Ego Lane Line Detection using Point Cloud Data

Table of Contents

1. Introduction

Background: Lane line detection is a critical component in the development of autonomous vehicle systems. It involves identifying the boundaries of lanes on roadways, which is essential for the safe navigation of autonomous vehicles. With the advancement of sensor technology, especially LiDAR, the use of point cloud data for lane detection has become increasingly viable. Point cloud data offers a detailed and accurate representation of the vehicle's surrounding environment, making it ideal for lane line detection.

Problem Statement: The primary challenge addressed in this project is the detection and visualization of left and right ego lane lines from raw point cloud data. This task is complicated due to the inherent characteristics of point cloud data, including its high dimensionality and the presence of noise and outliers. Additionally, the variability in road conditions and lane markings further complicates this process. Effectively managing these challenges is crucial for achieving accurate lane line detection, which is a fundamental requirement for the development of reliable autonomous driving systems.

Objective: The goal of this project is to develop a robust lane line detection system that can process point cloud data to identify and visualize lane lines. This system is composed of four main components: `PointCloudProcessor` for initial data preprocessing, `DetectLaneLines` for lane line identification, `LaneLinesProcessor` for calculating polynomial representations of these lines, and `Vis` for the visualization of detected lane lines. By leveraging Python and key libraries such as NumPy, Open3D and scikit-learn the project aims to streamline the lane line detection process from raw data handling to the final visualization stage. The ultimate objective is to provide a clear and accurate representation of lane lines in various environments.

2. Tools and Languages Used:

Programming Language: Python 3.8.0

Main Libraries:

- numpy: For numerical computations and matrix operations.
- Open3d: Utilized for processing and visualizing point clouds
- matplotlib: For plotting and visualizing data and outputs.
- scikit-learn: Utilized for any components such as RANSAC and DBSCAN.
- threading (Python Standard Library): Employed for parallel execution of processes, enhancing the overall efficiency and responsiveness of the application.

- Pynput: Used for monitoring and controlling keyboard inputs, facilitating interactive features and user controls in the application

3. High-Level Overview of the System:

Our lane line detection system is structured around four key classes:

i. **`PointCloudProcessor:`** Processes point cloud data in .bin format, removing duplicates and outliers, and segmenting roads.

ii. **`DetectLaneLines:`** Identifies and outputs left and right lane line points from the preprocessed data received from `PointCloudProcessor` class.

iii. **`LaneLinesProcessor:`** Fits third-degree polynomials to the lane line points detected by the `DetectLaneLines` class and stores the coefficients in a .txt file.

iv. **`Vis:`** Visualizes Lane lines on point clouds using coefficients from the `LaneLinesProcessor` class.

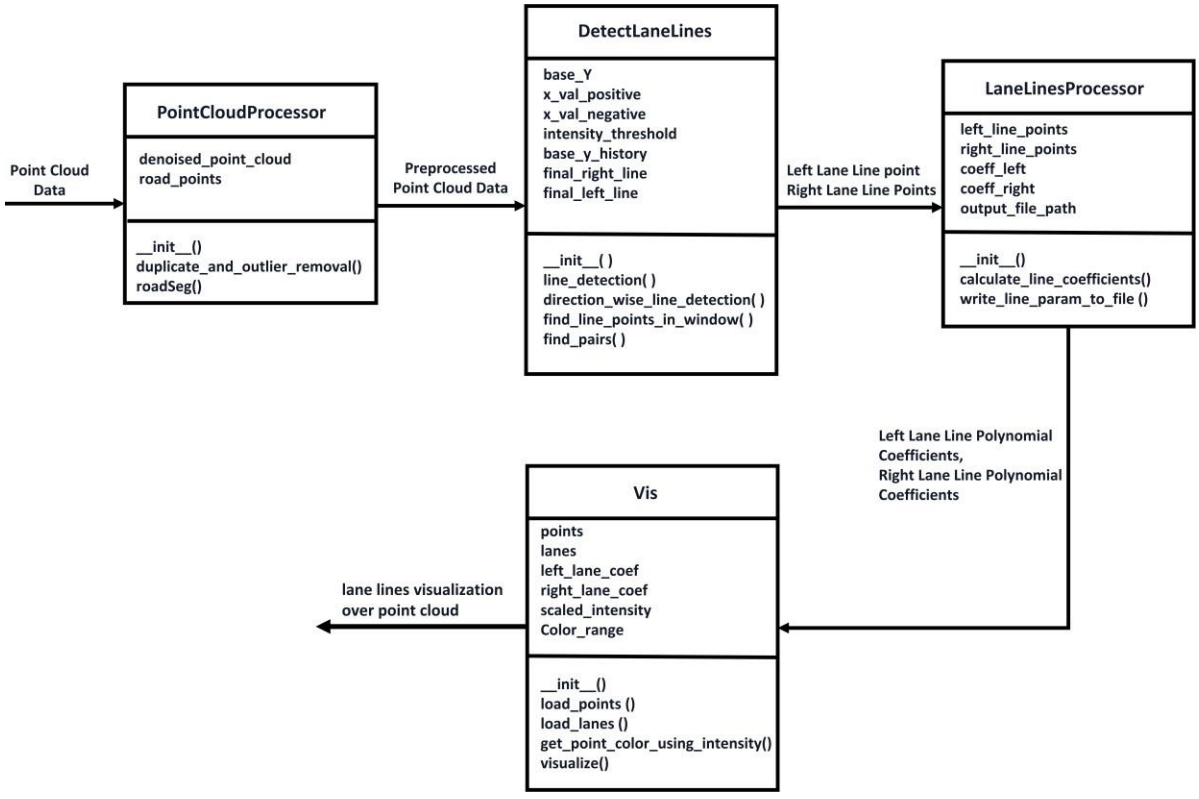

Figure: Class diagram of lane line detection system

4. Detailed Descriptions of Classes

This section presents detailed descriptions of each class and its methods:

4.1 `PointCloudProcessor`

The `PointCloudProcessor` class is designed to preprocess point cloud data. Its primary functions include removing duplicate points, filtering out noise and outliers, and identifying the main plane of the road in the point cloud. This preprocessing step is crucial for the efficiency of the lane lines detection algorithm.

- **Input:** A string indicating the path to the binary file containing the point cloud data.
- **Output:** A NumPy array of processed point cloud data, specifically containing points that represent the road surface.

Key Methods in `PointCloudProcessor`:

PointCloudProcessor class consists of a constructor `__init__()` and two other methods namely, `duplicate_and_outlier_removal()` and `roadSeg()`. Details of these methods are as follow:

i. `__init__(self, bin_path)`

Input:

- Path to the binary file containing the point cloud data.

Flow:

- Reads the point cloud data from the binary file as a 2D numpy array. Each row represents a point, with the first three columns for x, y, z coordinates, the fourth for intensity, and the fifth indicating the lidar beam used.
- Normalizes the intensity values in the range [0, 1].

Output:

- An instance of `PointCloudProcessor` class initialized with the point cloud data for further processing

ii. `duplicate_and_outlier_removal()`**:**

Input:

- None; uses `self.Point_cloud`

Flow:

- Removes duplicate points from the point cloud.
- Applies statistical outlier removal.
- Calls `roadSeg()` with the `denoised_point_cloud`.

Output:

- A cleaned point cloud of road points, without noise and outliers.

Description: This method removes duplicate points using numpy's `unique()` function. Moreover it employs Open3D's `remove_statistical_outlier()` function to remove outliers from the point cloud data.

iii. `roadSeg( )`**:**

Input:

- `denoised_point_cloud`: Point cloud data after outlier removal.

Flow:

- Segments the main plane using `RANSAC` (Random Sample Consensus).
- Applies `DBSCAN` (Density-Based Spatial Clustering of Applications with Noise) clustering to segment the road.

Output:

- Identifies and stores road points in `self.road_points`.

Description: This method applies a combination of `RANSAC` and `DBSCAN` algorithms. `RANSAC` is used for plane segmentation to identify the road surface, and `DBSCAN` is employed to cluster the inlier points. The output is a refined set of points that represent the road.

4.2 `DetectLaneLines`

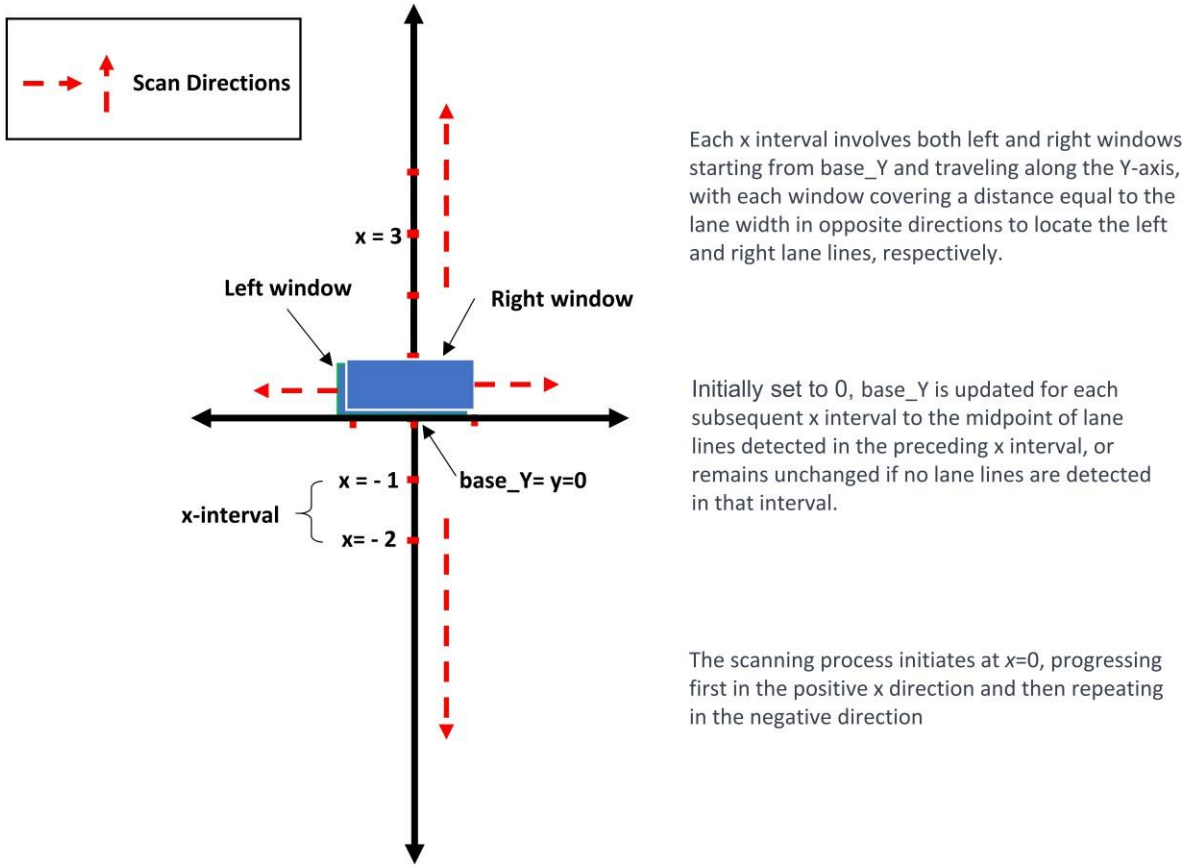

Figure 2: Overview of the sliding window scanning method

The `DetectLaneLines` class is integral to the lane detection process. It specializes in identifying potential left and right lane lines from the point cloud data. The class employs a window-sliding technique along the road's width and length to detect lane points, eventually compiling these points into distinct left and right lane lines.

- **Input:** `point_cloud`(i.e. road_points): A NumPy array representing the preprocessed point cloud data, which is used to detect lane lines.
- **Output:** `final_left_line` and `final_right_line`: Lists of tuples representing the coordinates of the detected left and right lane lines, respectively.

Key Methods in `DetectLaneLines`:

i. `line_detection ()`:

Input:

- None directly operates on the `point_cloud` instance variable.

Flow:

- Splits the `point_cloud` into segments based on positive and negative x values.
- For each segment, calls `direction_wise_line_detection()`.
- Combines the lane line points detected in both segments to form complete lane lines.

Output:

- Final lists of detected left and right lane lines across the entire range of the `point_cloud`.

Description: This method serves as the central coordinator of the lane line detection operation. It begins by dividing the point cloud into two segments based on x-coordinate values—positive and negative. For each segment, the method invokes `direction_wise_line_detection()` to identify lane line points. Post-processing by `direction_wise_line_detection()`, it collates the line points for both positive and negative x-values to form the final left and right lane lines, which are the outputs of the DetectLaneLines class.

ii. `direction_wise_line_detection ()`:

Input:

- `base_Y`: The starting y-axis position of scanning windows in each x-axis interval/segment[1].
- `x_val`: Array of x-axis intervals for vertical window sliding.
- `base_Y_history`: List for recording updates to `base_Y` during processing.

Flow:

- Iterates through each x-axis interval in `x_val`.
- For each x-interval, conducts a y-axis sliding window scan to identify potential line points.
- Uses `find_line_points_in_window()` and `find_pairs()` to detect actual lane line points.
- Updates `base_Y` based on identified points and records in `base_Y_history`.

Output:

- Lists of final right and left lane line points for each x-axis interval.
- Updated `base_Y_history`

Description: This method plays a pivotal role in scanning the segmented point cloud (positive or negative x direction) to identify potential lane line points. It leverages two critical functions: `find_line_points_in_window` and `find_pairs`.

[1] To facilitate lengthwise scanning, the x-axis is segmented according to the desired height of the scanning window, with each segment's height matching that of the window.

iii. `find_line_points_in_window ()`

Input:

- `lowerX, upperX`: x-axis boundaries of the scanning window.
- `LowerY, UpperY`: y-axis boundaries of the scanning window.
- `intensity_threshold`: Threshold for considering a point as a potential line point.

Flow:

- Filters `point_cloud` to find points within the specified window boundaries.
- Further filters these points based on the `intensity_threshold`.
- Calculates the average X and Y coordinates of the filtered points.

Output:

- The x and y coordinates of the representative point of the potential line points within the window (the representative point is acquired by averaging x and y coordinates of the detected lane line points) , or (`None, None`) if no suitable points are found.

Description: The function employs two sliding windows that move along the Y-axis at each x-coordinate value(`x_val`). Starting from `base_Y`, which represents the estimated position of the ego vehicle on the y-axis at that x-coordinate, these windows move in opposite directions along the y-axis to detect potential line points for the left and right lane lines. If `base_Y` is 0, for example, one window slides from 0 to `+lanewidth` (right side), and the other from 0 to `−lanewidth` (left side). This systematic approach ensures that potential line points are captured across the road's width for every x-coordinate value. The `base_Y` value is a critical element in the lane line detection process, dictating the starting point for the dual sliding windows along the y-axis at each x-coordinate interval. Initially, for the first x interval, `base_Y` is set to the origin, representing the initial position of the ego vehicle along the y-axis, which is typically 0. This initial `base_Y` is utilized to detect lane lines within the current x interval. As the process advances to subsequent x intervals, the method recalibrates `base_Y` based on the line detection results from the preceding interval. Specifically, `base_Y` for the next x interval (`x[i+1]`) is set to the midpoint of the y values of the identified left and right lane line points in the current interval (`x[i]`). This dynamic updating of `base_Y` ensures that the search for potential lane points is continuously centered around the most recent and relevant area, enhancing the precision and reliability of line point detection across the point cloud.

iv. `find_pairs()`

Input:

- `potential_line_XY_left`: List of potential left line points (x, y coordinates).
- `potential_line_XY_Right`: List of potential right line points (x, y coordinates).
- `right_line`: Current list of confirmed right line points.
- `left_line`: Current list of confirmed left line points.
- `base_Y`: Current Y-axis base value for lane line detection.

Flow:

- Forms all possible pairs from points in `potential_line_XY_left` and `potential_line_XY_Right` such that each pair has one member point from `potential_line_XY_left` and the other from `potential_line_XY_Right` representing left and right line points respectively.
- Evaluates each pair, comparing the distance between its members to the lane width.
- Selects pairs where this distance approximates the lane width, identifying them as corresponding line points for that X interval.
- Adds these selected pairs to the `right_line` and `left_line` lists as confirmed line points.

Output:

- Updated `right_line` and `left_line` lists

i. **Description:** Once potential line points are identified by `direction_wise_line_detection()`, `find_pairs()` is employed to validate and pair these points. The underlying principle is that at any x-coordinate, valid left and right line points must be approximately a lane's width apart, a criterion unlikely to be met by random noise. The function selects pairs that conform to this spacing rule, thereby determining the actual line points for both the left and right lines at each X-value.

4.3 **LaneLinesProcessor**

The `LaneLinesProcessor` class calculates and stores the coefficients of polynomial curves that best fit the left and right lane lines. These coefficients are crucial for reconstructing and visualizing the lane lines in the point cloud data.

Input:

- `left_line_points`: A list or NumPy array of points (coordinates) representing the left lane line.
- `right_line_points`: A list or NumPy array of points (coordinates) representing the right lane line.

Output:

- `coeff_left`: Polynomial coefficients of the fitted curve for the left lane line.
- `coeff_right`: Polynomial coefficients of the fitted curve for the right lane line.

Key Methods in `LaneLinesProcessor`:

i. `calculate_line_coefficients()`:

Input:

- No external input; uses `self.left_line_points` and `self.right_line_points`.

Flow:

- Calculates polynomial coefficients (of degree 3) for the left line using `np.polyfit` on `self.left_line_points`.
- Similarly calculates coefficients for the right line using `self.right_line_points`.

Output:

- Updates the instance variables `self.coeff_left` and `self.coeff_right` with the calculated polynomial coefficients for the left and right lines, respectively.

Description: Calculates the polynomial coefficients for the left and right line lines using NumPy's `polyfit()` function. It fits a third-degree (cubic) polynomial to the line points, providing a mathematical model for the line's curvature.

ii. `write_line_param_to_file()`:

Input:

- `left_line_params`: Polynomial coefficients of the left lane line.
- `right_line_params`: Polynomial coefficients of the right lane line.
- `output_file_path`: File path where the line parameters will be written.

Flow:

- Formats the `left_line_params` and `right_line_params` into semicolon-separated strings.
- Opens the file specified by `output_file_path` and writes the formatted strings for the left and right lines.

Output:

- Writes the left and right line parameters to a file at `output_file_path`. No return value from the function.

Description: A static method that formats the calculated polynomial coefficients into a string and writes them to a specified file. This function is crucial for documenting and storing the lane line data for later visualization or analysis.

4.4 `Vis`

The `Vis` class is designed for visualizing line detection results. It reads and processes point cloud data and corresponding lane line information, then renders these in a graphical interface. The class's primary function is to provide an interactive visualization of both the raw point cloud data and the detected lanes lines, helping to illustrate and validate the lane line detection process.

Input:

- `data_folder`: Directory path containing the point cloud data.
- `lane_folder`: Directory path containing the lane data.

Output:

- Interactive visualization of point clouds and lane lines.

Key Methods in `Vis`:

`load_points()`:

- Loads and processes the point cloud data from the current file, including applying color based on intensity.

`load_lanes()`:

- Reads the lane data and generates line sets for visualization based on polynomial coefficients.

`read_data()`:

- Retrieves and organizes the file paths for the point cloud and lane data.

`visualize()`:

- Initializes and controls the main visualization window, setting up the scene and rendering the data.

`update_geometry_from_input()`:

- Implements keyboard interaction for navigating through the data, updating the visualization accordingly.

`update_points and update_lanes()`:

- Methods to update the point cloud and lane lines in the visualization based on the current index.

`set_current_file():`

- Sets the current file for visualization based on a specified filename.

5. Conclusion

This implementation of lane line detection system has shown potential in lane line detection in many of the given scenarios, yet faces challenges with missing lane lines over extended distances and complex lane markings. For missing lane lines, we can utilize `base_Y_history` values and the rate of directional change, along with available partial lane line information if any. This method leverages the existing trajectory and directional trends to extrapolate the positions of missing lane lines, particularly when only one side of the lane is visible. This predictive modeling can help bridge gaps in data and maintain continuous lane detection. For lane markings we can implement an improved strategy within the `findpair()` function. When multiple pairs meet the lane width criteria — a common occurrence with aligned adjacent lane markings — a specialized scanning window can be activated. This window will assess these pairs along x direction, analyzing the length and shape of the detected lines or markings. By characterizing these features, the system can more accurately differentiate between actual lane lines and mere lane markings. Furthermore, orientation determination can be carried out using a circular sliding window that can be passed over some lidar scan rings to find high intensity lane line points and then utilize those points to determine orientation.