Developing, Analyzing, and Evaluating Self-Drive Algorithms Using Drive-by-Wire Electric Vehicles

Beñat Froemming-Aldanondo[1,*], Tatiana Rastoskueva[2], Michael Evans[3], Marcial Machado[4], Anna Vadella[5], Rickey Johnson[6], Luis Escamilla[7], Milan Jostes[8], Devson Butani[8], Ryan Kaddis[8], Chan-Jin Chung[8], and Joshua Siegel[9]

Abstract— **Reliable lane-following algorithms are essential for safe and effective autonomous driving. This project was primarily focused on developing and evaluating different lane-following programs to find the most reliable algorithm for a Vehicle to Everything (V2X) project. The algorithms were first tested on a simulator and then with real vehicles equipped with a drive-by-wire system using ROS (Robot Operating System). Their performance was assessed through reliability, comfort, speed, and adaptability metrics. The results show that the two most reliable approaches detect both lane lines and use unsupervised learning to separate them. These approaches proved to be robust in various driving scenarios, making them suitable candidates for integration into the V2X project.**

I. INTRODUCTION

In this paper, five-lane detection approaches are presented that were proven to work on real drive-by-wire vehicles. This project aimed to identify the most reliable one for use in a Vehicle-to-Everything (V2X) project [1], which focused on developing a low-cost Roadside Unit (RSU) for Adaptive Intersection Control of Autonomous Electric Vehicles. V2X is a communication technology that enables vehicles to interact with each other (V2V), with infrastructure (V2I), with pedestrians (V2N), and with other road users. It aims to improve road safety and traffic efficiency by providing real-time information that facilitates proactive responses to different driving scenarios [2].

To have a successful V2X communication system, autonomous vehicles must have some basic and essential components such as effective lane-following, which requires precise detection and tracking of lane markings to maintain a proper positioning on the road. The 5 algorithms presented in this paper are the following: Largest White Contour, Lane Line Approximation using Least Square Regression, Linear Lane Search with K-Means, Lane Line Discrimination using DBSCAN, and DeepLSD Lane Detection. Their performance was compared based on reliability, comfort, speed, and adaptability. Reliability was measured by the consistency of successful laps, comfort by the smoothness of the vehicle's movements, speed by lap time, and adaptability by the algorithms' performance under varying conditions. This paper also details lane following, the architecture used, real-life implementation challenges, and presents an experimental analysis to identify the best algorithm for the V2X project.

The remainder of this paper is organized as follows: Section II reviews related work; Section III outlines the methodology and resources; Section IV describes the software architecture with four nodes; Section V covers preprocessing; Section VI details five lane detection algorithms; Section VII discusses vehicle control algorithms; Section VIII presents experimental results; and Section IX concludes with future research directions. The project code is available at https://github.com/benatfroemming/REU-2024-Lane-Following.

II. RELATED WORK

Due to the complexities and cost of using real-scale autonomous vehicles, most research on lane-following has not been tested outside of a virtual environment. Much of this research does not effectively simulate lane following, and instead only focuses on lane detection using a single image or video. Hence, there is no reliable way to prove the efficiency and safety of those algorithms. The performance of an algorithm can be completely different in a real environment where there are a lot more factors to consider, such as the kinematics of the vehicle.

In the previous editions of the Self-Drive Research Experience for Undergraduates (REU) funded by the National Science Foundation (NSF) at Lawrence Technological University (LTU), several algorithms were developed and tested on real electric vehicles [3]. Traditional computer vision techniques were used for lane detection using OpenCV. This paper builds on those initial approaches, showing continued progress by introducing machine learning techniques.

III. MATERIAL AND METHODS

A. Simulation and Real Environment

The lane-following algorithms were tested on two different simulators: Simple-Sim [4] and Gazelle-Sim. Simple-Sim is a 2D simulator that allows for the control of a single robot in custom environments using ROS. Gazelle-Sim is an expanded version of the original, allowing for the simultaneous operation of multiple robots.

*Corresponding author: froem076@umn.edu

[1]University of Minnesota; [2]The University of Arizona; [3]Old Dominion University; [4]The Ohio State University; [5]Butler University; [6]North Carolina A&T State University; [7]New Mexico State University; [8]Lawrence Technological University; [9]DeepTech Lab, Michigan State University.

This project is supported by the National Science Foundation REU Site Awards #2150096 and #2150292. The electric vehicles used for this research are sponsored by MOBIS, US Army GVSC, NDIA, DENSO, SoarTech, Realtime Technologies, Veoneer, Dataspeed, GLS&T, and LTU.

After testing the algorithms within the simulator, they were then modified to work on real-life drive-by-wire vehicles. The real-world test course is in parking lot H, located at Lawrence Technological University in Southfield, Michigan, USA. This circular test course simulates real-world adverse conditions such as potholes, sharp curves, faded and narrow lane markings, cracks, and extraneous lines. It also introduces unpredictable external challenges like tree shadows, sun glare, and puddles. An aerial view of the course is shown in Fig. 1.

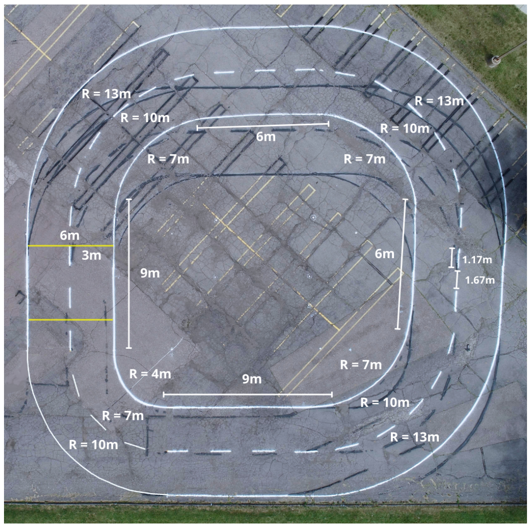


Fig. 1: Aerial View of the Lot H Course in LTU.

B. Vehicle Specifications

The vehicles used for testing were ACTors (**A**utonomous **C**ampus **T**ranspo**r**t) 1 and 2. Refer to Fig. 2. These vehicles were built on the Polaris Gem e2 platform, provided by MOBIS, and then modified by the Lawrence Technological University Intelligent Ground Vehicle Competition (IGVC) team. Each ACTor is equipped with essential self-driving hardware, including a Dataspeed Drive-by-Wire kit, an HDR camera for lane-following, 2D and 3D LiDAR sensors, and two Swift Piksi GPS modules. Additionally, both vehicles feature a Netgear router, power inverter, and a removable computer for networking and programming the Drive-by-Wire system using ROS (Robot Operating System). The Polaris Gem e2 boasts a top speed of 25 miles per hour and a range of approximately 30 miles.

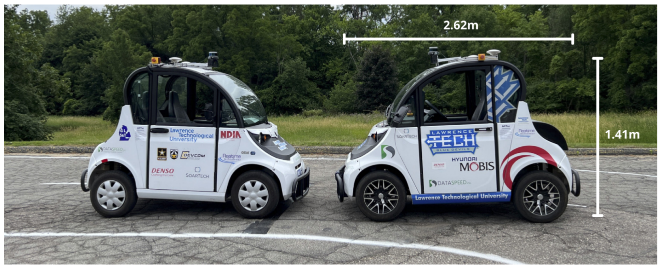


Fig. 2: ACTors 1 and 2.

IV. Architecture and Dynamic Reconfigure

All the lane-following algorithms share the same architecture to maintain simplicity and modularity. It was designed to facilitate easy switching and tuning of the algorithms and environments within the vehicle or simulator. This is accomplished by using arguments within the launch file and dynamic reconfigure. The dynamic reconfigure GUI allows the user to activate the vehicle, adjust the vehicle's speed, tune the algorithms' parameters, and modify the steering sensitivity. The ROS-based architecture consists of 4 nodes: Preprocessor, Lane Detector, Vehicle Controller, and Vehicle Node.

V. Preprocessing

The Preprocessor node is in charge of enhancing the raw frontal image of the vehicle to extract insightful data about the road markers. It does this by applying a series of different effects using the OpenCV library. The first of these effects is Median Blur, which is applied to remove noise while preserving edges. This creates a blur effect in the image and highlights white regions. The image is then converted from the RGB color space, which has three channels, to a single-dimensional color space of gray shades. The new image retains the white pixels and makes it easier to find the correct threshold for identifying them. After that, the white regions are masked out. Finally, by cropping and removing the top part of the image, the region of interest (ROI) is obtained. The ROI only looks at the road and avoids other extraneous noises like buildings, trees, and the sky.

These steps are common to all algorithms, although some may require more sophisticated processing as explained in the next section. They can be disabled or tuned using Dynamic Reconfigure. For example, the upper and lower white threshold values when creating the mask. Finally, the lane-detection node receives the modified image and follows a specific set of steps tailored to each algorithm.

VI. Lane Detection Algorithms

1) Largest White Contour: The first approach used for lane detection involves a basic line-following algorithm with an offset and is used to teach and test the basic concepts behind using the ACTor Vehicles [5]. Hence, it is a good starting point to add improvements. The basic algorithm looks for the largest contiguous collection of white pixels, or "contour", in the preprocessed image. OpenCV is used to obtain a list of all white contours by calling the find contours function. Then, by iterating through them and computing their areas, the largest one can be identified. Next, the spatial moments are computed to find the largest contour's centroid. Finally, a static offset is added to the centroid in the x-axis to approximate the center of the lane as seen in Fig. 3 (on the left of the line if driving in the right lane).

While this approach is reliable in ideal conditions, it encounters challenges on the real course. Cracks and poor lane markers can disrupt the largest contour, leading to incorrect marker identification. To address this, in the improved version, dilation is applied to enlarge white object areas, making them more pronounced. Applying Gaussian Blur also helps connect broken lines. Additionally, on the enhanced version, only contours on the right side of the image are considered to avoid detecting the left line. The downside of this approach is that it assumes a continuous white line exists on the right side of the lane, which may not always be the case in real-world scenarios. Moreover, the paper later demonstrates that following a single line from a lane is less reliable compared to tracking both lines.

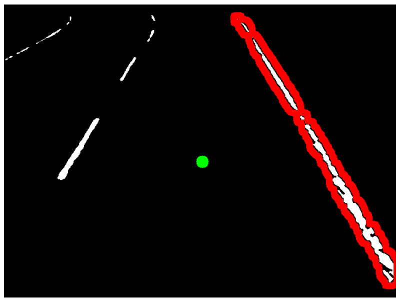


Fig. 3: Largest Contour and Offset Point.

2) Lane Line Approximation using Least Square Regression (LSRL): The second lane detection approach identifies both lane lines and computes approximate least squares regression lines for each. To improve line detection, particularly for curves, it takes advantage of a bird's-eye view perspective [6]. This transformation makes the image appear as if it were taken from above, causing the lines to appear parallel rather than converging due to depth. Refer to Fig. 4. Additionally, curves look straighter but slightly inclined, allowing the use of a first-degree polynomial to represent the lines.

After applying this perspective transform, the preprocessing requires additional steps that were not used in the first approach. Canny filtering is used on the white mask image to remove noise and only keep edges. The edges are found from gradient changes in pixels, non-maximum suppression, and thresholding [7]. The next step involves the Hough Lines Transform, a feature extraction technique used to detect straight lines [8]. All the straight edges of the Canny edges are extracted and stored as a list of start and end points. The points are then filtered by the minimum length and the slope of the corresponding lines. In most cases, it is safe to remove lines that are nearly horizontal as they are likely not part of the lane lines. If not enough points are obtained this way, another option is to draw the filtered lines in white on a black canvas, extract white pixels, and downsample them uniformly. The image is then divided into two sections: left and right. Points on the left part are classified as part of the left lane line, while points on the right side are classified as part of the right lane line. For each side, regression lines are computed, which are subsequently used to determine the lane's midpoint. The least squares regression line is given by $y = mx + b$ where the slope m and y-intercept b are calculated as seen in (1).

$$\begin{aligned} m &= \frac{n\sum xy - \sum x \sum y}{n\sum x^2 - (\sum x)^2} && \text{(Slope)} \\ b &= \frac{\sum y - m\sum x}{n} && \text{(Intercept)} \end{aligned} \tag{1}$$

To enhance reliability, additional methods are used. One method reduces noise by live cropping on curves, which minimizes white points along the lane edges for better detection. Another method uses an adaptive variable to divide the image, distinguishing between left and right lines during turns. It's useful when the lines are not strictly positioned to the left and right of the horizontal midpoint. Furthermore, if a line is not detected, the algorithm assumes the line is at the image's edge.

This algorithm can detect both lane lines simultaneously, providing a more accurate lane representation than single-line algorithms. It performs well on curves, even if one line isn't detected. However, the least squares regression can be sensitive to outliers, and despite noise reduction, residual points might still cause deviations from the lane.

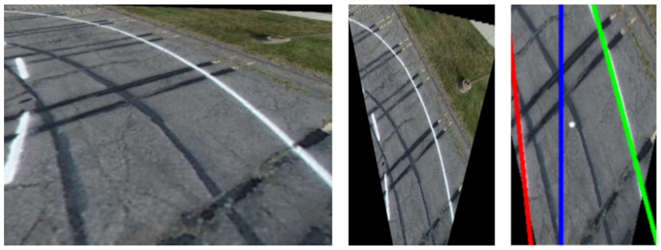


Fig. 4: Visualization of LSRL Lane Detection Algorithm.

3) Linear Lane Search with K-Means: The third approach focuses on finding lines using a single horizontal line of the image. In particular, it focuses on the central horizontal row only, finding all of the white pixels. In the most ideal case, two groups of white pixels are detected, one for each lane line. Instead of finding the mean of the detected points, it is a better approximation to apply K-Means with K = 2 to find the centroid of each group, and then find an average. K-means clustering initializes K cluster centers (centroids) and iteratively assigns each data point to the nearest centroid [9]. The centroids are updated to the mean of the assigned points, and this process repeats until the centroids converge and no longer change significantly. This method is implemented using the Scikit-Learn Python library.

The approach works well when both lane lines are continuous. However, it struggles when the middle line is discontinuous, potentially detecting only one or no lines. This issue can be identified by the proximity and size of the K-Means centroids, which should be close together or have a small number of cluster points. To address this, the two most recent centroids are stored as an approximation of lane positions. Refer to Fig. 5.

The algorithm is computationally fast, enabling quick processing speeds and rapid reactions. It also provides stability, as defects in individual images do not cause immediate incorrect responses. However, the main issue is extraneous white noise in the searched row, which can affect K-Means clustering results. This can be mitigated by using noise reduction techniques like histograms and thresholds.

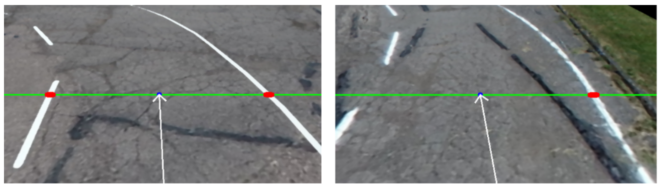


Fig. 5: Visualization of Linear Lane Search with K-Means.

4) Lane Line Discrimination using DBSCAN: The fourth approach developed for lane detection uses an algorithm called DBSCAN to identify and distinguish both lane lines. Similar to the LSRL algorithm, it obtains a list of points through Canny edge detection and Hough Lines Transform.

In the next step, an unsupervised learning method is employed to filter out noise and differentiate between the lane lines. Specifically, DBSCAN (Density-Based Spatial Clustering of Applications with Noise) is used, an algorithm also available in the Scikit-Learn Python library. DBSCAN clusters points that are densely packed together while identifying points in low-density regions as outliers. The algorithm selects a point and searches for other points within a specified radius, epsilon. If the number of points within this area exceeds a certain threshold (minPts), the point is classified as a ”core point,” and a new cluster is formed. Refer to Fig. 6. This process is then repeated to expand the clusters [10]. Density-based algorithms are effective for lane line detection because they can handle well-separated lines as seen in Fig. 7. They can also connect discontinuous segments of the center line, provided the epsilon value is appropriately set to avoid exceeding the minimum distance between lane lines.

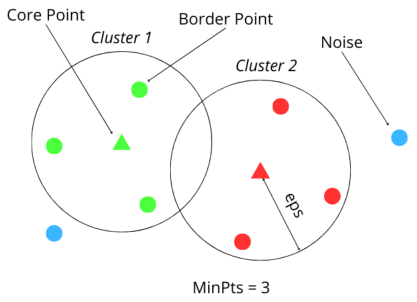


Fig. 6: DBSCAN Algorithm.

Once all of the clusters are computed, the focus is shifted to the two clusters directly in front of the vehicle. The algorithm sorts the clusters according to their y-axis values, where the y-axis increases from top to bottom in the image. It then identifies the two largest clusters nearest to the bottom of the image and checks if they are substantial enough to be lane lines by ensuring they contain a sufficient number of points. Subsequently, the centroids of these two clusters are computed and averaged to determine the lane's center. Averaging all the unclustered points was avoided because a line with more points could disproportionately influence the centroid. Therefore, clustering is essential for accurate lane detection.

The effectiveness of clustering in this approach depends on choosing the right epsilon value. In a raw frontal image, increasing distance can make lane lines appear closer, potentially merging them into one cluster if epsilon is too large. Transforming the image to a bird's-eye view, as used in the LSRL algorithm, and adjusting epsilon by extending shorter Hough lines can help, while overextension risks overlap on curves. Cropping the image's top can also reduce depth issues. DBSCAN separates lane lines more effectively than simple vertical splits or slope-based methods, which struggle with curved lanes.

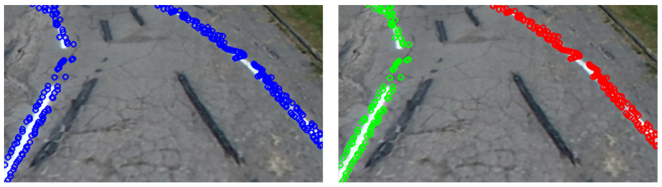


Fig. 7: DBSCAN Clustering Lane Lines.

5) DeepLSD Lane Detection: The final approach developed is based on supervised learning. The rise of deep learning has significantly impacted the vehicle industry, enabling the automation of various tasks [11]. Instead of building a deep learning model from scratch, which is a difficult task without the proper resources, we utilized a pre-trained model called DeepLSD. By combining deep learning with the precision of handcrafted detectors, it excels at extracting line segments from real-world images [12]. Using such models can streamline the lane line detection process compared to traditional computer vision algorithms.

DeepLSD performed well under ideal conditions but struggled with identifying lines on the test course. The model detected noise, such as cracks and potholes, and had difficulty with curved lines. Applying masking and blurring techniques from previous algorithms improved white-line identification. To address curve detection, horizontal lines were drawn into the image, converting the curved lines into short straight segments for better analysis. An example is shown in Fig. 8.

Once the lane lines are detected, they are filtered by length and slope, and one of the previous approaches can be used to separate the lanes, such as DBSCAN. When testing this system on the vehicles' laptops, inference took approximately 0.15 seconds on average. The device used for testing was an MSI Gaming Laptop with an Intel 8-Core i7-11800H processor, 16GB of RAM, a 512GB SSD, and a GeForce RTX 3050 Ti 4GB graphics card. Since the camera within the vehicle publishes images quicker than the model can process, some images are ignored, and the processing happens less frequently. Additionally, the inference is run in parallel to allow the publishing of motion commands to the vehicle at a consistent 50 Hz, which is necessary for the DBW system's heartbeat. If the vehicle does not receive this heartbeat signal, the system shuts down because it assumes unsafe driving conditions. The coordinates of the centroid between the lane lines, determined through inference, are stored globally and updated once the model finishes processing. Meanwhile, these coordinates are published to the vehicle during each callback.

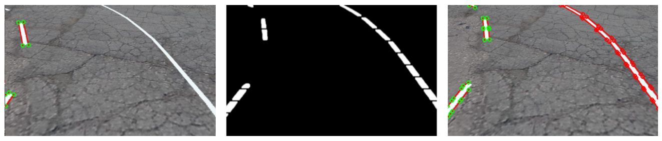
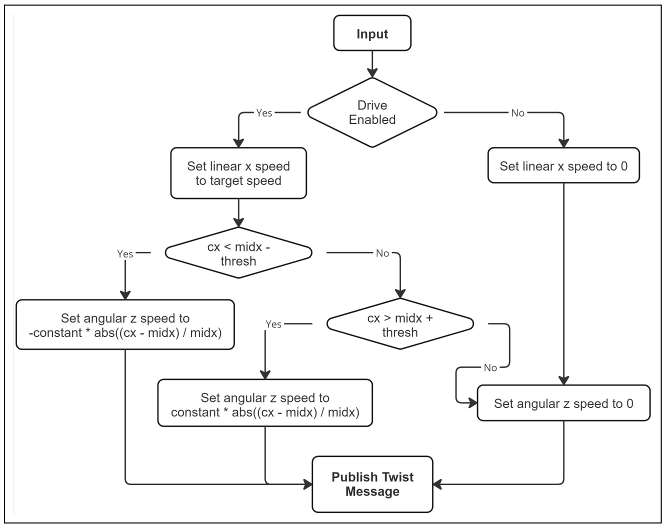


Fig. 8: Curve Detection Using DeepLSD.

VII. Lane Following Algorithms

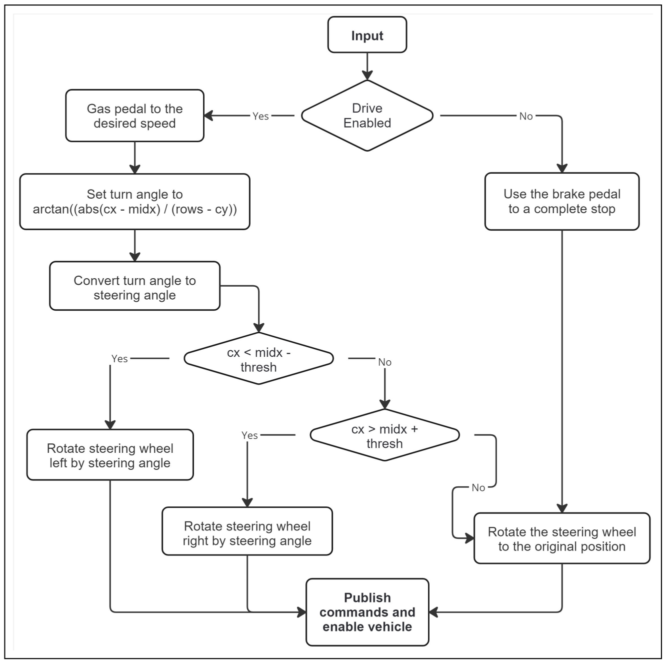
After detecting the lane lines, the next step is to center the vehicle within the lanes and translate this information into motion. This is done in the vehicle controller node. In this section, two different methods are presented for lane following. The first approach uses ROS Twist messages, which control the vehicle's motion with linear speed along the x-axis (measured in meters per second) and angular velocity, or yaw rate, along the z-axis (measured in radians per second). These values are published to the vehicle's vel_cmd topic. As an input, the algorithm only needs the offset between the center of the image (denoted as midx), and the center of the lane (denoted as cx) computed by the lane detection algorithms.


Input
Drive Enabled
Yes
No
Set linear x speed to target speed
Set linear x speed to 0
cx < midx - thresh
Yes
No
Set angular z speed to -constant * abs((cx - midx) / midx)
cx > midx + thresh
Yes
No
Set angular z speed to constant * abs((cx - midx) / midx)
Set angular z speed to 0
Publish Twist Message
Fig. 9: Lane Centering Using Yaw Rate.

While this approach works well at slower speeds, it fails when going faster due to the difficulty of establishing a proportional relationship between angular speed, linear speed, and cx. Yaw rate control, due to its role in rotational dynamics, is less intuitive and leads to difficulties in keeping the vehicle centered, resulting in an uncomfortable experience for the passenger.

To address these issues, an alternative control method is introduced for the Ackermann steering vehicle [13]. Instead of relying on yaw rate, this method controls the vehicle's steering and pedals directly, providing a more intuitive control experience. The Dataspeed drive-by-wire system in the ACTor vehicles uses custom ROS messages for this purpose. Actuator Messages (SteeringCmd) handle steering, while Unified Control Messages (UlcCmd) manage the brake and gas pedals.

A key improvement with this method is incorporating the y-offset (denoted as cy) of the computed lane center. Along with cx, midx, and the image height, a turning angle relative to the y-axis is calculated. This turning angle is then converted into a steering angle and transmitted via the SteeringCmd message. Before any movement is initiated, an enable message is sent to activate the vehicle's motion. These updates contribute to smoother driving and better lane centering.


Input
Drive Enabled
Yes
No
Gas pedal to the desired speed
Set turn angle to arctan((abs(cx - midx) / (rows - cy))
Use the brake pedal to a complete stop
Convert turn angle to steering angle
cx < midx - thresh
Yes
No
Rotate steering wheel left by steering angle
cx > midx + thresh
Yes
No
Rotate steering wheel right by steering angle
Rotate the steering wheel to the original position
Publish commands and enable vehicle
Fig. 10: Lane Centering with DBW Native Commands.

VIII. Experiment and Results

The performance of the five algorithms was evaluated on the Lot H course. The objective was to complete five consecutive laps on both the inner and outer lanes, driving on the right side. DBW Native Commands were used for lane following. In the outer lane, which is 97.54 meters long, we set a fixed speed of 2m/s, while in the inner lane of 78.67 meters, the speed was set to 1.5m/s. Across all algorithms, these target speeds were identified as being the most reliable and comfortable options during the testing phases. If an algorithm failed to successfully complete all five laps, it was repeated with a reduced speed if necessary. The experiment was conducted over several days, with weather conditions ranging from sunny to cloudy. This variability demonstrated each algorithm's adaptability to different environmental conditions.

All five algorithms were able to complete the set goal. The average speed and time showed minimal variation since the set speed was consistent for the algorithms. Therefore, the number of attempts is a more indicative measure of reliability. The only algorithm that successfully completed all 10 laps on the first attempt was the DBSCAN-based approach. In the inner lane, there is a sharp turn with a radius of just 4 meters where the lane-following algorithms fail more frequently.

Type	DeepLSD	DBSCAN	K-Means	LSRL	Largest Contour
Inner	5	1	2	2	2
Outer	1	1	1	5	3
Total	6	2	3	7	5

TABLE I: Attempts Needed by Each Algorithm.

As speed increased, issues with jerkiness and acceleration occurred during turns. To address this, we measured linear and angular momentum using GPS and an Inertial Measurement Unit (IMU). The IMU, equipped with accelerometers and gyroscopes, provides crucial data on the vehicle's acceleration and rotational rate, essential for navigation and stability control. We focused on the angular-z component, indicating angular momentum. During test runs, GPS coordinates (latitude and longitude) were recorded using Rosbags, with 30-second segments—approximately one lap—extracted for analysis. Linear momentum was calculated by determining the distance between consecutive GPS points using the Haversine formula [14]. Velocities were then computed by dividing these distances by the time differences between timestamps, as shown in Equation (2), and accelerations, as seen in Fig. 11, were obtained by differentiating velocity with respect to time, as seen in Equation (3).

$$v_i = \frac{d_i}{t_i - t_{i-1}} \quad (2) \qquad a_i = \frac{v_{i+1} - v_i}{t_{i+1} - t_i} \quad (3)$$

The results seen in Fig. 11 show that the K-Means and DBSCAN-based lane detection algorithms are the best at keeping a steady speed. Sharp turns, inclined slopes, potholes, and the algorithms themselves can cause variations from the target speed. The other three algorithms show more instability with larger spikes in their momentum.

In Fig. 12, the angular momentum plots show the completion of a lap with four turns. This can be seen from the four peaks. Smoother peaks indicate better comfort, as seen in the DBSCAN plot. Low peaks under zero indicate that the algorithm overcorrected and had to recover back, as seen in the DeepLSD plot, likely due to computational latency. In the LSRL plot, the IMU readings show sharp spikes at various points along each curve, followed by extended periods of remaining at zero. This may be due to the algorithm's inability to anticipate turns effectively, caused by its reliance on a bird's eye view. This view might not capture the road's curvature accurately, leading the vehicle to go straight more often and make sharper, more abrupt corrections, even on slight curves.

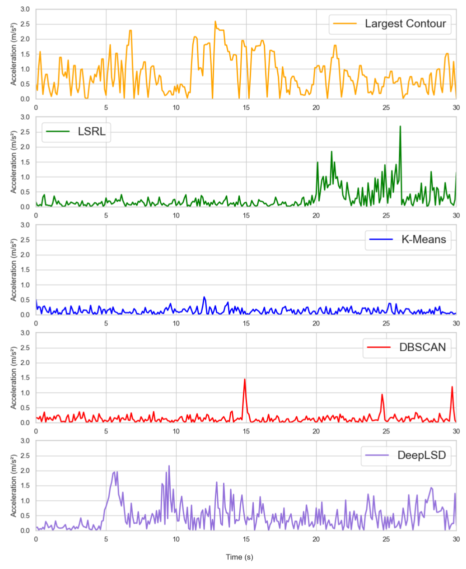


Fig. 11: Linear Momentum Measure Plot.

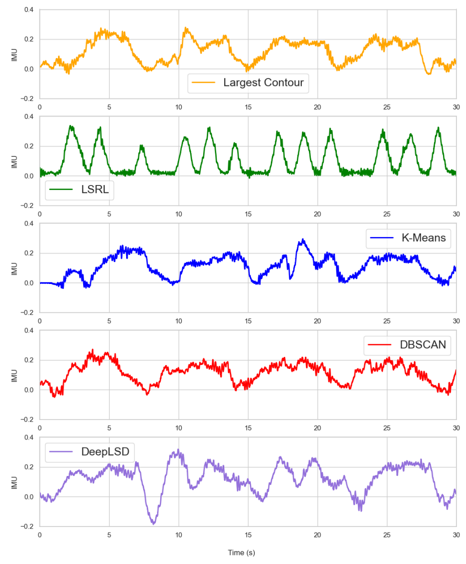


Fig. 12: Angular Momentum Measure Plot.

IX. Conclusion and Future Work

In conclusion, based on reliability, comfort, speed, and adaptability metrics, Linear Lane Search with K-Means and Lane Line Discrimination using DBSCAN performed the best out of the five tested algorithms. They both share the common approach of using unsupervised learning to distinguish between the two lane lines, enabling the vehicle to be centered between them. These two algorithms were subsequently integrated with adaptive speed control algorithms from the aforementioned V2X project to create a demonstration and gather additional data. They were able to achieve maximum speeds of 3.5m/s in the outer lane and 2.5m/s in the inner lane. In the future, we aim to conduct further research into deep learning-based approaches.

References

[1] LTU-REU, "Reu-2024," https://github.com/MMachado05/REU-2024, 2024, accessed: 2024-07-15.

[2] M. Noor-A-Rahim, Z. Liu, H. Lee, M. O. Khyam, J. He, D. Pesch, K. Moessner, W. Saad, and H. V. Poor, "6g for vehicle-to-everything (v2x) communications: Enabling technologies, challenges, and opportunities," *Proceedings of the IEEE*, vol. 110, no. 6, pp. 712–734, 2022.

[3] R. Kaddis, E. Stading, A. Bhuptani, H. Song, C.-J. Chung, and J. Siegel, "Developing, analyzing, and evaluating self-drive algorithms using electric vehicles on a test course," in *2022 IEEE 19th International Conference on Mobile Ad Hoc and Smart Systems (MASS)*, 2022, pp. 687–692.

[4] Ltu-Ros, "Ltu-ros/simple-sim-roads," Available at `https://github.com/ltu-ros/simple_sim_roads`, 2022, accessed: 2022-06-24.

[5] C.-J. Chung, "A simple lane following algorithm using a centroid of the largest blob," https://www.robofest.net/AutoEV/lanefollowing_algo22chung.pdf (accessed Aug 23, 2024), 2022, nSF Self-Drive REU 2022 Workshop at LTU. [Online]. Available: https://www.robofest.net/AutoEV/lanefollowing_algo22chung.pdf

[6] P. M.Venkatesh, "A simple bird's eye view transformation technique," *International Journal of Scientific Engineering Research*, 2024, accessed: 2022-05. [Online]. Available: https://www.ijser.org/researchpaper/A-Simple-Birds-Eye-View-Transformation-Technique.pdf

[7] J. Canny, "A computational approach to edge detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-8, no. 6, pp. 679–698, 1986.

[8] J. Matas, C. Galambos, and J. Kittler, "Robust detection of lines using the progressive probabilistic hough transform," *Computer Vision and Image Understanding*, vol. 78, no. 1, pp. 119–137, 2000. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1077314299908317

[9] A. M. Ikotun, A. E. Ezugwu, L. Abualigah, B. Abuhaija, and J. Heming, "K-means clustering algorithms: A comprehensive review, variants analysis, and advances in the era of big data," *Information Sciences*, vol. 622, pp. 178–210, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0020025522014633

[10] D. Deng, "Dbscan clustering algorithm based on density," in *2020 7th International Forum on Electrical Engineering and Automation (IFEEA)*, 2020, pp. 949–953.

[11] S. Yalabaka, C. R. Prasad, P. Sanjana, B. Lokesh, T. Shradha, and S. Samala, "Lane detection using deep learning techniques," in *2022 8th International Conference on Signal Processing and Communication (ICSC)*, 2022, pp. 412–416.

[12] R. Pautrat, D. Barath, V. Larsson, M. R. Oswald, and M. Pollefeys, "Deeplsd: Line segment detection and refinement with deep image gradients," in *2023 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2023, pp. 17 327–17 336.

[13] Z. M. U. Din, W. Razzaq, U. Arif, W. Ahmad, and W. Muhammad, "Real time ackerman steering angle control for self-driving car autonomous navigation," in *2019 4th International Conference on Emerging Trends in Engineering, Sciences and Technology (ICEEST)*, 2019, pp. 1–4.

[14] C. C. Robusto, "The cosine-haversine formula," *The American Mathematical Monthly*, vol. 64, no. 1, pp. 38–40, 1957. [Online]. Available: http://www.jstor.org/stable/2309088