential for autonomous vehicle navigation. We begin by explaining the transformation of LIDAR data from the ego-vehicle's reference frame to a unified global coordinate system, which is crucial for consistent spatial mapping. Next, we cover the methodology for accumulating transformed point clouds to build comprehensive 3D models. The chapter further explores the use of intensity-based filtering to enhance lane detection capabilities, followed by advanced techniques such as octrees for spatial partitioning and voxel grid filtering for downsampling point clouds. Lastly, we address outlier removal using the Statistical Outlier Removal (SOR) [7] filter to improve data quality, developing a robust pipeline for processing LIDAR data for autonomous driving applications.

2.1 Transformation and Accumulation of Point Clouds

LIDAR sensors mounted on an autonomous vehicle, referred to as the ego-vehicle, capture the surrounding environment in the form of point clouds. These point clouds represent the spatial distribution of points reflecting from objects and surfaces within the sensor's field of view. In our study, each LIDAR sensor's outputs are initially provided in the ego-vehicle's reference frame. This frame is fixed relative to the vehicle, thus moving and rotating with it.

For the purpose of generating a comprehensive 3D map of the urban landscape, it is necessary to transform these point clouds from the ego-vehicle's reference frame into a unified global coordinate system, typically defined as the city coordinate system. This coordinate system remains fixed and provides a consistent frame of reference that is crucial for tasks such as navigation, mapping, and multi-vehicle coordination.

2.1.1 Methodology for Transformation

For each LIDAR sweep, the 6-degrees-of-freedom (6-DOF) pose of the egovehicle is provided in the global coordinate system of the city. This pose includes both a translation vector (t_x, t_y, t_z) representing the position of the egovehicle, and a quaternion (q_w, q_x, q_y, q_z) representing the orientation of the vehicle. The

quaternion provides a compact and robust method for encoding the vehicle's rotation, avoiding the pitfalls of gimbal lock associated with Euler angles.

qw	qx	qy	qz	tx_m	ty_m	tz_m
-0.740565	-0.005635	-0.006869	-0.671926	747.405602	1275.325609	-24.255610
-0.740385	-0.005626	-0.006911	-0.672124	747.411245	1275.385425	-24.255906
-0.740167	-0.005545	-0.006873	-0.672365	747.419676	1275.474686	-24.256406
-0.740167	-0.005545	-0.006873	-0.672365	747.419676	1275.474686	-24.256406
-0.739890	-0.005492	-0.006953	-0.672669	747.428448	1275.567576	-24.258680
...	...	...	...	...	...	...
-0.694376	-0.001914	-0.006371	-0.719582	740.163738	1467.061503	-24.546971
-0.694326	-0.001983	-0.006233	-0.719631	740.160489	1467.147020	-24.545918
-0.694346	-0.001896	-0.006104	-0.719613	740.158684	1467.192399	-24.546316
-0.694307	-0.001763	-0.005998	-0.719652	740.155543	1467.286735	-24.549918
-0.694287	-0.001728	-0.005945	-0.719672	740.153742	1467.331549	-24.550363

Figure 2.1 Egovehicle pose table for different timestamps.

The transformation of a point $P_{local} = (x, y, z)$ in the LIDAR's local coordinate system to the global coordinate system P_{global} involves two main steps: rotation followed by translation [8]. Mathematically, this can be expressed as:

$$P_{global} = R(q) \cdot P_{local} + T$$

where $R(q)$ is the rotation matrix derived from the quaternion $q = (q_w, q_x, q_y, q_z)$, and $T = (t_x, t_y, t_z)$ is the translation vector. The rotation matrix $R(q)$ is computed as follows:

$$R(q) = \begin{bmatrix} 1 - 2q_y^2 - 2q_z^2 & 2q_xq_y - 2q_zq_w & 2q_xq_z + 2q_yq_w \\ 2q_xq_y + 2q_zq_w & 1 - 2q_x^2 - 2q_z^2 & 2q_yq_z - 2q_xq_w \\ 2q_xq_z - 2q_yq_w & 2q_yq_z + 2q_xq_w & 1 - 2q_x^2 - 2q_y^2 \end{bmatrix}$$

This rotation aligns the point cloud with the orientation of the global frame, and the translation adjusts its position to reflect the vehicle's location within the global space.

2.1.2 Methodology for Accumulation

Once the point clouds are transformed into the global coordinate system, they can be accumulated to build a comprehensive map of the environment. Accumulating point clouds involves merging multiple point clouds into a single dataset where each point's position is relative to the same global reference frame. This process enhances the density and coverage of the spatial data, enabling more detailed and accurate representations of the environment.

The primary challenge in point cloud accumulation is managing the vast amount of data. Efficient data structures, such as octrees (section 2.3), are crucial for

reducing the computational load by organizing the points in a hierarchical manner that allows for quick access and manipulation.

Additionally, it is essential to address potential issues such as data redundancy. Techniques such as voxel grid filtering (section 2.5) can be applied to downsample the point cloud, reducing its size while maintaining the structural integrity of the data.

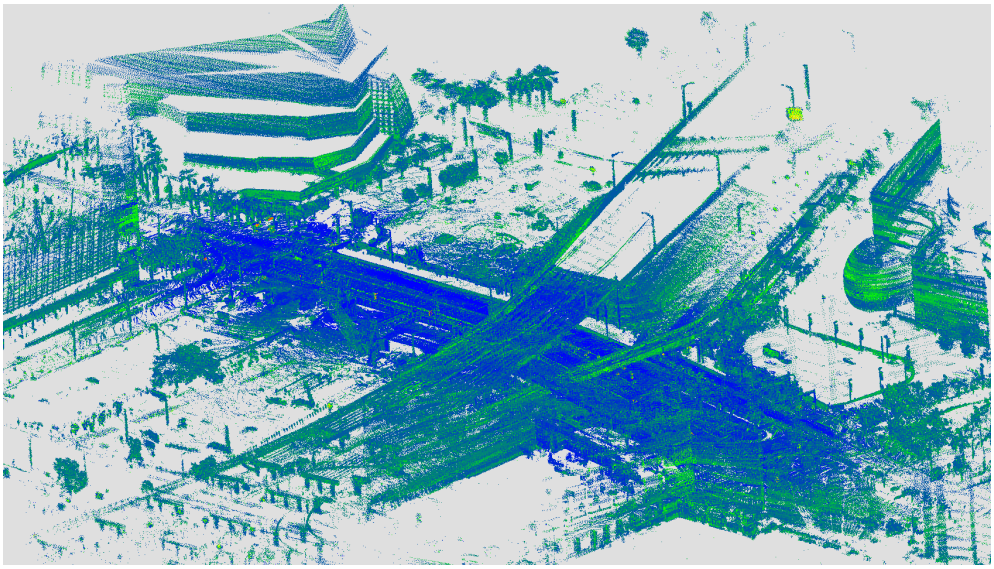

Figure 2.2 Accumulated PC1 before preprocessing (number of points: 18,220,744).

2.2 Intensity-based Filtering of Point Clouds

Following the transformation and accumulation of LIDAR point clouds into a unified global coordinate system, the next crucial step in enhancing lane detection capabilities involves filtering the point clouds based on intensity values. Intensity values in a LIDAR dataset represent the reflectivity of surfaces from which the LIDAR pulses bounce back. These values are crucial for distinguishing between different types of surfaces, such as lanes, which often have distinct reflective properties compared to other road elements.

2.2.1 Statistical Analysis of Intensity Values

Initial analysis of the intensity values from the accumulated point cloud (log ID: 0f0cdd79-bc6c-35cd-9d99-7ae2fc7e165c) reveals a distribution characterized by a mean of 30.42 and a standard deviation of 26.25. The distribution shown in the figure 2.3, skewed towards lower intensity values, indicates variability in surface reflectivity, which is typical in urban environments.

2.2.2 Threshold Determination for Intensity Filtering

To isolate potential lane markings, which generally exhibit higher reflectivity, a threshold of 60 for intensity values was selected. This threshold is set above the mean plus approximately one standard deviation, effectively focusing on the upper quartile of the intensity distribution. The goal is to minimize the inclusion of non-lane elements that typically have lower reflectivity and to enhance the detection of lanes that are crucial for autonomous navigation.

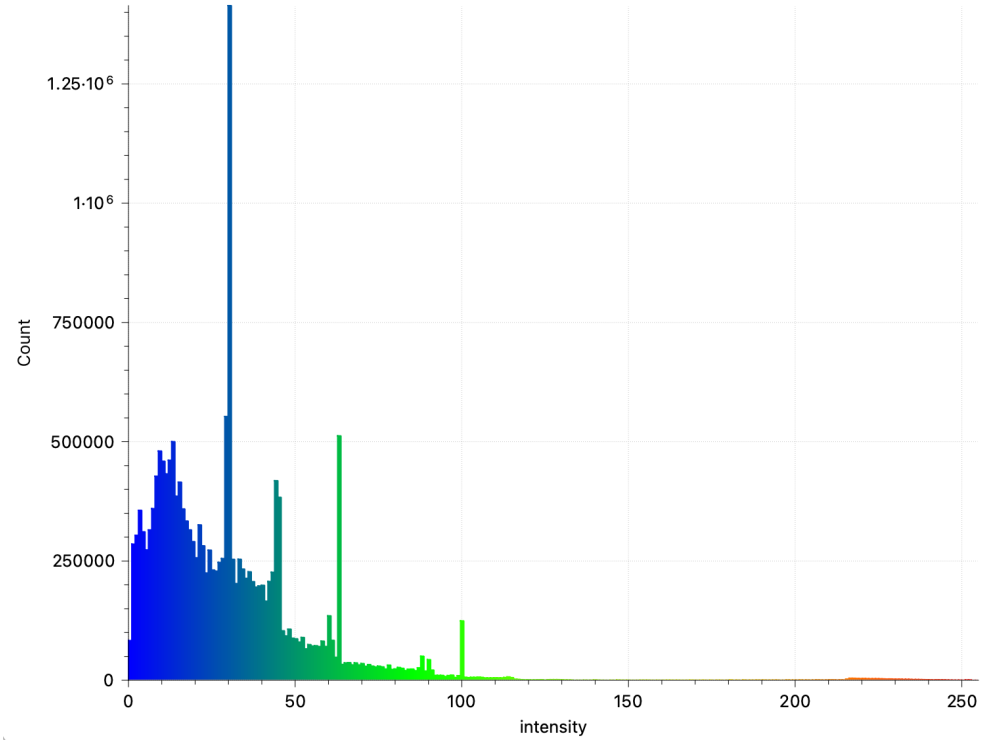

Figure 2.3 Histogram of intensity values with Gaussian fit.

2.2.3 Impact on Lane Detection

Selective filtering of point clouds based on intensity values greatly improves the detection of lane markings, which are crucial for guiding and controlling autonomous vehicles. By focusing on higher intensity points, this filtering process not only reduces computational effort but also enhances the accuracy and reliability of lane detection algorithms. This intensity-based filtering is a key improvement in our method for creating precise and detailed 3D maps.

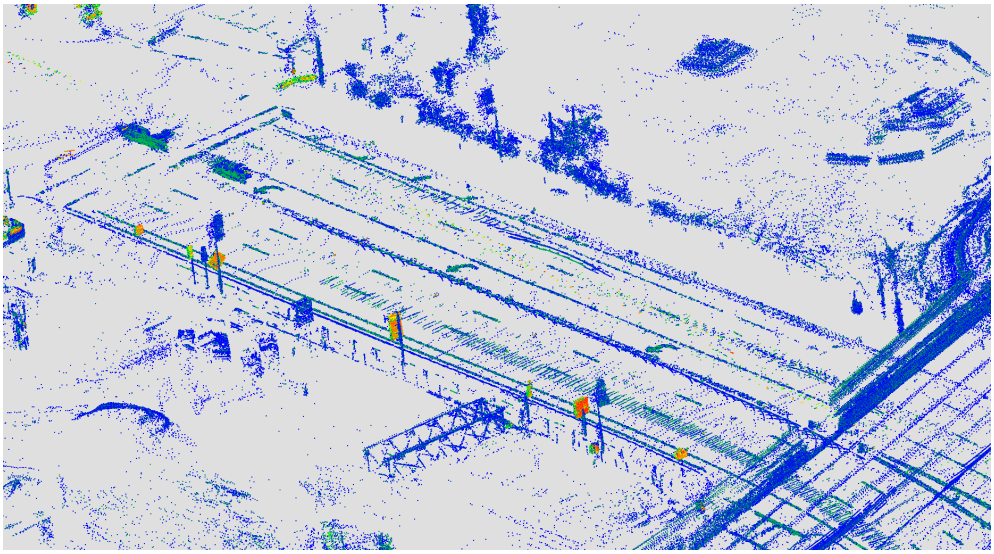

Figure 2.4 PC1 filtered by intensity value (number of points: 1,900,919).

2.3 Efficient Point Cloud Management Using Octrees

An octree [9] is a tree-based data structure that is widely used for the spatial partitioning of three-dimensional spaces. It subdivides space into eight octants at each node, hence the prefix 'oct'. This structure is particularly beneficial in managing large sets of three-dimensional data points, such as those found in LIDAR point clouds. By recursively breaking down the 3D space into smaller octants, octrees allow for efficient storage, manipulation, and retrieval of spatial data.

An octant is one of the eight divisions of a three-dimensional space relative to a central point. When a space (or a node in the context of octrees) is subdivided, each resulting section is an octant. These octants are the result of dividing the space along the three orthogonal axes (X, Y, and Z axes), each axis creating two halves, thus resulting in $2^3 = 8$ divisions or octants.

A voxel, short for "volume element," is the three-dimensional equivalent of a pixel (which represents two-dimensional elements). It is a value on a regular grid in three-dimensional space. Voxels represent a finite volume and typically contain additional information such as color, density, or scalar values representing some characteristic of the space they occupy.

2.3.1 Structure and Construction of Octrees

The structure of an octree is inherently hierarchical, with the division of three-dimensional space into voxels. Each node in the tree represents a voxel and can be further subdivided into eight smaller voxels, or children, each containing an

octant of the volume of its parent. This recursive subdivision continues until a pre-defined stopping criterion is met, typically based on voxel size, the number of points within a voxel, or tree depth.

Construction of an octree (Algorithm 1) begins with the encapsulation of the entire dataset within the root node's bounds. The space is then split into eight equal-sized octants. This division process recurses on each resulting child node until the termination condition is achieved. Each node stores pointers to its child nodes, and leaf nodes maintain a list of all points residing within their spatial boundaries.

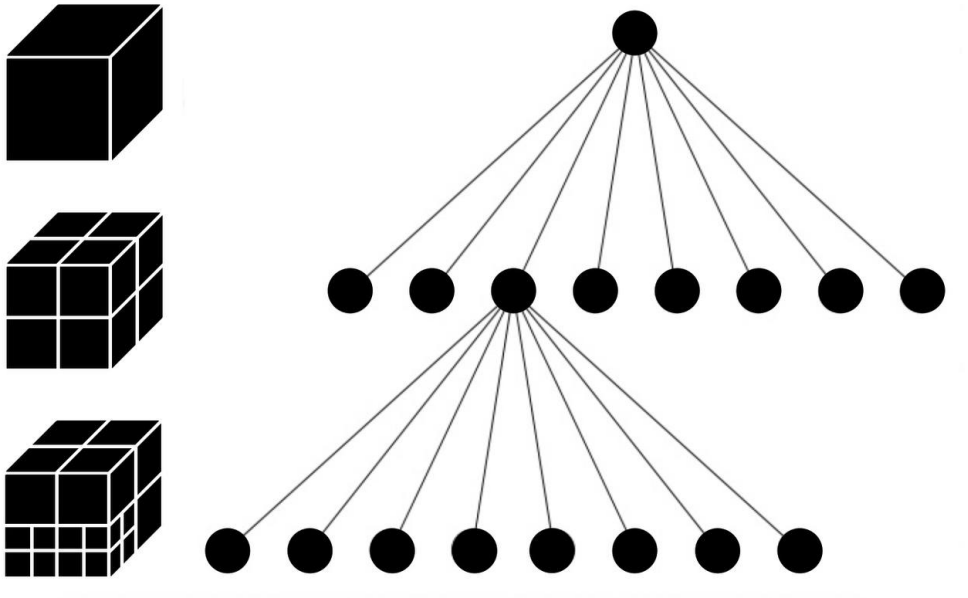

Figure 2.5 Left figure: Cube is recursively divided into eight octants. Right figure: Octree structure.

The mathematical representation of the octree subdivision can be defined as follows. Consider a voxel defined by its lower corner (x, y, z) and side length s. The center of the voxel (x_c, y_x, z_c) can be calculated as:

$$x_c = x + \frac{s}{2}, \quad y_c = y + \frac{s}{2}, \quad z_c = z + \frac{s}{2}$$

Using the center, the boundaries of the eight child voxels are computed. For instance, the boundaries for one of the octants would be $[x, x_c] \times [y, y_c] \times [z, z_c]$ and similarly for the others, adjusted according to their respective positions around the center.

In practice, the effectiveness of an octree depends on several factors, including the depth of the tree and the distribution of points within the space. Excessive depth can lead to over-partitioning and increased memory usage, while insufficient depth may not provide the desired efficiency gains. Therefore, balancing these aspects is crucial for optimizing octree performance in real-world applications.

Furthermore, dynamic updates to point clouds, such as those encountered in moving LIDAR systems, require efficient mechanisms for updating octrees without full reconstruction. Incremental update algorithms and adaptive partitioning strategies are areas of ongoing research aimed at addressing these challenges.

In conclusion, octrees provide a powerful tool for managing large-scale point cloud data, enabling efficient operations and optimizations that are critical for advanced 3D data processing applications like those found in autonomous vehicle navigation and urban modeling.

Algorithm 1 Construct Octree for 3D Point Cloud [9]

Require: Set of 3D points
Ensure: Octree structure where each node represents a voxel
  Initialize root node *root* with a voxel covering the entire dataset
  procedure ConstructOctree(*node*, *depth*)
    if number of points in *node* $\leq$ threshold or *depth* = maxDepth **then**
      Mark *node* as a leaf and store the points within
      return
    end if
    Determine the centroid of *node*'s voxel to split into 8 octants
    for each octant **do**
      Define voxel for the octant by subdividing *node*'s voxel
      Initialize a child node *child* for the octant
      Assign the voxel to *child*
      Filter and assign points within *child*'s voxel
      if *child* contains points **then**
        ConstructOctree(*child*, *depth* + 1)
      else
        Mark *child* as a leaf without points
      end if
    end for
  end procedure
  ConstructOctree(*root*, 0)

2.3.2 Comparison with k-d Trees

While octrees are highly effective for managing 3D point clouds, another prevalent data structure used for spatial partitioning is the k-d tree [10]. The k-d tree is a binary tree used for organizing points in k-dimensional spaces, typically with k=3 for spatial data applications such as those involving LIDAR data.

K-d trees partition space along data axes, alternating between them at each level of the tree. This results in a structure where each node splits the space into two, based on the median value of the points' coordinates in the chosen axis. This method of partitioning can be particularly efficient for balanced point clouds but may become less efficient as the distribution of points becomes skewed.

K-d trees excel in applications where nearest neighbor searches are predominant, particularly in datasets where data distributions are relatively uniform. They offer excellent logarithmic query times for these cases. However, for applications involving frequent updates, such as dynamic point clouds from LIDAR sweeps in autonomous vehicle contexts, octrees may be preferred. The hierarchical cubic subdivision of octrees can handle spatially uneven data distributions more gracefully, and they can adapt more robustly to changes in the dataset's spatial bounds.

2.4 Removing Outliers from Point Cloud Using Statistical Outlier Removal Filter

In the processing of LIDAR point clouds, outlier removal is a crucial step to enhance the quality and accuracy of the data. Outliers in a point cloud are typically points that deviate significantly from their surrounding data points and can arise due to various factors such as sensor noise, atmospheric conditions, or reflective surfaces. These outliers can affect subsequent processing steps such as feature extraction, segmentation, and object recognition. A common approach to addressing this issue is the Statistical Outlier Removal (SOR) filter, which uses statistical analysis to identify and remove outliers based on the distribution of point-to-point distances in the data.

2.4.1 The SOR Filtering Algorithm

The SOR [7] filter works by analyzing the distances between each point in the point cloud and its neighboring points. The algorithm assumes that the majority of points in the dataset form a surface while outliers are distant from this main group.

For the Statistical Outlier Removal process, each point p in the point cloud dataset undergoes a series of checks and calculations to determine if it should be classified as an outlier. The parameter k represents the number of nearest neighbors considered for each point p in the dataset. Initially, the k nearest neighbors of p are identified based on Euclidean distance, serving as the basis for further analysis. The mean distance μ and the standard deviation σ of these distances are then calculated. The k_factor, on the other hand, acts as a scaling factor that adjusts the sensitivity of the outlier detection mechanism. It is used to establish a threshold for determining whether a point is an outlier. A threshold for outlier detection is established at $\mu + k\_factor \cdot \sigma$, where k_factor acts as a scaling factor that modulates the sensitivity of the filter to outlier detection. Points whose mean distance to their neighbors exceeds this threshold are deemed outliers and are subsequently removed from the record. This method effectively reduces noise and improves the fidelity of the point cloud by discarding anomalous

points that deviate significantly from their local point neighborhood.

The steps of the algorithm are outlined as follows [7]:

Algorithm 2 Statistical Outlier Removal for Point Cloud Cleaning

Require: PointCloud P, Integer k, Float k_factor
Ensure: Cleaned PointCloud P_clean
  Initialize P_clean as an empty point cloud
  for each point p in P **do**
    $neighbors \leftarrow$ findKNearestNeighbors(p, k)
    $distances \leftarrow$ computeDistances(p, $neighbors$)
    $mean\_dist \leftarrow$ mean($distances$)
    $std\_dev \leftarrow$ standardDeviation($distances$)
    $threshold \leftarrow mean\_dist + k\_factor \times std\_dev$
    if $mean\_dist \leq threshold$ **then**
      Add p to P_clean
    end if
  end for
  return P_clean

2.4.2 Effectiveness Considerations

The effectiveness of the Statistical Outlier Removal (SOR) filter depends heavily on the choice of parameters k and k_factor. The parameter k, should be chosen based on the density of the point cloud; a larger k smooths over local variations and is less likely to remove points that are not true outliers but merely part of a sparse region. Conversely, k_factor, adjusts the strictness of the outlier definition. Higher values allow more inclusion, potentially keeping noise, whereas lower values make the filter stricter, potentially eliminating valuable information.

For instance, in one of our processing logs, the application of the SOR filter with these parameters reduced the point cloud from 1,900,919 (figure 2.4) data points to 1,776,476. This reduction underscores the filter's capacity to effectively manage data by removing statistical outliers, thus enhancing the quality and reliability of the subsequent processing steps.

Despite its effectiveness, the SOR filter has limitations. It may fail in scenarios where outliers form clusters dense enough to appear as valid regions, or in very sparse point clouds where even normal points may appear isolated. Furthermore, the filter can be computationally intensive, especially for large datasets, as it requires extensive calculations and neighbor searches for each point in the cloud.

The Statistical Outlier Removal filter is a robust method for cleaning LIDAR point cloud data, significantly improving the downstream processing and analysis steps. By effectively identifying and removing statistical outliers, the resulting models' quality and reliability are enhanced. Careful tuning of parameters and under-

standing of the dataset's characteristics are essential to optimize the performance and outcome of the SOR filter.

2.5 Voxel Grid Filtering for Point Cloud Downsampling

Voxel grid filtering is a common technique used for downsampling large point clouds to reduce their density while preserving the overall geometric structure of the data. This method involves overlaying a grid of voxels over the point cloud and then approximating the set of points within each voxel with their centroid or some other representative point. Typically, this representative point is calculated as the centroid of all points within the voxel, although other methods such as the average or median can also be used depending on the desired accuracy and computational constraints.

The voxel grid filtering process can be mathematically outlined as follows:

1. Define the dimensions of each voxel as (v_x, v_y, v_z), which determines the level of downsampling.
2. Compute the voxel index for each point in the original cloud using:

$$i = \left\lfloor \frac{x}{v_x} \right\rfloor, \quad j = \left\lfloor \frac{y}{v_y} \right\rfloor, \quad k = \left\lfloor \frac{z}{v_z} \right\rfloor$$

   where (x, y, z) are the Cartesian coordinates of a point in the cloud, and (i, j, k) represents the indices of the voxel in which the point is located.
3. For each voxel, aggregate the points it contains and compute the representative point, generally using the centroid formula:

$$\text{Centroid} = \left(\frac{\sum x}{n}, \frac{\sum y}{n}, \frac{\sum z}{n} \right)$$

   where n is the number of points in the voxel, and $\sum x, \sum y, \sum z$ are the sums of the respective coordinate values of the points within the voxel.

This approach is particularly useful in reducing the computational complexity of subsequent processing tasks.

2.5.1 Integration with Octree Structure

The integration of voxel grid filtering with the previously discussed octree structure (2.3) provides an efficient framework for managing and processing downscaled point

clouds. Octrees inherently divide the space into hierarchical cubic volumes, which can be leveraged to implement voxel grid filtering effectively. The adaptability of octree resolutions aligns well with the variable resolutions required in voxel grid filtering, facilitating a more dynamic and resource-efficient approach to point cloud management.

2.5.2 Voxel Grid Filtering Algorithm

The voxel grid filtering algorithm can be directly applied using the octree generated from the original point cloud. Each leaf node of the octree, which already contains a subset of the points grouped by spatial proximity, can be treated as a voxel. The filtering process then involves computing the centroid of the points within each voxel and replacing these points with the centroid.

Algorithm 3 Voxel Grid Filtering for Point Cloud Downsampling

Require: Octree root, empty PointCloud filteredCloud
Ensure: Downsampled PointCloud
  if root is a leaf node **then**
    Calculate the centroid of all points in this voxel
    Add the centroid to filteredCloud
    return
  end if
  For each child of root:
  Recursively call VoucherGridFilter with child, filteredCloud
  return filteredCloud

2.5.3 Effectiveness Considerations

Implementing voxel grid filtering involves choosing appropriate parameters that balance the level of downsampling and the fidelity of the resultant point cloud. The size of the voxels is a critical parameter that directly influences the number of points in the filtered point cloud; smaller voxels result in less downsampling and finer detail, while larger voxels produce more significant downsampling but may lead to loss of important geometric features.

By reducing the data volume, the filtering process enables faster decision-making and efficient data handling, so voxel grid filtering serves as an essential pre-processing step of our approach, efficiently managing the vast quantities of data generated by LIDAR sensors, which can capture millions of data points per second. For instance, in one of our processing logs (ID: 0f0cdd79-bc6c-35cd-9d99-7ae2fc7e165c), the voxel grid filtering was applied to a point cloud initially containing 1,776,476 data points. After filtering the number of points was effectively reduced to 507,235. By downsampling the point cloud, computational and memory requirements are substantially decreased.

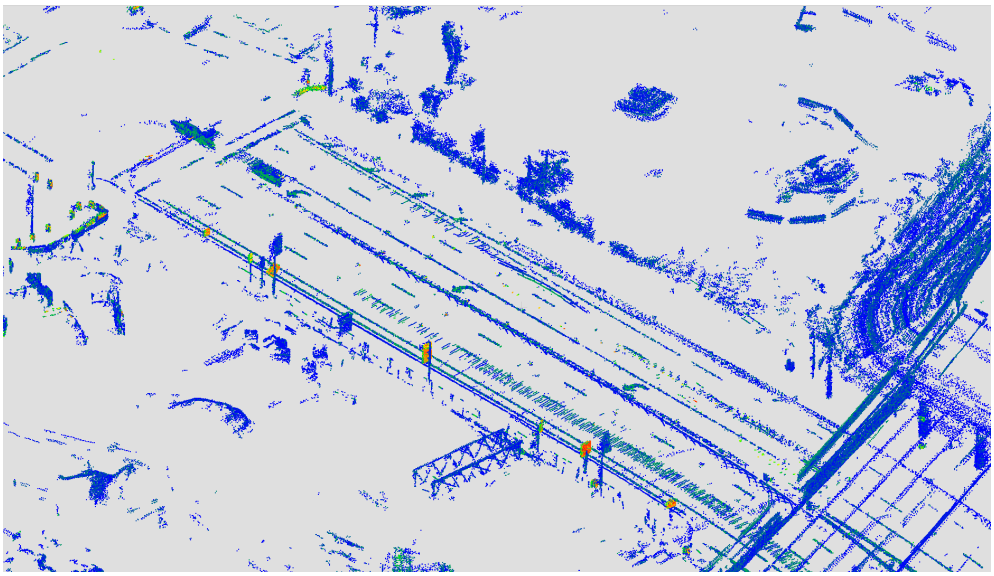

Figure 2.6 PC1 after SOR filtering and Voxel Grid Downsampling (number of points: 507,235).

3 Methodology for Lane Detection

In this chapter, we explore several unsupervised learning techniques that are pivotal for detecting lanes in LIDAR data. The focus is on three main methodologies to interpret the complex spatial data provided by LIDAR sensors: DBSCAN clustering, RANSAC fitting, and Principal Component Analysis (PCA). These methods are combined with custom-developed algorithms for lane connecting and lane classification to create a novel approach.

DBSCAN clustering is particularly advantageous for its ability to form clusters based on the density of points. By grouping points that are closely packed, DBSCAN helps highlight areas that represent lanes while distinguish them from lower-density regions which are typically background noise or non-lane objects.

RANSAC, or Random Sample Consensus, is another robust technique used in this context to identify and model lanes through iterative approximation. This method fits linear models to subsets of the data, estimating the parameters of a lane while classifying outliers that do not fit the model. RANSAC is particularly useful for its robustness against outliers, making it suitable for noisy data environments where other methods might fail.

Principal Component Analysis (PCA) is utilized to reduce the dimensionality of the data, simplifying the complex data into principal components that can represent the main direction or orientation of lanes. PCA helps in distinguishing lanes from other features by highlighting their linear alignment in the direction of roadways. This is particularly useful in intersections or areas where lane orientations significantly vary.

These techniques are not just discussed in isolation but are considered in how they integrate within the broader context of autonomous driving systems. We examine their practical implementations, challenges faced in real-world scenarios, and how they complement each other to enhance the reliability and accuracy of lane detection. This comprehensive approach helps in understanding the strengths and limitations of each technique, guiding future applications and improvements in autonomous navigation technologies.

3.1 DBSCAN Clustering

After the preprocessing steps that include intensity filtering, noise reduction, and downsampling, the next essential phase in lane detection using LIDAR data is the segmentation of the point cloud into meaningful clusters. Density-Based Spatial Clustering of Applications with Noise (DBSCAN) [11] is a robust, unsupervised clustering algorithm particularly suited for this task due to its ability to identify clusters of varying shapes and sizes based on density. This section explores the application of DBSCAN in the context of lane detection in LIDAR data.

3.1.1 Overview of DBSCAN

DBSCAN is a clustering algorithm that groups together closely packed points, marking as outliers the points that lie alone in low-density regions. This characteristic makes it highly effective for use in LIDAR data where lanes and other road markers form dense linear structures. The fundamental parameters of DBSCAN are:

- ϵ (*eps*): The maximum distance between two samples for one to be considered as in the neighborhood of the other.
- $MinPts$: The number of points required to form a dense region, which implies a minimum cluster size.

The DBSCAN algorithm (4) can be described by the following steps:

1. For each point in the dataset, if it has not yet been visited, retrieve its ϵ-neighborhood.
2. If the ϵ-neighborhood contains fewer than $MinPts$, mark the point as noise. Otherwise, create a new cluster and expand it by iteratively adding all density-reachable points from the ϵ-neighborhood. This means that for each new point added to the cluster, the process is repeated to find and include its ϵ-neighborhood points until no more points can be added.
3. Continue the process until all points are classified as either part of a cluster or as noise.

Mathematically, the neighborhood of a point p, denoted as $N_\epsilon(p)$, can be defined as:

$$N_\epsilon(p) = \{q \in D \mid \text{dist}(p, q) \leq \epsilon\}$$

where D is the dataset and $\text{dist}(p, q)$ is the distance between points p and q.

Algorithm 4 DBSCAN Clustering [11]

Require: Point set D, distance threshold ϵ, minimum points $MinPts$
Ensure: Clustered PointCloud

```
Initialize C = 0                                   ▷ Cluster counter set to 0
Initialize all points as unvisited
for each point p ∈ D do
    if p is visited then
        continue to next point
    end if
    Mark p as visited
    N = RetrieveNeighbors(p, ε)                    ▷ Find neighbors within ε radius
    if size of N < MinPts then
        Mark p as noise
    else
        C = C + 1                                  ▷ Start a new cluster
        ExpandCluster(p, N, C, ε, MinPts)
    end if
end for
procedure ExpandCluster(p, N, C, ε, MinPts)
    Add p to cluster C
    for each point q ∈ N do
        if q is not visited then
            Mark q as visited
            N' = RetrieveNeighbors(q, ε)
            if size of N' ≥ MinPts then
                N = N ∪ N'                         ▷ Add new neighbors to N
            end if
        end if
        if q is not yet member of any cluster then
            Add q to cluster C
        end if
    end for
end procedure
return All clusters
```

3.1.2 Application to Lane Detection

DBSCAN's ability to identify clusters based on local point densities and its insensitivity to the shape of the data distribution makes it particularly beneficial for lane detection in LIDAR data, where lanes can curve and vary in width. Furthermore, DBSCAN's trait of treating low-density groups as noise helps in effectively distinguishing between actual lane data and background noise such as pedestrians, bicycles, or other non-lane objects.

Implementing DBSCAN for lane detection in LiDAR data requires careful consideration of various factors to optimize the clustering process. The choice of parameters such as ϵ (the neighborhood distance threshold) and $MinPts$ (the minimum number of points to form a dense region) is critical. These parameters have an impact on the performance and effectiveness of the clustering, particularly in the context of accurately identifying lanes.

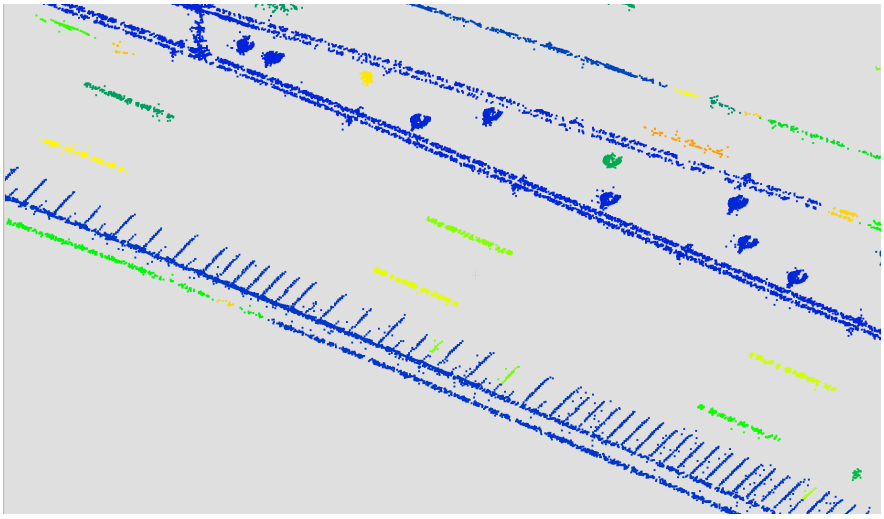

Figure 3.1 Optimal clustering result showing distinct lane markings ($\epsilon = 0.3$, $MinPts = 10$), where each color represents a distinct cluster.

3.1.3 Effectiveness Considerations

A smaller ϵ tended to break a single lane into multiple clusters (figure 3.3), especially in curved lane segments. Conversely, a larger ϵ sometimes resulted in multiple adjacent lanes being detected as a single cluster (figure 3.2), particularly in areas where lanes are close to each other.

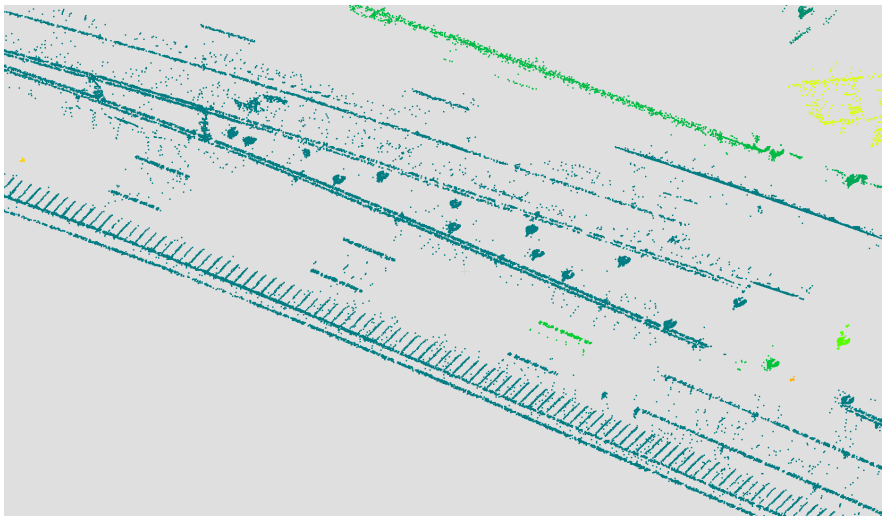

Figure 3.2 Under-clustering example with a larger ϵ (1.0) causing multiple lanes to merge (each color represents a distinct cluster).

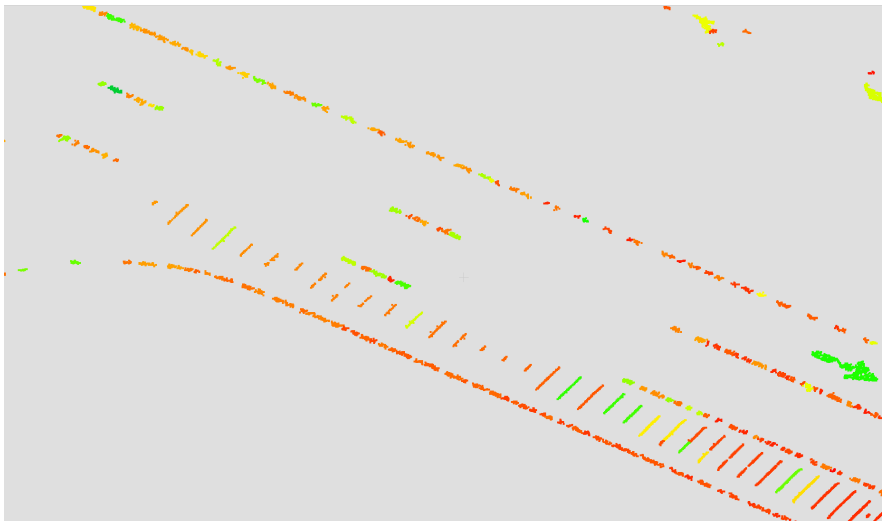

Figure 3.3 Over-clustering example where a smaller ϵ (0.1) breaks a single lane into multiple clusters (each color represents a distinct cluster).

3.2 The Random Sample Consensus (RANSAC) algorithm

The Random Sample Consensus (RANSAC) [12] algorithm is a robust statistical method used to estimate parameters of a mathematical model from a dataset that contains outliers. It was introduced by Fischler and Bolles in 1981 as a solution to the problem of fitting a model to experimental data contaminated with gross errors, which are typical in data captured by sensors like LIDAR. RANSAC is especially useful in the context of lane detection in LIDAR data due to its ability to effectively distinguish between inliers (data points that fit the model) and outliers (data points that do not fit the model).

3.2.1 Overview of RANSAC

RANSAC operates by iteratively selecting a random subset of the original data. These subsets are used to estimate the model parameters, which in the context of this thesis refers to either a line or a plane. The algorithm consists of the following steps:

Algorithm 5 RANSAC Algorithm

Require: Data points D, number of iterations N, threshold t, number of required inliers d
Ensure: Best fitting model parameters θ^*
  Initialize best inliers $I^* = \emptyset$
  Initialize best model $\theta^* = \text{null}$
  for $i = 1$ to N **do**
    Randomly select subset $S \subseteq D$ ▷ Minimal subset required for model
    Estimate model $\theta = \text{FitModel}(S)$
    Determine inliers $I = \{x \in D \mid \text{Distance}(x, \theta) < t\}$
    if $|I| > |I^*|$ **then**
      $I^* = I$
      $\theta^* = \theta$
      if $|I^*| \geq d$ **then**
        break
      end if
    end if
  end for
  return θ^*

The required number of iterations, N, can be calculated based on the probability that at least one random sample is free of outliers. The formula to determine N ensuring that the probability of choosing at least one good subset is at least p, given the proportion of outliers e, is given by:

$$N = \frac{\log(1-p)}{\log(1-(1-e)^s)}$$

where s is the size of the sample needed to fit the model. This formula ensures that RANSAC is capable of providing a reliable model estimate with a high confidence level.

3.2.2 Application to Lane Detection

Fitting Planes with RANSAC

In the processing of LIDAR data for lane detection, the first crucial step often involves fitting planes to the data, which can help in identifying the road surface. Using RANSAC for plane fitting is particularly advantageous because it can effectively ignore outliers such as vehicles, pedestrians, or debris, which do not belong to the road surface.

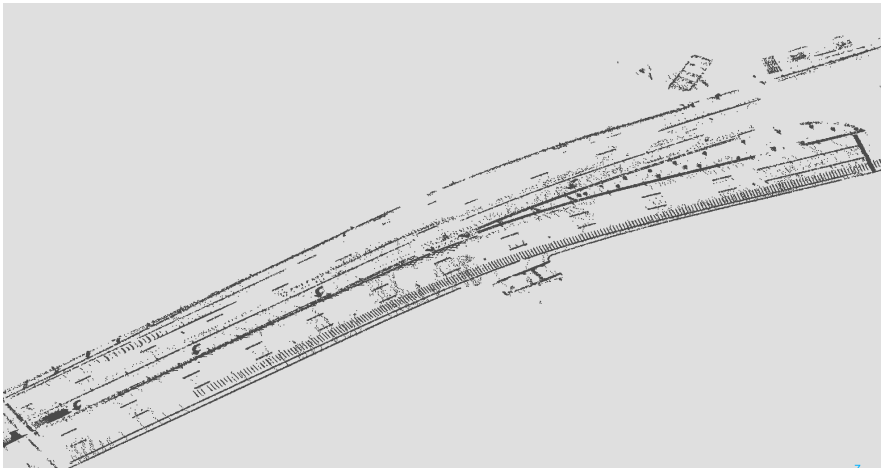

Figure 3.4 Projection of extracted ground plane (PC1).

The RANSAC algorithm adapted for plane fitting in LIDAR data typically involves:

- **Selection**: Randomly selecting a subset of LIDAR points that could potentially form a part of the road surface.
- **Model Estimation**: Fitting a plane model to these points. The mathematical model for a plane in a three-dimensional space can be represented as:

$$ax + by + cz + d = 0$$

- **Consensus**: Determining how many points from the entire dataset fit this plane model within a predefined error threshold, which classifies them as inliers.

After fitting the plane, the inlier points are projected onto a new plane where the z-axis is zero. This projection simplifies subsequent processing steps by reducing the dimensionality of the data and aligning it with the road surface, making it easier to detect lane markings and other relevant features.

This plane fitting process helps in establishing a baseline road level, from which deviations can be identified as potential lane markings or obstacles.

Fitting Lines for Lane Markings

After establishing the road surface, the next step is to detect lane markings. This involves an additional clustering step using DBSCAN (Density-Based Spatial Clustering of Applications with Noise) to group the LIDAR points that are likely part of lane markings based on spatial proximity and reflectivity values. DBSCAN helps in identifying clusters of points that represent different lane markings without the need for specifying the number of clusters in advance. Following the clustering process on the extracted plane, RANSAC is then applied to each detected cluster to fit line models to the features that correspond to lane markings. Line fitting with RANSAC in the context of LIDAR data for lane detection involves:

- **Selection**: Selecting each cluster previously identified by DBSCAN that is likely to contain parts of lane markings.
- **Model Estimation**: Fit a line model to the points within each cluster. The line model in a two-dimensional plane can be represented as:

  $$y = mx + b$$

  where m is the slope, and b is the intercept.
- **Consensus**: Measure how many points within the cluster agree with this line model within a certain tolerance, classifying them as inliers to the line model. The validity of this line is determined based on the number of inliers.

This method ensures accurate detection of each lane marking by fitting a line model to each cluster and selecting the line based on the number of inliers that meet a specified threshold. This approach allows for precise detection of lane boundaries. In multi-lane scenarios, the algorithm can be adapted by adjusting detection parameters to identify lines with different orientations, corresponding to various lane directions. This enables the accurate detection and classification of multiple lanes within a single framework.

These two stages, clustering with DBSCAN followed by line fitting with RANSAC, effectively segment the complex LIDAR data into meaningful lane marking models, crucial for autonomous vehicle navigation.

3.2.3 Effectiveness Considerations

RANSAC is highly regarded for its robustness against outliers, which are prevalent in real-world scenarios, making it particularly useful for applications like lane detection in LIDAR data where accuracy is critical despite noise and other data corruptions.

However, the algorithm's performance is heavily dependent on the correct setting of its parameters, such as the inlier threshold and the number of iterations. These parameters must be adjusted based on the specific characteristics of the dataset to avoid poor performance. Moreover, RANSAC can be computationally expensive, particularly with large datasets, as the process of random sampling may require substantial computational resources. These aspects highlight the need for careful implementation and optimization of RANSAC.

3.3 Principal Component Analysis (PCA)

Principal Component Analysis (PCA) is a statistical technique used to simplify complex datasets by transforming them into a set of orthogonal components, called principal components. These components capture the maximum variance in the data, with each subsequent component capturing progressively less variance. PCA is widely used for dimensionality reduction, noise reduction, and feature extraction in various fields.

3.3.1 Overview of PCA

Mathematically, PCA involves the following steps [13]:

- **Standardization**: Standardize the dataset to have zero mean and unit variance for each feature. This step is crucial to ensure that PCA does not become biased towards features with larger scales.

$$\mathbf{X}_{\text{stand}} = \frac{\mathbf{X} - \mu}{\sigma}$$

  where $\mathbf{X}$ is the original data matrix, μ is the mean vector, and σ is the standard deviation vector.

- **Covariance Matrix Computation**: Compute the covariance matrix of the standardized dataset.

$$\mathbf{C} = \frac{1}{n-1}\mathbf{X}_{\text{stand}}^T\mathbf{X}_{\text{stand}}$$

  where n is the number of observations.

- **Eigenvalue Decomposition**: Perform eigenvalue decomposition on the covariance matrix to obtain the eigenvalues and eigenvectors.

$$\mathbf{CV} = \mathbf{V}\boldsymbol{\Lambda}$$

where $\mathbf{V}$ is the matrix of eigenvectors and $\boldsymbol{\Lambda}$ is the diagonal matrix of eigenvalues.

- **Principal Components**: The principal components are the eigenvectors corresponding to the largest eigenvalues. These eigenvectors form a new basis set for the data, ordered by the amount of variance they explain.

$$\mathbf{Y} = \mathbf{X}_{\text{stand}}\mathbf{V}$$

where $\mathbf{Y}$ is the transformed dataset in the new principal component space.

3.3.2 Application to Lane Detection

In the context of lane detection using LIDAR data, PCA can be applied to identify linear features in point clouds. By applying PCA to clusters of LIDAR points, we can determine the direction of potential lane markings. Specifically, the first principal component (PC1) should capture a significant percentage of the variance, indicating the primary direction of the points in the cluster. The second principal component (PC2) should capture a relatively smaller percentage of the variance, confirming that the points are aligned along a line.

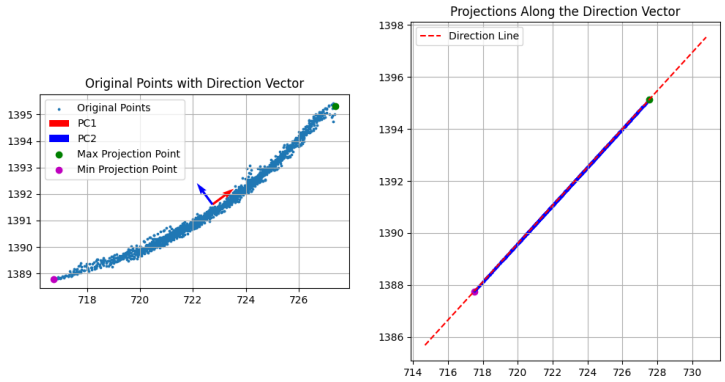

Figure 3.5 The result of PCA analysis for one cluster, blue points represent potential lane marking.

The steps for using PCA in lane detection are as follows:

- **Clustering**: Segment the LIDAR data into clusters using an algorithm like DBSCAN.
- **PCA Fitting**: Apply PCA to each cluster to obtain the principal components.

- **Variance Explained**: Analyze the percentage of variance explained by the first principal component (PC1) and ensure that the second principal component (PC2) captures significantly less variance. A higher variance along PC1 and a lower variance along PC2 indicate that the cluster likely represents a linear feature, such as a lane marking.
- **Direction Vector**: The first principal component (PC1) provides the direction vector of the line.
- **Projections**: Project the points along the direction vector to confirm the alignment and improve the detection accuracy.

3.3.3 Effectiveness Considerations

Using Principal Component Analysis (PCA) for lane detection in LIDAR data effectively identifies linear features and enhances accuracy by focusing on the main direction of point clusters. Combined with DBSCAN clustering and RANSAC fitting, PCA reduces noise and improves reliability. However, this method can be computationally intensive and depends on the quality of initial clustering. It is also primarily suited for detecting linear features, which may limit its effectiveness in more complex scenarios.

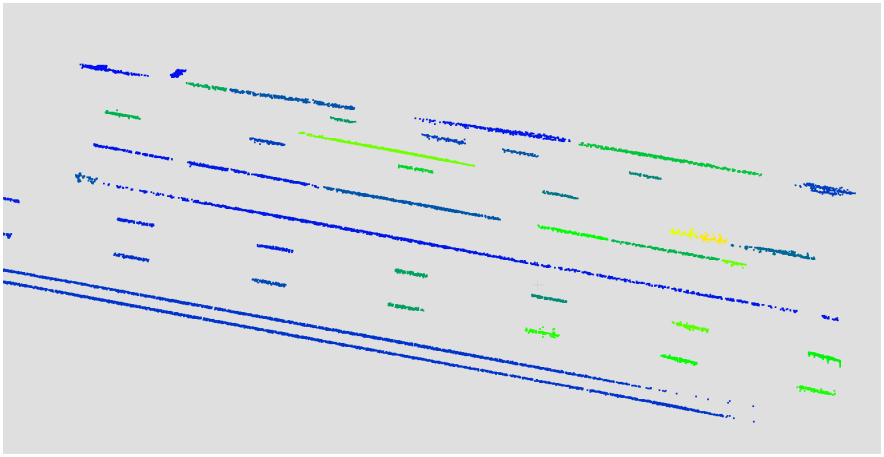

Figure 3.6 The result of detected lanes after RANSAC and PCA based filtering, each color represents distinct cluster.

3.4 Connecting Clusters into Lane Entities

In lane detection, it is often necessary to connect several clusters that represent individual line segments into a continuous lane entity, especially for dashed lanes. This section outlines the methodology used for connecting clusters into lane entities using two approaches: fitting lines with RANSAC and utilizing direction vectors obtained from PCA. Both methods have their advantages and limitations, which are discussed in detail.

3.4.1 Algorithm Description

The algorithm for connecting clusters into lane entities involves several key steps:

1. **Finding Nearest Points**: The algorithm begins by identifying the nearest points between clusters to determine potential connections. This is achieved by calculating the Euclidean distance between the boundary points of each cluster. The pair of points with the shortest distance is considered for potential connection.

2. **Inlier Identification**: For each candidate connection, the algorithm checks if the points in the neighboring cluster align with the direction vector (obtained using PCA) or the line fitted by RANSAC. This involves computing the perpendicular distances of the points to the line or vector and identifying those within a specified threshold as inliers. For a line defined by points P_1 and P_2, the perpendicular distance of a point Q is given by:

$$\text{distance}(Q, \text{line}) = \frac{|(P_2^x - P_1^x)(P_1^y - Q^y) - (P_1^x - Q^x)(P_2^y - P_1^y)|}{\sqrt{(P_2^x - P_1^x)^2 + (P_2^y - P_1^y)^2}}$$

3. **Cluster Connection**: Clusters are connected if the points are within a specified distance threshold. This step ensures that only clusters with consistent directionality and proximity are merged.

4. **Line and Direction Vector Calculation**: Once two clusters are connected, the new cluster formed by their union is analyzed to update the line model (using RANSAC) and the direction vector (using PCA). This involves recomputing the principal components and fitting a new line to the combined set of points.

3.4.2 Connecting Lanes Using Fitted Lines

The RANSAC (Random Sample Consensus) algorithm is used to iteratively fit lines to the filtered clusters. RANSAC is robust to noise and can accurately identify straight lanes by focusing on the inliers that fit the line model. Unfortunately, it struggles with connecting lanes on turns, as the model assumes linearity which may not hold for curved lanes.

3.4.3 Connecting Lanes Using Direction Vectors

Principal Component Analysis (PCA) is used to determine the direction vectors for each cluster. These vectors are then used to connect clusters into continuous lanes, especially useful for lanes that curve. PCA is applied to each cluster to obtain the principal components. The first principal component provides the

direction vector, which is used to connect clusters by checking the alignment and spatial proximity of their direction vectors. PCA is effective in connecting lanes on turns as it captures the dominant direction of the clusters, but it may fail if the direction vector is not accurately determined, leading to incorrect connections.

3.4.4 Comparison of Approaches

In this thesis, we employ two custom approaches for lane detection: one using RANSAC and the other using PCA. While RANSAC excels at connecting straight lanes, it is less effective on turns where the lanes are curved. On the other hand, PCA-based direction vectors are better at handling curved lanes but may occasionally produce errors if the direction vectors are not correctly identified.

The metrics for these two approaches will be compared in Chapter 5. This comparison will provide a detailed analysis of their strengths and weaknesses in different lane detection scenarios.

To visually compare the effectiveness of both approaches, consider the figures 3.7 and 3.8. The main full bottom line is not connected using RANSAC due to its inability to handle non-straight paths effectively. However, this line is successfully connected using PCA, which is adept at dealing with curved segments. Despite this advantage, PCA sometimes connects parts of two parallel dashed lines as one continuous line due to incorrect estimation of the direction vector, potentially caused by noise in the data.

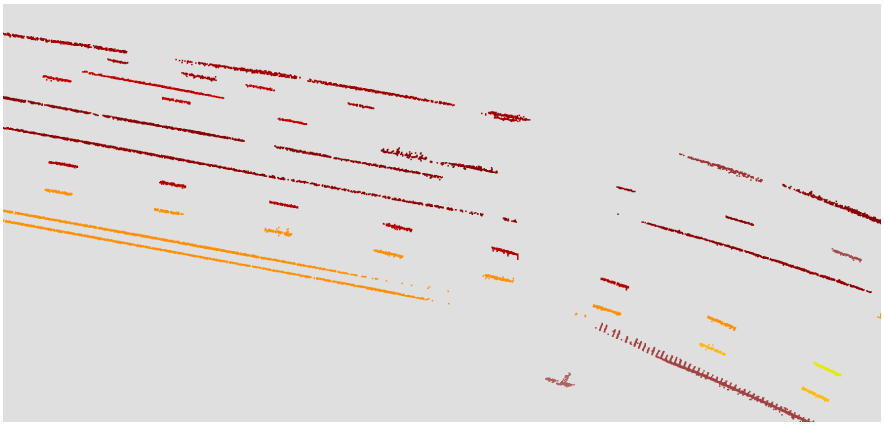

Figure 3.7 Lanes connected using RANSAC, each color represents line entity.

Both RANSAC and PCA-based methods have their specific use cases and can complement each other in a robust lane detection system. RANSAC is preferred for straight lane segments, while PCA is better suited for detecting and connecting curved lanes. The choice of method depends on the specific characteristics of the road and the required accuracy for lane detection. By integrating both approaches, it is possible to achieve a more comprehensive and accurate lane detection system.

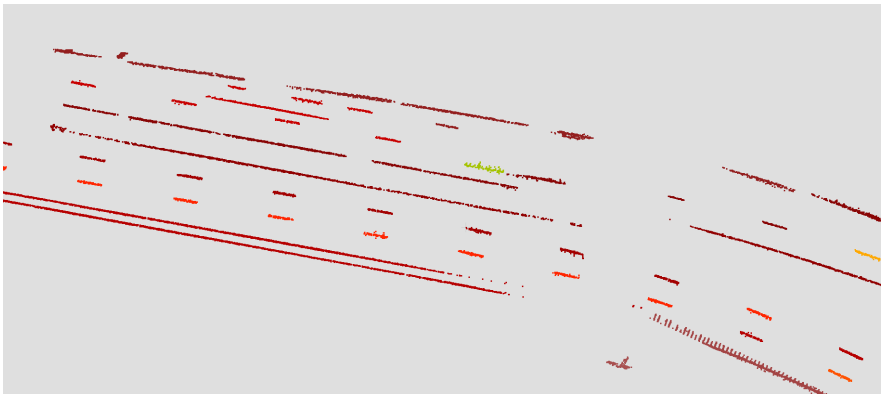

Figure 3.8 Lanes connected using PCA direction vectors, each color represents line entity.

3.5 Classification of Dashed or Full Lanes

Classifying lanes as dashed or full is a crucial step in understanding road markings and ensuring accurate lane detection. This classification is based on the analysis of the clusters that form the lane lines. By examining the distance between clusters and the size of the clusters, we can determine whether a lane is dashed or full.

3.5.1 Algorithm Description

The classification process involves two distinct approaches: size-based classification and distance-based classification. Each approach has its advantages and limitations, and they can be compared for effectiveness in different scenarios.

Size-Based Classification

In the size-based classification approach, the focus is on evaluating the size of each cluster that constitutes a lane.

- **Cluster Size Evaluation**: Measure the size of each cluster along the direction of the lane. This involves calculating the length of the cluster in the direction of the lane's principal axis.
- **Standard Deviation Calculation**: Calculate the standard deviation of the cluster sizes. This gives a measure of the variation in cluster sizes.

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2}$$

where x_i is the size of the i-th cluster, μ is the mean size, and N is the number of clusters.

- **Classification Criteria**:
  - If the standard deviation of the cluster sizes tends to zero and the clusters are relatively large, classify the lane as full.
  - If the standard deviation of the cluster sizes is small and the clusters are of similar, smaller sizes, classify the lane as dashed.

Distance-Based Classification

In the distance-based classification approach, the focus is on measuring the distances between adjacent clusters.

- **Distance Measurement**: Measure the distances between the endpoints of adjacent clusters. This involves calculating the Euclidean distance between the end of one cluster and the start of the next.

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

  where (x_1, y_1) and (x_2, y_2) are the coordinates of the endpoints of adjacent clusters.

- **Standard Deviation Calculation**: Calculate the standard deviation of these distances to assess their consistency.

$$\sigma_d = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(d_i - \mu_d)^2}$$

  where d_i is the distance between the i-th pair of clusters, μ_d is the mean distance, and N is the number of distances measured.

- **Classification Criteria**:
  - If the standard deviation of the distances tends to zero and the distances are relatively small, classify the lane as full.
  - If the standard deviation of the distances is small and the distances are consistent, classify the lane as dashed.

3.5.2 Comparison of Approaches

Both size-based and distance-based classification methods have their advantages and limitations:

- **Size-Based Classification**:

  - **Advantages**: Effective for identifying lanes where cluster sizes are uniform, easy to implement.
  - **Limitations**: May misclassify dashed lanes if cluster sizes vary due to noise or measurement errors.

- **Distance-Based Classification**:
  - **Advantages**: Effective for identifying lanes where the spacing between clusters is uniform, robust to variations in cluster size.
  - **Limitations**: May misclassify full lanes if distances between clusters vary due to noise or measurement errors.

To visually compare the effectiveness of both approaches, consider the figures 3.10 and 3.9.

Distance-based classification generally provides better results due to its robustness to variations in cluster size. By focusing on the distances between clusters, this method is less affected by the inherent variability in cluster sizes that can arise from noise or measurement errors.

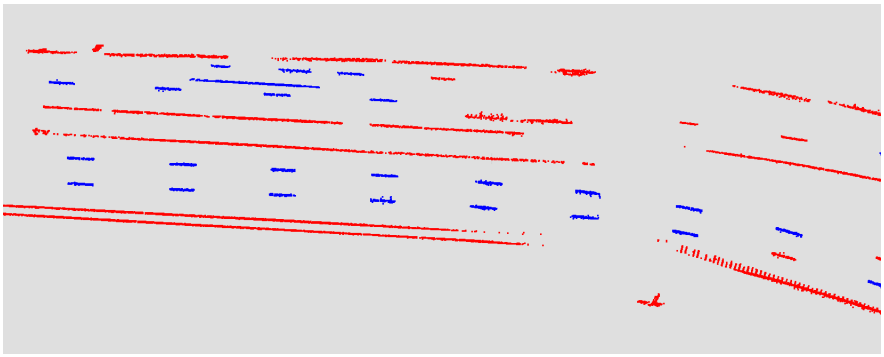

Figure 3.9 Distance-based classification. Consistent small distances between clusters indicate a full lane, while consistent larger distances indicate a dashed lane.

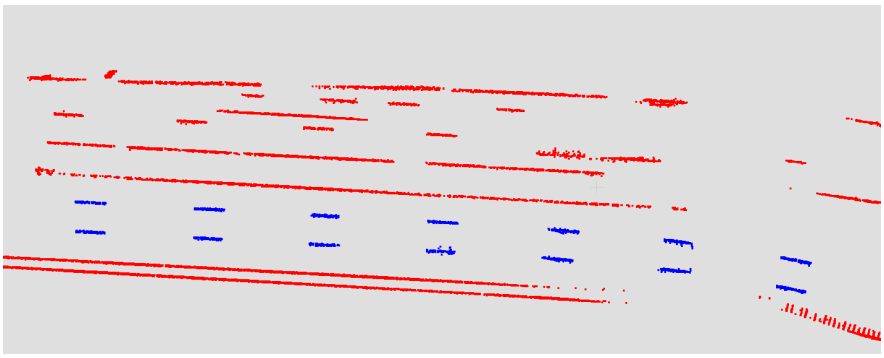

Figure 3.10 Size-based classification. Consistent large cluster sizes indicate a full lane, while consistent smaller sizes indicate a dashed lane.

Classifying lanes as dashed or full based on cluster analysis allows for a more nuanced understanding of road markings. By measuring the distances between clusters and evaluating their sizes, we can accurately distinguish between dashed

and full lanes, enhancing the overall lane detection process. This methodology, combined with the calculation of standard deviations, ensures a robust classification system that adapts to varying road conditions and markings. Both size-based and distance-based approaches offer valuable insights and can be used complementarily to improve the accuracy of lane classification.

3.6 Algorithm Description

Our final lane detection method includes the following key steps:

1. **Preprocessing:**
    - **Downsampling:** Reduce the number of points in the point cloud using voxel grid downsampling to ensure efficient processing while retaining essential structural features. (Section 2.5)
    - **Outlier Removal:** Use Statistical Outlier Removal to eliminate noise from the point cloud by analyzing the distribution of distances to a point's neighbors. (Section 2.4)
2. **Extraction of Ground Plane:**
    - Apply the RANSAC algorithm to segment the ground plane from the point cloud data. This involves fitting a plane model and distinguishing inliers that lie on the plane from outliers. (Section 3.2)
3. **Clustering:**
    - Use DBSCAN to divide the remaining data into clusters. DBSCAN groups points that are closely packed together while marking points that lie alone in low-density regions as outliers. (Section 3.1)
4. **Filtering Non-Lane Clusters:**
    - **RANSAC Line Fitting:** For each cluster, apply the RANSAC algorithm to fit a line. Clusters that do not form a valid line are filtered out. (Section 3.2)
    - **PCA Analysis:** Perform Principal Component Analysis on each cluster to analyze its geometric features. Clusters that do not align with the principal direction are filtered out. (Section 3.3)
5. **Connecting Lane Clusters:**
    - Use custom algorithm to connect the remaining clusters into lane entities. This process utilizes direction vectors derived from PCA to ensure that connected clusters maintain consistent directionality. (Section 3.4)
6. **Classification of Lane Entities:**

- Use custom algorithm to classify the connected lane entities into two categories: full lanes and dashed lanes. This classification is based on the size of the clusters and the distances between them. (Section 3.5)

3.7 Conclusion

In this section, we present a comprehensive methodology for lane detection that integrates well-known algorithms with our unique approach. Our method uses the strengths of DBSCAN clustering, RANSAC, and PCA, enhanced by custom algorithms that utilize geometric features of the data for filtering non-lanes, connecting lane objects, and classification.

Our approach begins with DBSCAN clustering to identify potential lane segments. Following this, we apply a combination of PCA and RANSAC to refine these clusters. RANSAC is used in identifying inliers that fit a lane model, ensuring that only relevant points are considered. PCA is used to determine the principal direction of each cluster, enabling us to filter out non-lane points by analyzing their alignment with the direction vector.

To connect lane objects, we use custom algorithms that assess the geometric properties of the clusters. These algorithms calculate the nearest points between clusters and evaluate their alignment using PCA and RANSAC-fitted lines. This ensures that connected clusters maintain consistent directionality and proximity, forming continuous lane entities.

For classification, our method evaluates both the size and distance between clusters. By calculating the standard deviation of cluster sizes and the distances between adjacent clusters, we classify lanes as either full or dashed based on their geometric characteristics.

Evaluation results and comparisons with other methods will be detailed in chapter 5. This will provide a clearer picture of the performance and accuracy of our approach in real-world scenarios.

Chapter 4 will delve into the implementation considerations and parameter estimation for each algorithm, providing practical insights into the deployment of our lane detection methodology.

4 Experiments and Implementation Considerations

In this chapter, we delve into the practical aspects of our lane detection methodology (3.6). This includes a detailed description of the implementation, the scripts developed for each task, and the experiments conducted to estimate the optimal parameters for each algorithm. By providing insights into the coding and experimental processes, this chapter aims to offer a comprehensive understanding of how our approach was realized and validated.

We will start by presenting the scripts that were developed to perform various tasks within our lane detection framework. Each script is designed to address specific challenges, from clustering and line fitting to connecting lane segments and classifying lanes as dashed or full. Following the script descriptions, we will discuss the experimental setups and results that informed our parameter choices and demonstrated the effectiveness of our approach.

4.1 Data Loading and Management

To facilitate interaction with the data provided in the Agroverse 2 Sensor dataset, a script named `dataloader.py` has been developed. This script serves as an abstraction layer for retrieving and managing data from the dataset. It includes functionality to synchronize and transform sensor data between different reference frames and temporal points, ensuring the integrity and usability of the data for research purposes. Additionally, the dataloader supports the extraction and management of vehicle poses, LIDAR paths, and various other elements crucial for processing the dataset.

4.1.1 Script Overview

The main class in this script is `SensorDataLoader`, which provides several key methods:

- `init(self, data_dir:, labels_dir: Path) -> None:`

  Initializes paths for data and labels directories.

- `get_city_pose(self, log_id: str, timestamp_ns: int) -> SE3:`

  Retrieves the vehicle's pose as an `SE3` object from the dataset for a given log ID and timestamp.

- `get_log_map_path(self, log_id: str) -> Path:`
  Returns the path to the map directory for a given vehicle log.
- `get_city_name(self, log_id: str) -> str:`
  Extracts the city name from the vector map file associated with the log ID.
- `list_log_ids(self) -> list:`
  Lists all log IDs available in the dataset directory.
- `get_lidar_path(self, log_id: str, timestamp_ns: int) -> Path:`
  Provides the path to the LIDAR data file for a specific log ID and timestamp.
- `get_ordered_lidar_timestamps(self, log_id: str) -> list:`
  Returns a list of chronologically ordered timestamps for each LIDAR sweep in a log.
- `get_ordered_lidar_paths(self, log_id: str) -> list:`
  Retrieves and sorts all LIDAR file paths in a single vehicle log.

The script also includes helper functions to convert and manipulate pose data:

- `convert_pose_to_SE3(df: pd.DataFrame) -> SE3:`
  Converts a DataFrame containing pose data into an `SE3` transformation object. This involves extracting quaternion and translation components and converting them into a rotation matrix.
- `quat_to_mat(quat_wxyz: NDArrayFloat) -> NDArrayFloat:`
  Converts a quaternion to a rotation matrix.

4.1.2 IO Utilities

In addition to the `dataloader.py` script, another crucial component of our data processing pipeline is the `io_utils.py` script. This script provides utility functions for reading data from Feather files, which are a common format used in the Agroverse 2 Sensor dataset. The functions in this script streamline the process of loading and converting data into usable formats for further processing.

The `io_utils.py` script provides the following key functions:

- `read_feather(path: Path, columns: [Tuple[str, ...]] = None):`
  This function reads data from a Feather file located at the specified path and returns it as a pandas DataFrame. An optional parameter allows the selection of specific columns to be read, improving efficiency when only a subset of the data is needed.

- `read_lidar_sweep(path: Path)`:

  This function reads a LIDAR sweep from a Feather file and converts it into a NumPy array of floating-point numbers. The resulting array includes the `x`, `y`, `z` coordinates and the `intensity` values of the LIDAR points.

The `io_utils.py` script relies on the `pyarrow.feather` module for reading Feather files. This module is optimized for high-performance reading and writing of columnar data, making it well-suited for handling large-scale datasets commonly encountered in autonomous vehicle research.

4.1.3 SE3 Class for Rigid Body Transformations

The `SE3` class is a fundamental component used in the `dataloader.py` script. The `se3.py` encapsulates functionalities for performing rigid transformations on point clouds using the SE(3) lie group [8]. This script provides the mathematical foundation for accurately transforming point cloud data from the ego-vehicle's local coordinate frame to a global city coordinate system by applying rotation and translation parameters encapsulated in the SE(3) transformation matrices.

When retrieving the vehicle's pose at a specific timestamp, the `get_city_pose` method utilizes the `SE3` class to convert the pose data from the dataset into an `SE3` object. This object can then be used to transform LIDAR points from the vehicle's local frame to the global frame, ensuring all data is aligned correctly for further processing.

4.2 Point Cloud Accumulation

The `accumulate.py` script is an essential component of our data processing pipeline, designed to aggregate point cloud data from multiple LIDAR sweeps into a single, comprehensive dataset. This accumulated point cloud provides a detailed and continuous representation of the environment, crucial for tasks such as mapping and lane detection. This script processes individual LIDAR sweeps, transforms their coordinates from the ego-vehicle frame to the global city frame, and aggregates the transformed points into a single dataset. This process enhances the spatial coverage and density of the point cloud, enabling more accurate and detailed analysis.

4.2.1 Script Overview

The main steps in the `accumulate.py` script include:

- Initialization: The script initializes the data loader and sets up paths for data directories.
- Processing Each Log: For each log in the dataset, the script retrieves and processes all LIDAR sweeps.
- Transformation and Accumulation: Each LIDAR sweep is transformed from the ego-vehicle frame to the city frame and then accumulated into a single dataset.
- Saving Accumulated Points: The aggregated point cloud data is saved to a file for further analysis.

The `accumulate.py` script uses functions from the `io_utils.py` and `dataloader.py` scripts to read and transform data efficiently.

4.3 Integration with Point Cloud Library

To effectively manage the complexity of operations required for the manipulation of accumulated LIDAR point clouds, our pipeline integrates the Point Cloud Library (PCL). [14]

Recognized for its comprehensive suite of tools designed for 3D data processing, PCL enables efficient and robust management of spatial data. The library includes advanced algorithms such as Octrees (Section 2.3) and KD-trees for spatial partitioning, voxel grid downsampling (Section 2.5) to reduce data size while preserving essential structural features, and a statistical outlier removal filter that improves data quality by eliminating noise (Section 2.4).

The incorporation of PCL into our system is driven by its optimized performance capabilities, which are essential for handling the large volumes of data generated by high-resolution LIDAR sensors. Its flexibility allows for adaptation to various 3D data structures and processing needs, supporting a broad range of applications in autonomous vehicle navigation and 3D mapping. PCL's reliability, underpinned by a community of expert developers, ensures the accuracy and robustness of its 3D spatial computations.

Incorporating specific classes from PCL enhances our system's processing capabilities. The use of `pcl::StatisticalOutlierRemoval`, `pcl::Octree`, and `pcl::VoxelGrid`, as outlined in PCL documentation, allows for structured and optimized handling of tasks such as data structuring, downsampling, and noise reduction. These classes support the high-performance demands of our applications, ensuring the creation of accurate and reliable environmental models necessary for autonomous vehicle operations.

4.3.1 Preprocessing LIDAR Point Clouds

The `preprocessing.cpp` script performs essential preprocessing tasks on LIDAR point clouds, including statistical outlier removal and voxel grid downsampling. The script reads input point clouds, applies filters to clean and downsample the data, and writes the processed point cloud to an output file.

Workflow of Preprocessing

1. **Reading the Point Cloud:** The script begins by reading the input point cloud file using `pcl::PCDReader`. This file contains the raw LIDAR data.
2. **Statistical Outlier Removal:** The `pcl::StatisticalOutlierRemoval` filter is used to remove noise from the point cloud. The effectiveness of the Statistical Outlier Removal depends heavily on the choice of parameters k and k_factor, designated in the implementation as `MeanK` and `StddevMulThresh`.
   - `meanK`: Number of nearest neighbors to analyze for each point.
   - `stddevMulThresh`: Standard deviation multiplier threshold. Points with a mean distance larger than this threshold from the average distance are considered outliers and removed.

   The chosen parameters ensure a balance between removing noise and retaining important structural features.
3. **Voxel Grid Downsampling:** The `pcl::VoxelGrid` filter downsamples the point cloud to reduce its size while preserving the geometric structure. The `leaf_size` parameter defines the size of the voxels used for downsampling.
4. **Writing the Processed Point Cloud:** Finally, the processed point cloud is written to an output file using `pcl::PCDWriter`. This file contains the filtered and optionally downsampled LIDAR data, ready for further processing.

Parameter Selection

In our specific implementation, as documented in the `preprocessing.cpp` file, `MeanK` was set to 100, indicating that each point's status (inlier or outlier) is determined based on its relationship with the nearest 100 points. This choice reflects a balance between ignoring too many data points and avoiding the removal of too few outliers. The standard deviation multiplier threshold (`StddevMulThresh`) was set to 1.0, which establishes a moderate boundary for determining outliers, aiming to keep the data clean yet comprehensive.

The parameter `leaf_size` was set to 0.1. This parameter is critical as it defines the size of the voxel in all three spatial dimensions (X, Y, and Z). In the context

of our project, the chosen voxel size is 0.1 units, which effectively represents the resolution of the point cloud after downsampling.

4.3.2 Ground Plane Extraction and Projection

The `ground_extraction.cpp` script focuses on identifying the ground plane within a LIDAR point cloud and projecting the points onto a 2D plane. This process is critical for tasks such as road surface detection and lane marking extraction.

Workflow of Ground Extraction

1. **Loading the Point Cloud:** The script loads the original point cloud data using `pcl::io::loadPCDFile`, which reads the LIDAR data into a `pcl::PointCloudpcl::PointXYZ` object.
2. **Normal Estimation:** Normals for each point are computed using `pcl::NormalEstimation`. This step involves:
   - Setting the input cloud and search method.
   - Using a KdTree for efficient neighbor search.
   - Computing the normals with `KSearch = 50` (the number of nearest neighbors to use when estimating the normals of the points in the point cloud), which ensures a robust estimation.
3. **Ground Plane Segmentation:** The ground plane is segmented using `pcl::SACSegmentationFromNormals`, configured with the following parameters:
   - `setModelType(pcl::SACMODEL_NORMAL_PLANE)`: Specifies the model to fit a plane using point normals.
   - `setMethodType(pcl::SAC_RANSAC)`: Utilizes the RANSAC algorithm for robust fitting.
   - `setNormalDistanceWeight(0.1)`: Balances the influence of point normals in the segmentation.
   - `setMaxIterations(1000)`: Ensures a thorough search for the best model.
   - `setDistanceThreshold(2)`: Defines the maximum distance for a point to be considered an inlier.
4. **Extracting the Ground Plane:** The ground plane points are extracted using `pcl::ExtractIndices`. This filter isolates the inliers identified during segmentation, producing a point cloud that represents the ground surface.
5. **Projecting to 2D Plane:** The extracted ground plane is projected onto a 2D plane using `pcl::ProjectInliers`. This step involves defining a model

coefficient that specifies the projection plane and applying the projection to the point cloud.

6. **Writing the Projected Point Cloud:** The final projected point cloud is written to an output file using `pcl::PCDWriter`.

Parameter Selection

The parameter selection includes `setKSearch(50)` for providing a reliable neighborhood size for normal estimation, `setNormalDistanceWeight(0.1)` to balance normal vector influence in segmentation, `setMaxIterations(1000)` to ensure comprehensive model fitting, and `setDistanceThreshold(2)` to define the tolerance for inliers in the plane model.

4.4 DBSCAN Clustering

A script named `clustering.py` was developed to handle the DBSCAN clustering. This script is designed to intake preprocessed LIDAR point cloud data, where noise has been minimized and the points have been downsampled. The values chosen for ϵ and $MinPts$ were determined to balance the sensitivity of the algorithm to small variations in density while preventing the merging of distinct lanes into a single cluster.

4.4.1 Script Overview

The `clustering.py` script performs the following key tasks:

1. Loading Point Cloud Data: The script starts by loading the preprocessed point cloud data from a `.ply` file using the `PyntCloud` library. This data represents the LIDAR point cloud after noise reduction and downsampling.

2. Applying DBSCAN Clustering: The `apply_dbscan_clustering` function applies the DBSCAN algorithm to the point cloud data. The parameters ϵ and $MinPts$ control the sensitivity and minimum cluster size, respectively. The function returns labels for each point, indicating the cluster to which it belongs. Points labeled as noise (with a label of -1) are filtered out. This ensures that only points belonging to meaningful clusters are retained.

3. Saving Clustered Data: The `save_segments_to_point_cloud` function saves the clustered point cloud data into a new `.ply` file. Each cluster is assigned a unique label, which can be used for visualization and further analysis.

4.4.2 Practical Implementation and Parameter Selection

In our initial tests, several configurations of ϵ and $MinPts$ were experimented with:

- A smaller ϵ tended to break a single lane into multiple clusters, especially in curved lane segments (Figure 3.3).
- Conversely, a larger ϵ sometimes resulted in multiple adjacent lanes being detected as a single cluster, particularly in areas where lanes are close to each other (Figure 3.2).

Adjustments were made to improve the lane detection process. After several iterations, we found optimal parameter settings that clearly separated the lanes without mistakenly combining them or breaking them into smaller parts. The parameter ϵ was set to 0.3 and $MinPts$ was set to 10. The result of these settings is shown in the Figure 3.1.

The practical implementation of DBSCAN in this script not only highlights the challenges faced but also showcases the potential of this clustering technique in real-world applications like autonomous driving. The visual results, as shown in the figures, clearly illustrate the importance of parameter tuning in achieving precise and reliable lane detection in LIDAR data.

By integrating DBSCAN clustering with careful parameter selection, our approach effectively identifies and separates lanes in the LIDAR point cloud data, laying the groundwork for further processing and analysis in autonomous vehicle navigation systems.

4.5 RANSAC Filtering

Line fitting is implemented using a Python script named `ransac_filter.py`. This script applies the RANSAC algorithm to each cluster detected previously using the DBSCAN clustering algorithm. Here, a line is fitted to the points within each cluster, and the validity of this line is determined based on the number of inliers. If the number of inliers is greater than 60% of the total points in the cluster, the cluster is considered to effectively represent a lane marking.

4.5.1 Script Overview

The `ransac_filter.py` script performs the following key tasks:

1. Loading Point Cloud Data: The script starts by loading the clustered point cloud data from a `.ply` file using the `PyntCloud` library. This data represents the LIDAR point cloud after clustering with DBSCAN.

2. Applying RANSAC Fitting: The `apply_ransac_fitting` function applies the RANSAC algorithm to the point cloud data. The parameters `residual_threshold` controls the maximum distance a point can have from the fitted line to be considered an inlier. The function returns a dictionary of line models for each cluster that meets the inlier criteria.

3. Filtering Out Small Clusters: Clusters with fewer than 10 points are filtered out. This step removes noise and insignificant data that could otherwise lead to incorrect lane detection. Additionally, clusters with a high percentage of inliers are retained, ensuring that only significant lane markings are considered.

4. Saving Filtered Line Segments: The `save_segments_to_point_cloud` function saves the filtered line segments into a new `.ply` file. Each line segment is assigned a unique label, which can be used for visualization and further analysis.

The line models resulting from the RANSAC fitting process are stored as objects in a dictionary. Each object contains:

- **Intercept**: The y-intercept of the line model.
- **Coefficient**: The slope of the line.
- **Inlier Mask**: A binary mask indicating which points in the cluster are inliers to the line.

This structured representation not only facilitates the manipulation and analysis of detected lines but also helps with subsequent processes such as visualization and further statistical analysis.

4.5.2 Practical Implementation and Parameter Selection

In our implementation, the optimal residual threshold for determining inliers versus outliers during the line fitting process was set to 2. This threshold value balances the need for precision in line detection against the robustness required to handle typical noise and irregularities in LIDAR data.

The practical implementation of RANSAC in this script enhances the accuracy of lane detection by ensuring that only significant lane markings are retained.

4.6 PCA Filtering

The implementation of PCA for lane detection involves specific tools and scripts to ensure accurate and efficient processing of LIDAR data. The clustering step is implemented using the DBSCAN algorithm to segment the LIDAR point cloud into clusters. Each cluster is then processed individually using PCA.

4.6.1 Script Overview

The `pca_filter.py` script performs the following key tasks:

1. Loading Point Cloud Data: The script starts by loading the clustered point cloud data from a `.ply` file using the `PyntCloud` library. This data represents the LIDAR point cloud after clustering with DBSCAN and filtering with RANSAC.
2. Applying PCA: The `make_cluster_dict` function applies the PCA algorithm to the point cloud data. The function computes the principal components and the percentage of variance explained by each component. The direction vector of the line is extracted from the first principal component (PC1).
3. Filtering Based on Variance: Clusters are filtered based on the variance explained by PC1 and PC2. If the variance explained by PC2 is less than 0.05, the cluster is considered a potential lane marking. This ensures that only clusters with significant linear characteristics are retained.
4. Extracting Direction Vectors: For each cluster, the direction vector of the line is extracted from PC1. Additionally, the mean point, the minimum point, and the maximum point along the direction vector are calculated.
5. Saving Filtered Line Segments: The `save_segments_to_point_cloud` function saves the filtered line segments into a new `.ply` file. Each line segment is assigned a unique label, which can be used for visualization and further analysis.

The resulting line models are stored as objects in a dictionary. Each object contains:

- **Points**: The original points in the cluster.
- **Mean Point**: The mean point of the cluster.
- **Point Min**: The minimum point along the direction vector.
- **Point Max**: The maximum point along the direction vector.

- **Direction Vector**: The direction vector of the line determined by PC1.

This structured representation facilitates further processing, visualization, and analysis of detected lanes.

4.6.2 Practical Implementation and Parameter Selection

In our implementation, the threshold for the variance explained by PC1 was set to 0.3, and the threshold for the variance explained by PC2 was set to 0.05. These thresholds ensure that clusters with significant linear characteristics are identified as potential lane markings.

The practical implementation of PCA in this script enhances the accuracy of lane detection by ensuring that only significant linear features are retained.

4.7 Joining Clusters into Lane Entities

The process of joining clusters into lane entities is crucial for accurately representing continuous lanes from segmented LIDAR data. The script `join.py` is designed to merge these clusters based on geometric and statistical properties, ensuring that the resulting lane entities are coherent and accurately reflect the road structure.

4.7.1 Script Overview

The `join.py` script performs the following key tasks:

1. Loading Cluster Data: The script starts by loading the cluster data from a `.pickle` file using the `pickle` module. This data contains the clusters identified from the previous DBSCAN, PCA and RANSAC processing steps.
2. Finding Nearest Clusters: The `find_nearest_points` function calculates the minimum distance between the boundary points of two clusters. This is done by considering all pairwise distances between the minimum and maximum points of the clusters. The nearest pair of points determines the potential connection between clusters.
3. Joining Clusters: The `find_next_cluster` function identifies the best cluster to join based on the distance and inlier percentage. The script iterates through the clusters and uses the PCA direction vector or RANSAC line to check if the clusters are aligned. If the minimum distance is less than the threshold (10 units) and the inliers percentage is greater than 70%, the clusters are joined. The `join_clusters` function performs the actual merging

of clusters. It concatenates the points from two clusters and recalculates the PCA components and RANSAC line for the new cluster. The merged cluster is then added to the dictionary of clusters, and the process repeats until no more clusters can be joined.

4. Recomputing Principal Components and RANSAC Lines: For each newly formed cluster, the script recalculates the principal components using PCA. The first principal component provides the new direction vector of the cluster. Additionally, the script fits a line using the RANSAC algorithm to ensure the linearity of the cluster. This helps in accurately modeling the lane markings.

5. Filtering by Common Direction: The `most_common_direction` function filters clusters based on the similarity of their direction vectors. This function calculates the mean direction vector of all clusters and normalizes it to a unit vector. It then checks the cosine similarity between the mean direction and the direction vector of each cluster. Clusters with a cosine similarity below a specified threshold are considered to have a common direction and are retained.

6. Saving Joined Clusters: The `save_segments_to_point_cloud` function saves the joined clusters into a new point cloud file (`joined.ply`). Each cluster is assigned a unique label for visualization. The script ensures that the points are correctly labeled and saved for further use.

4.7.2 Practical Implementation and Parameter Selection

The main parameters involved in the clustering and joining process include the threshold for the minimum distance between clusters (set to 10) ensuring clusters are close enough to be considered for joining and the inlier percentage threshold for considering a valid connection (set to 70%).

The practical implementation of cluster joining in this script enhances the coherence of lane entities by merging fragmented clusters based on geometric alignment and proximity.

4.8 Classification of Dashed or Full Lanes

The script `classify.py` is designed to classify lane entities as either dashed or full based on the size of the clusters and the distances between them. This classification is essential for understanding the nature of lane markings and ensuring accurate lane detection.

4.8.1 Script Overview

The `classify.py` script performs the following key tasks:

1. Loading Point Cloud Data: The script starts by loading the joined point cloud data from a `.ply` file using the `PyntCloud` library. This data represents the LIDAR point cloud after the clusters have been joined into lane entities.
2. Classifying Clusters: The script applies size-based and distance-based classification to determine whether each lane entity is dashed or full.
3. Saving Classified Data: The resulting classifications are saved into a new point cloud file for visualization and further analysis.

4.8.2 Practical Implementation and Parameter Selection

The classification process uses the following thresholds:

- `distance-based classification`: The standard deviation threshold for distances between clusters is set to 5.
- `size-based classification`: The standard deviation threshold for the size of clusters is set to 300.

The practical implementation of this script effectively classifies lane entities based on their geometric properties. The visual results, as saved in the `classify.ply` file, provide clear differentiation between dashed and full lanes, facilitating accurate lane detection and mapping.

By integrating this script into our pipeline, we achieve a robust classification of lane entities, improving the accuracy and reliability of lane detection in autonomous vehicle navigation systems.

5 Evaluation Results

This chapter evaluates the performance of lane detection using various metrics, including Overall Accuracy, Mean Intersection over Union (IoU), Mean Class Accuracy, and Mean Average Precision (mAP). The metrics were tested on both binary segmentation (whether a point in the point cloud is a lane or not) and multi-class segmentation (where each lane type, such as dashed or full, is classified separately).

5.1 Metrics and Formulas

5.1.1 Overall Accuracy

Overall Accuracy measures the proportion of correctly classified points over the total number of points. It is calculated as:

$$\text{Overall Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

where TP is True Positives, TN is True Negatives, FP is False Positives, and FN is False Negatives.

5.1.2 Mean Intersection over Union (IoU)

Mean IoU evaluates the average overlap between the predicted segmentation and the ground truth segmentation. It is calculated as:

$$\text{IoU} = \frac{TP}{TP + FP + FN}$$

Mean IoU is the average IoU across all classes.

5.1.3 Mean Class Accuracy

Mean Class Accuracy measures the average accuracy of each class. It is calculated as:

$$\text{Class Accuracy} = \frac{TP}{TP + FN}$$

Mean Class Accuracy is the average Class Accuracy across all classes.

5.1.4 Mean Average Precision (mAP)

Mean Average Precision assesses the precision of the model averaged over all classes and thresholds. Precision is calculated as:

$$\text{Precision} = \frac{TP}{TP + FP}$$

mAP is the mean of the Average Precision scores for all classes.

5.2 Results

The evaluation was conducted on original point clouds before pre-processing. These point clouds were manually labeled to establish ground truth, as illustrated in Figures 5.1 and 5.2. The primary objective of this task was to develop a segmentation method. It is important to note that while our method focuses on segmentation, other existing methods typically fit polylines for lane detection. This fundamental difference in approach makes direct comparisons with existing methods challenging. The results of our evaluation, therefore, highlight the effectiveness of our segmentation-based approach within the scope of this project.

The results indicate that binary segmentation yields higher performance across all metrics compared to multi-class segmentation. The improved performance in binary segmentation is attributed to its ability to simplify the classification process, reducing the likelihood of misclassification and enhancing overall model accuracy.

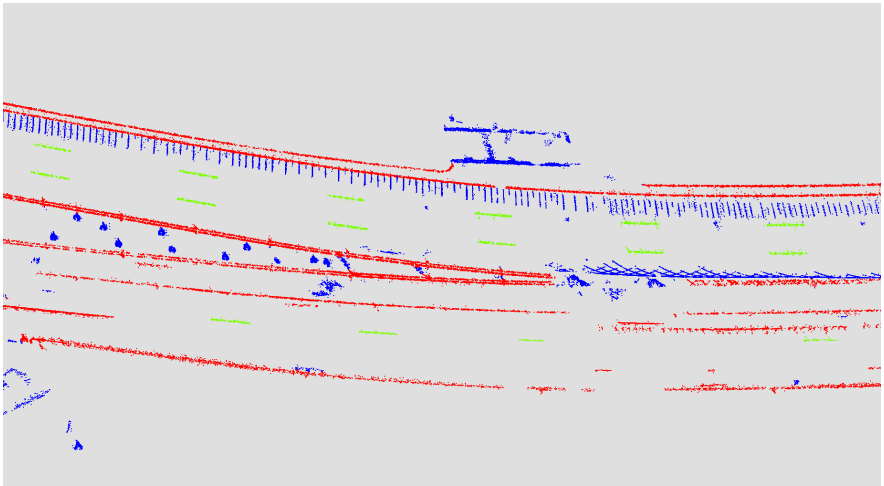

Figure 5.1 Ground Truth for P1

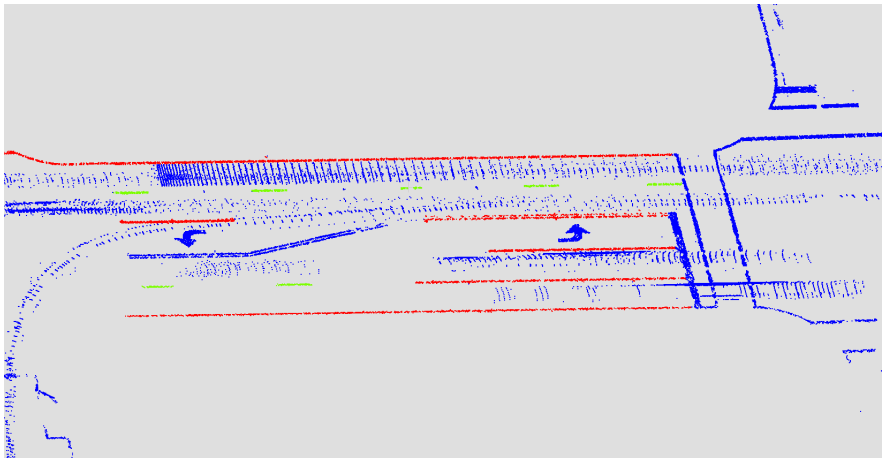

Figure 5.2 Ground Truth for PC2

The results for each point cloud, as presented in the following tables, were calculated using the parameters specified in Chapter 4 and the algorithm described in Section 3.6. These parameters have been identified as optimal, as they yield the highest metrics compared to other tested parameters and approaches. Specifically, we used PCA-based connection methods, which involve aligning clusters based on their principal component directions to form lane entities (Section 4.7). Additionally, we employed a classification approach based on both the size of the clusters and the distances between them, which ensures accurate differentiation between full and dashed lanes by assessing the geometric properties of the lane segments (Section 4.8).

For binary segmentation, the metrics results on four point clouds are as follows:

Metric	**PC 1**	**PC 2**	**PC 3**	**PC 4**
Overall Accuracy	0.9883	0.9909	0.9903	0.9815
Mean IoU	0.6844	0.6284	0.6598	0.6605
Mean Class Accuracy	0.7046	0.6911	0.6925	0.7029
Mean Average Precision	0.6720	0.5889	0.6486	0.6688

Table 5.1 Binary Segmentation Results with Optimal parameters and PCA-based Connection

For multi-class segmentation, the metrics results on four point clouds are as follows:

Metric	PC 1	PC 2	PC 3	PC 4
Overall Accuracy	0.9879	0.9906	0.9903	0.9815
Mean IoU	0.6324	0.5001	0.5598	0.5605
Mean Class Accuracy	0.6904	0.6230	0.6725	0.6829
Mean Average Precision	0.6087	0.4472	0.5486	0.5788

Table 5.2 Multi-Class Segmentation Results with Optimal parameters and PCA-based Connection

We also experimented with various modifications of our approach. For instance, the table below demonstrates that utilizing RANSAC-based connections for lane entities results in worse performance compared to the PCA-based approach. In the RANSAC-based method, clusters are connected using fitted lines derived from the RANSAC algorithm. This approach is less effective at connecting lanes, especially at road turns, leading to discontinuities in lane entities. Consequently, this variation yields lower performance metrics across all categories, including Mean IoU, Mean Class Accuracy, and Mean Average Precision. The PCA-based approach, which aligns clusters based on their principal component directions, provides smoother connections and better handles the curvature of road lanes, resulting in better performance in all evaluated metrics. Detailed comparison of both approaches is explained in Section 3.4.4.

Metric	PC 1	PC 2	PC 3	PC 4
Overall Accuracy	0.9809	0.9918	0.9884	0.9832
Mean IoU	0.5818	0.4840	0.5203	0.5312
Mean Class Accuracy	0.6217	0.6082	0.6358	0.6489
Mean Average Precision	0.5690	0.4287	0.4314	0.4968

Table 5.3 Multi-Class Segmentation with RANSAC-based Connection Results

Additionally, we explored a variation where clusters were connected using fitted lines from the RANSAC algorithm, and classification was based on the size of the clusters instead of using both size and distance-based criteria (Section 3.5). This approach resulted in significantly degraded performance. The use of size-based classification failed to accurately distinguish between full and dashed lanes, leading to misclassifications. Consequently, this variation produced noticeably lower metrics, including Mean IoU, Mean Class Accuracy, and Mean Average Precision. The combined use of PCA-based connections and both size and distance-based classification criteria in the original approach proved more robust and accurate in lane detection tasks.

Metric	PC 1	PC 2	PC 3	PC 4
Overall Accuracy	0.9787	0.9906	0.9861	0.9843
Mean IoU	0.5121	0.4244	0.4732	0.4487
Mean Class Accuracy	0.6258	0.5885	0.6114	0.6072
Mean Average Precision	0.5089	0.3857	0.4913	0.4112

Table 5.4 Multi-Class Segmentation with RANSAC-based Connection and Size-based Classification Results

The high Overall Accuracy and Mean Class Accuracy in our results can be attributed to the dominance of non-lane points in the LIDAR point clouds. Since the majority of points are not part of a lane, correctly classifying these points as non-lane significantly boosts these metrics.

However, Mean IoU and Mean Average Precision (mAP) are lower. Mean IoU penalizes false positives and false negatives more heavily, highlighting the model's performance in distinguishing lane points from non-lane points. Mean Average Precision (mAP) further considers the precision-recall trade-off for each class. Since lane points are less frequent, small misclassifications impact these metrics more, resulting in lower values.

Conclusion

This thesis has focused on investigating and developing LIDAR-based lane detection techniques, using the capabilities of machine learning algorithms to achieve robust and accurate lane identification.

Throughout this research, we have proposed and implemented a novel method that integrates several well-known unsupervised machine learning techniques, including Principal Component Analysis (PCA), Random Sample Consensus (RANSAC), and Density-Based Spatial Clustering of Applications with Noise (DBSCAN) clustering. The primary objective was to achieve three-dimensional segmentation of LIDAR point clouds, enabling the precise labeling of each point as lane or non-lane and classifying each detected lane by type.

One of the significant advantages of employing unsupervised learning techniques is the ability to perform fast and efficient lane detection without the need for large datasets with ground truth labels. Labeling such datasets manually is not only time-consuming but also prone to human error. Our approach avoids this challenge, providing a reliable solution for real-time applications in autonomous vehicles.

The methods developed in this thesis are not only effective for immediate lane detection but also serve as powerful tools for the pre-processing and labeling of large point cloud datasets. These labeled datasets can subsequently be used to train supervised machine learning models, including deep neural networks, enhancing their performance and accuracy in complex driving environments.

The evaluation results demonstrated the effectiveness of our approach and the integration of unsupervised learning techniques into the lane detection pipeline has proven to be a promising direction for future research and development.

In conclusion, this thesis contributes to the advancement of autonomous vehicle navigation by providing a novel, efficient, and reliable lane detection solution using LIDAR data. The developed methodologies offer benefits for both real-time lane detection and the creation of high-quality training datasets for supervised learning models.

Bibliography

1. Boschetto, Alberto; Bottini, Luana; Macera, Luciano. Design and fabrication by selective laser melting of a LIDAR reflective unit using metal matrix composite material. *The International Journal of Advanced Manufacturing Technology*. 2023, vol. 126, pp. 1–16. Available from doi: 10.1007/s00170-023-11131-8.

2. Zeng, Honghao; Jiang, Shihong; Cui, Tianxiang; Lu, Zheng; Li, Jiawei; Lee, Boon-Giin; Zhu, Junsong; Yang, Xiaoying. ScatterHough: Automatic Lane Detection from Noisy LiDAR Data. *Sensors*. 2022, vol. 22, no. 14. issn 1424-8220. Available from doi: 10.3390/s22145424.

3. Jung, Jiyoung; Bae, Sung-Ho. Real-Time Road Lane Detection in Urban Areas Using LiDAR Data. *Electronics*. 2018, vol. 7, no. 11. issn 2079-9292. Available from doi: 10.3390/electronics7110276.

4. Zhao, Runkai; Heng, Yuwen; Wang, Heng; Gao, Yuanda; Liu, Shilei; Yao, Changhao; Chen, Jiawen; Cai, Weidong. *Advancements in 3D Lane Detection Using LiDAR Point Clouds: From Data Collection to Model Development*. 2024. Available from arXiv: 2309.13596 [cs.CV].

5. Paek, Dong-Hee; Wijaya, Kevin Tirta; Kong, Seung-Hyun. *Row-wise LiDAR Lane Detection Network with Lane Correlation Refinement*. 2022. Available from arXiv: 2210.08745 [cs.CV].

6. Wilson, Benjamin; Qi, William; Agarwal, Tanmay; Lambert, John; Singh, Jagjeet; Khandelwal, Siddhesh; Pan, Bowen; Kumar, Ratnesh; Hartnett, Andrew; Pontes, Jhony Kaesemodel; Ramanan, Deva; Carr, Peter; Hays, James. Argoverse 2: Next Generation Datasets for Self-driving Perception and Forecasting. In: *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks (NeurIPS Datasets and Benchmarks 2021)*. 2021.

7. Balta, Haris; Velagic, Jasmin; Bosschaerts, Walter; De Cubber, Geert; Siciliano, Bruno. Fast Statistical Outlier Removal Based Method for Large 3D Point Clouds of Outdoor Environments. *IFAC-PapersOnLine*. 2018, vol. 51, no. 22, pp. 348–353. issn 2405-8963. Available from doi: https://doi.org/10.1016/j.ifacol.2018.11.566. 12th IFAC Symposium on Robot Control SYROCO 2018.

8. Blanco-Claraco, José Luis. A tutorial on SE(3) transformation parameterizations and on-manifold optimization. *CoRR*. 2021, vol. abs/2103.15980. Available from arXiv: 2103.15980.

9. Chen, Homer H; Huang, Thomas S. A survey of construction and manipulation of octrees. *Computer Vision, Graphics, and Image Processing*. 1988, vol. 43, no. 3, pp. 409–431. issn 0734-189X. Available from doi: https://doi.org/10.1016/0734-189X(88)90092-8.

10. Tiwari, Vijay. Developments in KD Tree and KNN Searches. *International Journal of Computer Applications*. 2023, vol. 185, pp. 17–23. Available from doi: 10.5120/ijca2023922879.

11. Ram, Anant; Sunita, Jalal; Jalal, Anand; Manoj, Kumar. A Density Based Algorithm for Discovering Density Varied Clusters in Large Spatial Databases. *International Journal of Computer Applications*. 2010, vol. 3. Available from DOI: 10.5120/739-1038.

12. Ruzgienė, Birutė; Förstner, Wolfgang. Ransac for outlier detection. *Geodesy and cartography*. 2012, vol. 31, pp. 83–87. Available from DOI: 10.3846/13921541.2005.9636670.

13. Greenacre, Michael; Groenen, Patrick; Hastie, Trevor; Iodice D'Enza, Alfonso; Markos, Angelos; Tuzhilina, Elena. Principal component analysis. *Nature Reviews Methods Primers*. 2022, vol. 2, p. 100. Available from DOI: 10.1038/s43586-022-00184-w.

14. Rusu, Radu Bogdan; Cousins, Steve. 3D is here: Point Cloud Library (PCL). In: *IEEE International Conference on Robotics and Automation (ICRA)*. Shanghai, China, 2011.

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

1. **LIDAR**: Light Detection and Ranging
2. **SOR**: Statistical Outlier Removal
3. **DBSCAN**: Density-Based Spatial Clustering of Applications with Noise
4. **RANSAC**: Random Sample Consensus
5. **PCA**: Principal Component Analysis
6. **PCL**: Point Cloud Library

A Attachments

An attachment in ZIP format that contains various files for an autonomous lane detection framework using LIDAR data. This ZIP file includes Python scripts, C++ source files, a Bash script for dataset downloading, and a CMake configuration file for compiling the C++ files.

A.1 Python Scripts

`dataloader.py`: Manages LIDAR data loading and transformations.

`io_utils.py`: Handles reading and processing of Feather format data.

`se3.py`: Implements SE3 transformations for 3D point clouds.

`accumulate.py`: Aggregates LIDAR sweeps into a comprehensive dataset.

`clustering.py`: Performs DBSCAN clustering on LIDAR data.

`ransac_filter.py`: Applies RANSAC algorithm for line fitting in point clouds.

`pca_filter.py`: Utilizes PCA for analyzing and filtering point cloud clusters.

`join.py`: Merges clusters to form continuous lane entities.

`classify.py`: Classifies lane entities based on cluster characteristics.

A.2 C++ Source Files

`preprocessing.cpp`: Applies preprocessing like noise filtering and downsampling.

`ground_extraction.cpp`: Extracts and projects ground plane from LIDAR data.

A.3 Bash Script

`dataset_download.sh`: Automates downloading of the Agroverse 2 dataset.

A.4 CMake Configuration File

`CMakeLists.txt`: Specifies build settings and dependencies for C++ files.