DEPARTMENT OF TRANSPORTATION

Development of a smart phone app to warn the driver of unintentional lane departure using GPS technology

M. I. Hayee, Principal Investigator
University of Minnesota Duluth

October 2024

Research Project
Final Report 2024-25

LRRB
LOCAL ROAD RESEARCH BOARD

To get this document in an alternative format or language, please call 651-366-4720 (711 or 1-800-627-3529 for MN Relay). You can also email your request to ADArequest.dot@state.mn.us. Please make your request at least two weeks before you need the document.

Technical Report Documentation Page

1. Report No.	2.	3. Recipients Accession No.
MN 2024-25		

4. Title and Subtitle	5. Report Date
Development of a smart phone app to warn the driver of unintentional lane departure using GPS technology	October 2024
	6.

7. Author(s)	8. Performing Organization Report No.
M. I. Hayee & N. Z. Tasnim	

9. Performing Organization Name and Address	10. Project/Task/Work Unit No.
Electrical Engineering Department University of Minnesota Duluth 1023 University Drive, 271 MWAH Duluth, MN 55812	# 2021015
	11. Contract (C) or Grant (G) No. (c) 1036207

12. Sponsoring Organization Name and Address	13. Type of Report and Period Covered
Minnesota Department of Transportation Office of Research & Innovation 395 John Ireland Boulevard, MS 330 St. Paul, Minnesota 55155-1899	Final Report
	14. Sponsoring Agency Code

15. Supplementary Notes

http://mdl.mndot.gov/

16. Abstract (Limit: 250 words)

Unintentional lane departure is a significant safety risk. Currently, available commercial lane departure warning systems use vision-based or GPS technology with lane-level resolution. These techniques have their own performance limitations in poor weather conditions. We have previously developed a lane departure detection (LDD) algorithm using standard GPS technology. Our algorithm acquires the trajectory of a moving vehicle in real-time from a standard GPS receiver and compares it with a road reference heading (RRH) to detect any potential lane departure. The necessary RRH is obtained from one or more past trajectories using our RRH generation algorithm. This approach has a significant limitation due to its dependency on past trajectories. To overcome this limitation, we have integrated Google routes in addition to past trajectories to extract the RRH of any given road. This advancement has been incorporated into a newly developed smartphone app, which now combines our previously developed LDD algorithm with the enhanced RRH generation algorithm. The app effectively detects lane departures and provides real-time audible warnings to drivers. Additionally, we have designed the app's database structure and completed the programming of the necessary algorithms. To evaluate the performance of the newly developed smartphone app, we perform many field tests on a freeway. Our field test results show that our smartphone app can accurately detect all lane departures on long straight sections of the freeway irrespective of whether the RRH is generated from a Google route or past trajectory.

17. Document Analysis/Descriptors	18. Availability Statement
Smartphones, Mobility applications, Lane keeping, Warning systems	No restrictions. Document available from: National Technical Information Services, Alexandria, Virginia 22312

19. Security Class (this report)	20. Security Class (this page)	21. No. of Pages	22. Price
Unclassified	Unclassified	74	

Development of a smart phone app to warn the driver of unintentional lane departure using GPS technology

Final Report

Prepared by:

M. I. Hayee
N. Z. Tasnim
Electrical Engineering Department
University of Minnesota Duluth

October 2024

Published by:

Minnesota Department of Transportation
Office of Research & Innovation
395 John Ireland Boulevard, MS 330
St. Paul, Minnesota 55155-1899

Acknowledgements

The authors would like to thank Victor Lund, the project technical liaison at the St. Louis County Public Works, for his frequent discussions providing valuable insights into the project. In addition, the authors would like to thank Jackie Jiran for her assistance as project coordinator.

The research project benefited greatly from the feedback and input provided by members of the Technical Advisory Panel. Specifically, the authors would like to thank Rashmi Brewer (State Aid for Local Transportation), Bradley Estochen (Ramsey County), Joe Gustafson, (Washington County Public Works), Cory Johnson (CAV-X), Holly Kostrzewski (NHTSA), Derek Leuer (Traffic Engineering), Allison Nicolson (Essentia Health), Tara Olds (CAV-X), Brent Rusco (Research and Innovation) and Julie Swiler (Research and Innovation) for their help in completing the project.

Table of Contents

Chapter 1: Introduction **1**

1.1 Background and Motivation 1

1.2 System Architecture 3

Chapter 2: Development of a Backend Browser Application to Generate RRH from Google Routes **5**

2.1 RRH Generation from Google Routes 5

2.1.1 Algorithm and Parameter Optimization 7

2.2 Comparison between RRH Generated from a Google Route and a Past Trajectory 10

Chapter 3: Field Tests and Results **12**

3.1 Lane Departure Detection using an RRH Generated from a Google Route 12

3.2 Lane Departure Detection using an RRH Generation from a Past Trajectory 13

Chapter 4: Development of an App to Detect Lane Departure in Real Time **15**

4.1 Interconnection between the Smartphone App and the Browser Application 15

4.2 Architecture of the Smartphone App and RRH Database 16

4.2.1 Secure Connection 16

4.2.2 Trajectory Upload 17

4.2.3 RRH Extraction and Retrieval 18

4.3 App Database Structure 18

4.3.1 RRH Database 19

4.4 App User Interface 23

4.5 Scope of the App 25

4.6 Demo Link 26

Chapter 5: Expected Benefits, Future Work and Conclusions **28**

5.1 Expected Benefits and Implementation Steps 28

5.2 Conclusions 29

References **30**

Appendix A: Code Used for Implementation of RRH Generation Algorithm for Web Browser Tool

Appendix B: Code used for Implementation of LDD Algorithm for Web Browser Tool

Appendix C: Code used for Implementation of LDD Algorithm in Smartphone App

List of Figures

Figure 1.1 Schematic architecture of the originally proposed smartphone app. 3

Figure 1.2 (a) Architecture of the LDWS system. (b) Conceptual diagram demonstrating that RRH (blue dotted line) can be generated from a Google route (blue solid line, where points A and B are the start and end points, respectively) to detect an intentional lane change from right to left (red dotted line). The road illustrated in this figure is a 4-lane road with the Google route shown in the middle. 4

Figure 2. 1 (a) Browser application user interface showing the user-specified route on the Google map to initiate RRH generation from the Google route. Points A and B indicate the start and end location of the desired route, respectively. (b) Google route superimposed on the generated RRH with straight, curve and transition sections indicated by red, blue, and green colors along the route..... 7

Figure 2.2 Lowpass filter with a corner frequency of 20 Hz and a total stop frequency of 125 Hz..... 8

Figure 2.3 Google heading (magenta) and filtered Google heading (green) vs. distance..... 9

Figure 2.4 Differential (red) and average differential heading (blue) vs. distance. The horizontal (black dashed) lines indicate the threshold which decides the markings of the sections..... 10

Figure 2.5 A graph plotting the Google route heading (green), RRH generated (black solid line) from the Google route, and a mask (black dotted line) to indicate the straight, curved, and transition sections.... 11

Figure 2.6 A past trajectory heading (green) and corresponding RRH (solid black) vs. distance. A mask (black dotted line) is drawn to indicate the straight, curve, and transition sections..... 12

Figure 3.1 (a) RRH resulting from the Google route (black) along with the headings of four different trajectories (red, blue, purple & orange) on the same route vs. distance. A zoomed in portion of the Google RRH and 4 trajectories between 3000 to 3500 m distance range is shown as an in-set in the figure. (b) Vehicle accumulative lateral distance vs. traveled distance for four different driven trajectories (red, blue, purple & orange). The vehicle accumulative lateral distance is calculated using Google RRH..... 14

Figure 3.2 (a) RRH resulting from one of the past trajectories (black) along with the headings of three different trajectories (red, blue & purple) vs. distance. (b) Vehicle accumulative lateral distance vs. traveled distance for three different driven trajectories (red, blue & purple). The vehicle accumulative lateral distance is calculated using the RRH extracted from a past trajectory..... 15

Figure 4.1 System architecture demonstrating the interconnection between the smartphone app and the backend browser-based application……..16

Figure 4.2 Schematic architecture of the smartphone app and its interconnection with GCP containing app database……18

Figure 4.3: Snippet of data stored in JSON format……19

Figure 4.4 A typical road infrastructure containing two freeways to illustrate RRH database structure. The colored dashed lines are trajectories on that road obtained at different times. Points A and B on FW1 indicate the start and end points of a Google route for which an RRH has been generated……21

Figure 4.5 (a) Screenshot of the app seeking user permission to access the device's location. (b) Screenshot of the registration page of the app. (c) Screenshot of the app showing the device location on the map……24

Figure 4.6 Picture from the Google Map of the 7 km route on Interstate I-35 with one lane departure highlighted with a yellow circle. Below is a zoomed in picture of the lane departure portion circled in yellow indicating the lane width with yellow lines. The lane change inside the yellow circle was made from left to right……25

Figure 4.7 External GPS receiver device along with the Android phone……26

Figure 4.8 Location symbol used in the demo video……28

List of Tables

Table 4. 1 RRH database showing updated database after every new trajectory……22

List of Abbreviations

LDWS: Lane Departure Warning System

RRH: Road Reference Heading

LDD: Lane Departure Detection

ADAS: Advanced Driver Assistance Systems

PAH: Path Average Heading

PAHS: Path Average Heading Slope

IH: Initial Heading

GCP: Google Cloud Platform

Executive Summary

Unintentional lane departure is a significant safety risk. There are many commercially available types of advanced driver assistance systems (ADASs) that can detect an unintentional lane departure and warn the driver in real time. Almost all commercially available lane departure warning systems use vision-based or optical-scanning technologies to detect unintentional lane departure. Although such systems work well in favorable weather and road conditions, their performance deteriorates when the road conditions are not favorable, including when an absent or irregular/broken lane marking is present or in harsh weather conditions such as fog, rain, or snow, resulting in inaccurate lane departure detection. Furthermore, such systems are complex and expensive to implement, inhibiting their widespread market penetration.

We previously developed a lane departure warning system (LDWS) based on a standard GPS receiver incorporating two algorithms to detect an unintentional lane departure. The current project is in fact the third phase of the same research project. In this phase, we develop a smartphone app that can alert a driver when an unintentional lane departure takes place. The smartphone app incorporates two previously developed algorithms, namely the road reference heading (RRH) algorithm and the lane departure detection (LDD) algorithm, to detect an unintentional lane departure. The RRH algorithm generates the RRH from a vehicle's past trajectory, while the LDD algorithm compares a vehicle's current trajectory on that road to the RRH of a given road to detect lane departure in real time. In addition to developing the smartphone app, we also incorporate another feature that was lacking in the lane detection method using the previously developed RRH algorithm.

A significant limitation of the previous method was the dependency on past trajectories. A vehicle must have traveled on the road at least once in the past to use the trajectory for RRH generation to detect unintentional lane departures of a future trip on the same route. To avoid dependency on past trajectories, we came up with a modified RRH algorithm, which can extract RRH from Google routes in addition to the past trajectories for any given road. In addition, we enhanced the smartphone app to accommodate this feature. In the updated app, we have incorporated necessary modifications and enhancements of the RRH generation algorithm to accommodate RRH generation from Google routes in addition to a vehicle's past trajectories. With this additional feature of extracting RRH from Google routes, our app is capable of detecting lane departures of a vehicle in real-time on any given road, regardless of whether the vehicle has traveled on that road in the past.

In addition to developing a smartphone app, we have also developed a backend browser application to facilitate the generation of RRH from Google routes or a vehicle's past trajectories. The backend browser application is capable of extracting RRH either from a Google route or from a vehicle's past trajectory for any given road. The addition of the new feature of extracting RRH from Google routes makes the app more feasible when a past trajectory is not available. When generating RRH from a past trajectory, a user can upload the past trajectory on any given road in the backend browser for RRH to be generated for that road. On the other hand, a user can also specify the start and end location points on Google Maps for RRH to be generated from the Google route. Once the RRH is extracted either from a past

trajectory or from a Google route using our developed backend browser application, it is saved for future use for running the LDD algorithm in the smartphone app.

To evaluate the effectiveness of the new feature of extracting RRH from Google routes, we have compared the RRH generated from a Google route trajectory with that of a past trajectory and found both RRHs to be comparable. Furthermore, to evaluate the accuracy of lane departure detection using Google RRH, we performed many field tests on a freeway. Our field test results showed that our lane LDD algorithm running in the smartphone app can accurately detect all lane departures on long straight sections of the freeway irrespective of whether the RRH was generated from a Google route or a past trajectory.

Our current form of smartphone app has some limitations. We have written the code of the smartphone app in a language (Dart) that can be used to compile the code for both Android and iOS platforms. However, the current version of the developed smartphone app cannot be used for the iOS platform (Apple devices) because we have to use an external GPS receiver for the app to work. This limitation is due to the fact that GPS receivers in both Apple and Android devices are forced to operate at 1 Hz frequency as opposed to 10 Hz or higher frequency that they are capable of. Both iOS and Android OS do this to prolong the battery life of smartphones even if the smartphone is on charger. For our LDD algorithm to successfully detect lane departure, we need the GPS receiver to operate at 10 Hz frequency.

To alleviate this problem, we are currently using an external GPS device with the smartphone, which can give us more control over the frequency of the GPS receiver. Due to this, our app currently does not work for Apple devices as the iOS platform prohibits the use of accessing GPS data from an external GPS receiver. Although, our app can run on any Android phone, it cannot work unless an external GPS device is available. Despite this, these limitations do not change the end goal of the project. We are still able to develop a marketable app that can be used to secure future partnerships, e.g., Google, to adapt this as a feature in their maps. Also, if a partnership is secured, it is straightforward for both Google and Apple to make changes in their operating system to let the GPS operate at 10 Hz, eliminating the reliance on any external device. The use of an external device is only an intermediate step for us to demonstrate the power/feature of our app in absence of not being able to control the operating system of the smartphones.

Chapter 1: Introduction

This chapter covers the background and motivation behind this project. It also covers the importance of lane departure warning systems (LDWS) and summarizes relevant past work.

1.1 Background and Motivation

According to a World Health Organization (WHO) report, roughly 1.3 million people die yearly from road traffic crashes *(1)*. A survey that examined the national sample of US crashes from 2005 to 2007 identified driver error as the critical reason contributing to 94 percent of crashes *(2)*. Advanced driver assistance systems (ADAS) *(3)* facilitate drivers to make well-informed decisions and consequently help them avoid collisions. The recent advancements in ADAS technologies have increased safety for both drivers and pedestrians. The analysis of a study that estimated the safety benefits of in-vehicle lane departure warning (LDW) and lane-keeping aid (LKA) systems in reducing relevant car injury crashes demonstrated that such systems reduced head-on and single-vehicle injury crashes by 53% at a 95% confidence interval on Swedish roads within a specific speed limit range and road surface conditions *(4)*. Most lane departure warning systems (LDWSs) depend on image processing and use cameras, infrared, or laser sensors to estimate a vehicle's lateral shift within a lane to detect an unintentional lane departure *(5-12)*. Vision-based LDWSs face challenges adapting to diverse weather conditions, handling light changes, and mitigating shadow effects. Blockage of lane markings by other vehicles in LDWS images can also cause detection failures. Real-time processing of captured images is crucial for vision-based LDWS, necessitating synchronous image processing speed to ensure timely and safe detection *(13)*. In *(14)*, a real-time calibration-free lane departure warning system algorithm using the Gaussian pyramid preprocessing and Edge Drawing Lines algorithm was proposed, achieving high accuracy (99.36%) and fast processing (80 fps), suitable for integration into self-driving systems in vehilces, outperforming existing algorithms.

Although advanced image processing techniques work well in diminished lighting scenarios *(15, 16)*, many of today's commercially available image processing-based LDWSs have performance issues under unfavorable weather or road conditions like fog, snow, or worn-out road markings potentially leading to inaccurate lane detection or overlooking genuine lane markings. To tackle the limitations posed by adverse weather conditions and degraded road markings on the performance of image processing-based LDWSs, recent advancements have emerged. For instance, in *(17)*, a lane departure warning system was introduced, specifically designed to combat illumination challenges prevalent in driving environments. This system demonstrated a 93% detection rate, surpassing traditional approaches, even in scenarios with blurred lane markings or low sun angles. Furthermore, in *(18)*, a self-tuned algorithm was developed that integrated fuzzy logic-based adaptive functions with edge identification and line detection modules. This algorithm aimed to enhance image quality in challenging weather conditions, although it may have limitations in handling multiple and curved lines. To address some of these performance issues, Global Positioning System (GPS) technology was integrated within a vision-based LDWS. To estimate a vehicle's lateral movement in its lane, such systems employ differential GPS (DGPS)

technology *(19)* and/or inertial navigation and/or odometry sensors *(20)* in addition to high-resolution digital maps, making them challenging to deploy and expensive *(21)*. In *(22)*, a lane departure detection algorithm for vehicles was proposed using Geographic Information System (GIS) and DGPS data with high positioning accuracy under 20 cm. It calculated lane segment distances based on vehicle position, using Bezier curves for curved sections. Although GPS is a very helpful tool for navigation, there are situations in which signal interference, multipath effects, or complex road networks may make it less reliable in streets and urban areas. In *(23)*, it is demonstrated that combining vision, height measurements, and a lane map can significantly reduce drift when a GPS signal is lost, such as in dense foliage or urban environments. By using existing sensors in commercial vehicles, the system achieves submeter accuracy in lateral distance measurement.

Our previously developed LDWS system was based on a standard GPS receiver without using any image processing or optical scanning devices, or high-resolution digital maps. To detect a lane departure, our developed LDWS system estimated lateral vehicle shift using standard GPS technology. The lateral vehicle shift was estimated by comparing the vehicle trajectory acquired by a standard GPS receiver with a road reference heading (RRH), which was obtained using a newly developed algorithm *(24)*. Our lane departure detection (LDD) algorithm accumulated instantaneous vehicle lateral shifts to detect an unintentional lane departure in real-time *(25)*. The current project was in fact the third phase of an ongoing research project funded by the Minnesota Local Research Board (LRRB) and Minnesota Department of Transportation (MnDOT). In the first two phases of this project, we developed two novel algorithms (LDD and RRH). The current phase of the project included the development of a smartphone app that can alert a driver when an unintentional lane departure takes place. The smartphone app incorporated two previously developed algorithms, namely road reference heading (RRH) algorithm and lane departure detection (LDD) algorithm to detect an unintentional lane departure. The RRH algorithm generated the RRH from a vehicle's past trajectory, while the LDD algorithm compared a vehicle's current trajectory on that road with the RRH of a given road to detect lane departure in real time.

The schematic architecture of the smartphone app, as proposed at the beginning of this project (current phase of the project), is shown in Figure 1.1. The proposed smartphone app incorporates our previously developed LDWS that would extract RRH of a given road from a vehicle's past trajectories on that road and save it for future use in the app database. In the future, if the vehicle departs a lane on that road, the app will extract the corresponding RRH from the database residing in a cloud server and use the LDD algorithm running in the phone to detect a lane departure.

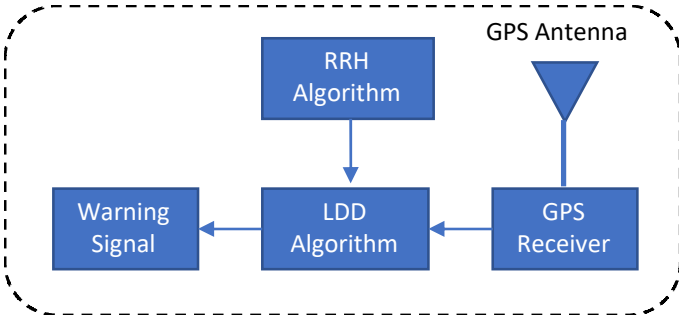


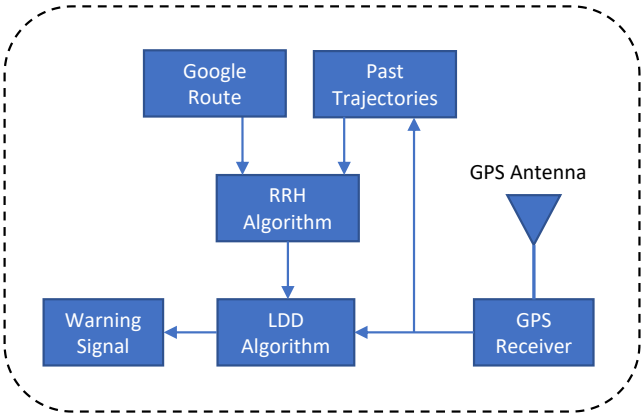
Figure 1.1 Schematic architecture of the originally proposed smartphone app.

One major limitation of the previously developed LDWS system is the dependency on past trajectories. To detect an unintentional lane departure of any vehicle in real-time on a given road, the vehicle must travel on the same road at least once in the past to use that trajectory for RRH generation. During subsequent trips on the same route, the system can detect a potential lane departure using already generated RRH to warn the driver. To avoid dependency on past trajectories, in the smartphone app developed as a result of this project, we also included a Google route option to extract the RRH of any given road in addition to extracting RRH from past trajectories. Google routes are available for all roads within the US through Google Maps as navigational routes. We have used a Google navigational route on any given road provided by Google Maps as a Google route to extract the RRH of that road using our previously developed RRH generation algorithm with some modifications. The extracted RRH from Google routes was used to detect an unintentional lane departure in real time and alert the driver with an audible warning. We also compared the lane departure detection results using RRH extracted from Google routes to the ones from past trajectories and found that lane departure detection resulting from Google RRH and RRH from the past trajectories were comparable. We also performed field tests to evaluate the results showing that our system can accurately detect lane departure on long straight sections of freeways. This showed the potential of our LDWS to be used for all US roads without the dependency on past trajectories.

1.2 System Architecture

The previously developed LDWS system has been enhanced to work for both past vehicle trajectories and Google routes as shown in Figure 1.2(a). The enhanced LDWS can either generate RRH using past trajectories or from Google routes obtained from Google Maps. In the updated LDWS, we have incorporated the necessary modifications and enhancements of the RRH generation algorithm to accommodate RRH generation from Google trajectories. With this additional feature of extracting RRH, our LDWS is capable of detecting lane departures of a vehicle in real-time on any road, regardless of whether the vehicle has traveled on that road in the past.

(a)



(b)

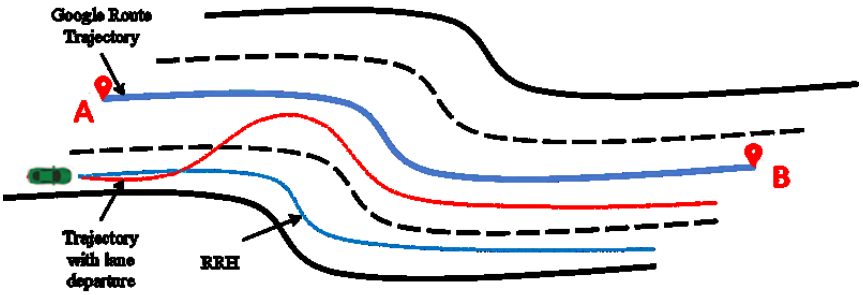


Figure 1.2 (a) Architecture of the LDWS system. (b) Conceptual diagram demonstrating that RRH (blue dotted line) can be generated from a Google route (blue solid line, where points A and B are the start and end points, respectively) to detect an intentional lane change from right to left (red dotted line). The road illustrated in this figure is a 4-lane road with the Google route shown in the middle.

Our LDWS compares the vehicle trajectory in real-time with the RRH of that road to detect a lane departure using our LDD algorithm as depicted in Figure 1.2(b), where a vehicle trajectory with a lane departure (red dotted line) can be compared with the RRH (blue dotted line) obtained from the Google route (blue solid line) or from one of the past trajectories (not shown in Figure 1.2(b)) to determine lane departure using our LDD algorithm (18). We have also made enhancements to our LDD algorithm to detect lane departure more efficiently.

The next chapter describes the development of a backend browser application to generate RRH from Google routes and compares the RRH generated from Google routes with the RRH generated from past trajectories. Chapter 3 discusses the field tests and results where the effectiveness of the LDWS system is evaluated and compared for RRH generated from both Google routes and past trajectories. Chapter 4 is based on the app development and the interconnection between the app and the browser-based application to detect lane departure in real time, followed by conclusion and future work in Chapter 5.

Chapter 2: Development of a Backend Browser Application to Generate RRH from Google Routes

This chapter highlights the generation of an RRH for any given road from Google routes obtained from Google Maps.

2.1 RRH Generation from Google Routes

We have developed a backend browser-based application (Figure 2.1) where we can specify the start and end points of the desired route on a Google map to obtain the RRH from the Google route on almost any road within the US. The code for the backend browser-based application is written in Angular using TypeScript. The user specifies the start and end location points in Google Maps to define the Google route necessary for RRH generation. Our backend browser application accesses Google Maps' API for turn-by-turn directions between the start and end points of the route and returns an array of locations with latitudes and longitudes. In this array of locations provided by Google Maps, there are more location points on a curve segment of the road as compared to a straight segment of the road. This array of latitudes and longitudes represents the Google route from which RRH is generated.

We preprocess the array of location points to obtain a uniform distance resolution before applying our RRH generation algorithm. To obtain a uniform distance resolution, we have to access Google Maps API a few times between intermediate location points in the array until we get the desired resolution so that every two consecutive points in the Google Maps array of locations have the same distance as the desired resolution typically in 1 to 3 m range. We have to do more iterations on straight sections in the array as compared to a curve section because there are fewer number of points on a straight section as compared to a curve section.

Our developed RRH generation algorithm characterizes a typical road segment into straight and curve sections, as any typical road can have a combination of straight and curve portions. However, the roads do not curve abruptly; therefore, the section between a straight and a curve is characterized as a transition section by the algorithm. Our algorithm generates an RRH from the preprocessed Google route (uniform array of locations) for any given portion of the road in three major steps.

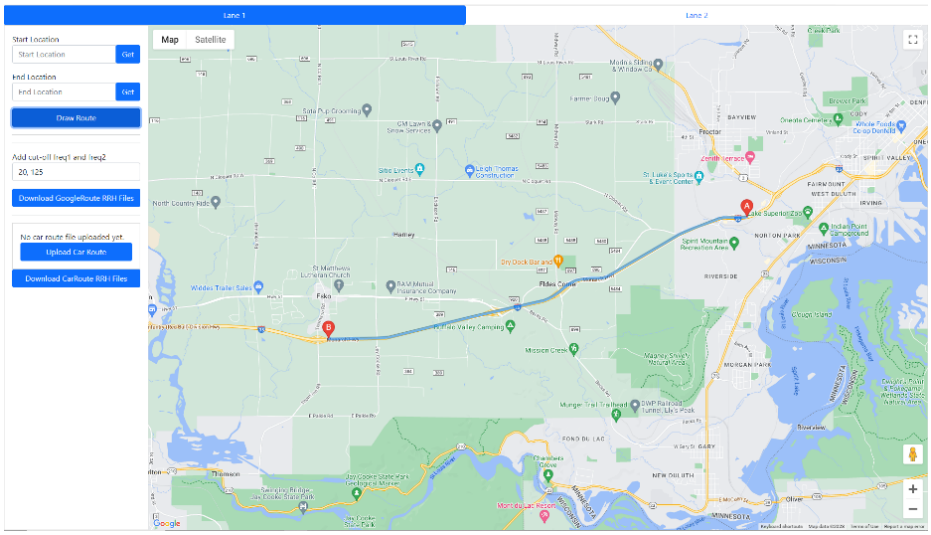
1) ***Identification:*** In the first step, all straight, curve and transition sections are identified on that road.

2) ***Characterization:*** After identification, each section is characterized with a set of optimized parameters that determines the RRH value at any given point on that road. Every straight section is characterized by a Path Average Heading (PAH), as the heading remains constant for the entire length of a straight section. Since the heading of a curve section varies uniformly with distance, it is characterized by a Path Average Heading Slope (PAHS) and an initial heading (IH), where IH is the heading at the

beginning point of the curve section. Each transition section is characterized the same way as a curve section.

3) ***Aggregation:*** Finally, all the individual road sections are combined to obtain a composite RRH for that road to be used with the LDD algorithm for lane departure detection.

(a)



(b)

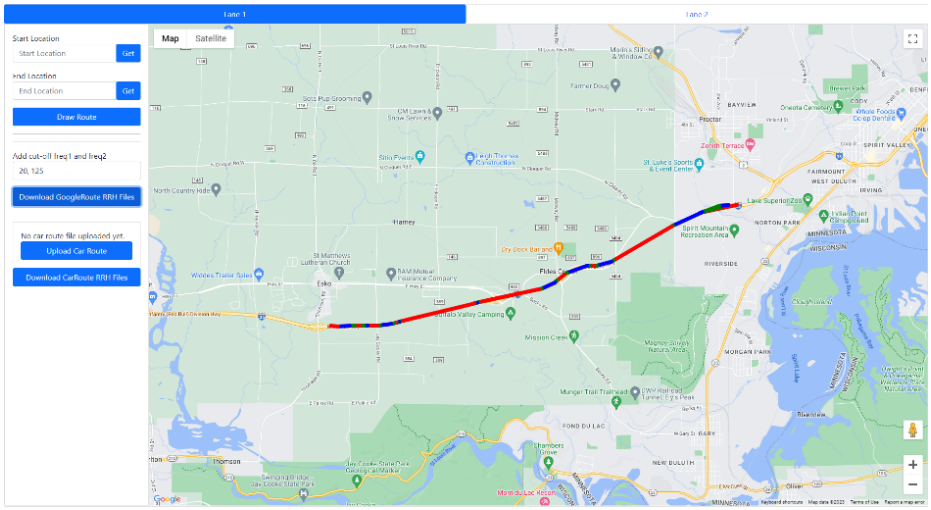


Figure 2. 1 (a) Browser application user interface showing the user-specified route on the Google map to initiate RRH generation from the Google route. Points A and B indicate the start and end location of the desired route, respectively. (b) Google route superimposed on the generated RRH with straight, curve and transition sections indicated by red, blue, and green colors along the route.

The process of generating RRH from the Google route using the browser application is illustrated in Figure 2.1. Our browser application lets the user specify the start and end points on a given road on

Google Maps, as shown in Figure 2.1(a), where points A and B are the start and end points, respectively. After the user specifies the start and end points, our browser application runs our RRH generation algorithm and converts the Google route to RRH consisting of all sections of the road, as shown in Figure 2.1(b), where the red, blue, and green colors along the Google route indicate straight, curve and transition sections, respectively. Our browser application also generates a corresponding data file containing all the relevant information of all the sections of RRH within the Google route. The information in this datafile contains the start and end points of each section (straight, curve and transition) and the relevant parameter values characterizing each section to determine the RRH value for any point on the road. Using this information, an RRH value can be calculated on any given point of the road between the start and end points specified by the user.

2.1.1 Algorithm and Parameter Optimization

The implementation steps of our RRH generation algorithm are as follows:

1) First, the Google route is obtained from the browser-based application containing the array of points in terms of longitude and latitude for each point. The Google route is then preprocessed for uniform distance resolution so that each of the two consecutive points has a constant distance between them (1-3 m range), as explained earlier.

2) Then, the heading between two consecutive points is calculated from the longitude and latitude of those points *(24)*. The heading array is then filtered with a lowpass filter having a corner frequency of 20 Hz (cycles/m) and a total stop frequency of 125 Hz (cycles/m). Figure 2.2 is the frequency response of the lowpass filter used.

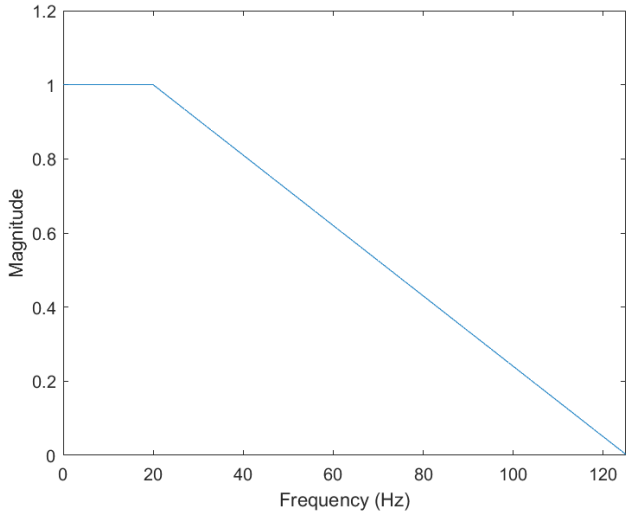


Figure 2.2 Lowpass filter with a corner frequency of 20 Hz and a total stop frequency of 125 Hz.

The filtering is done to smoothen the heading array obtained from Google Maps. The choice of corner and stop frequencies is made to ensure the removal of undesired ripple effects from the heading array as shown by the filtered Google heading in Figure 2.3.

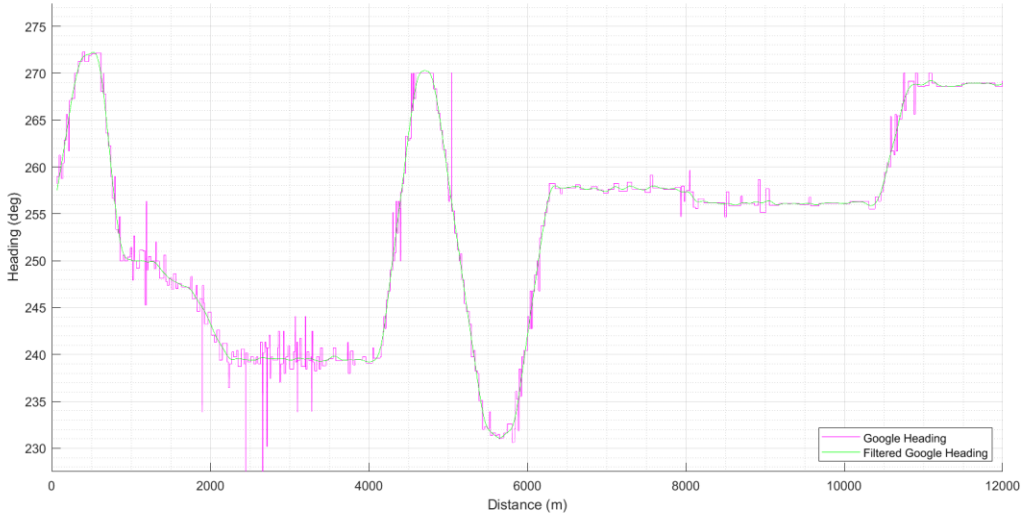


Figure 2.3 Google heading (magenta) and filtered Google heading (green) vs. distance.

3) Next, differential headings per meter are calculated from the filtered heading array. The differential heading is then smoothened to further minimize undesired ripples as shown in Figure 2.4. The smoothening is obtained by averaging differential heading at each point over 40 meters, i.e., ± 20m on either side of the given point.

4) The average differential heading array is then used to identify the straight sections using a threshold of 0.002 deg/m (Figure 2.4), i.e., any consecutive portion with a differential heading of less than the threshold of 0.002 deg/m will be considered a straight section *(24)*. This step will generate the start and end points of each of the straight sections present in the Google trajectory. Please note that we combined two consecutive straight sections that are not too far apart from each other because a road cannot curve abruptly. For that purpose, we are using a range of 75-100 m as a parameter to combine two consecutive straight sections. Straight sections are then characterized by calculating a PAH between the start and end points of each straight section.

5) After identifying and characterizing straight sections, curve sections are identified and characterized. For this purpose, any portions of the Google route between two consecutive straight sections are considered curve sections, i.e., any contiguous portions on a Google route having a differential heading above the average differential heading threshold of 0.002 deg/m. A PAHS value for each of the curve sections between two consecutive straight sections is then calculated from the average differential heading. The beginning and end points of each of the curve sections are identified where the average differential heading is closest to the calculated PAHS on that curve section. The heading at the beginning point of each curve section becomes the IH for that curve section.

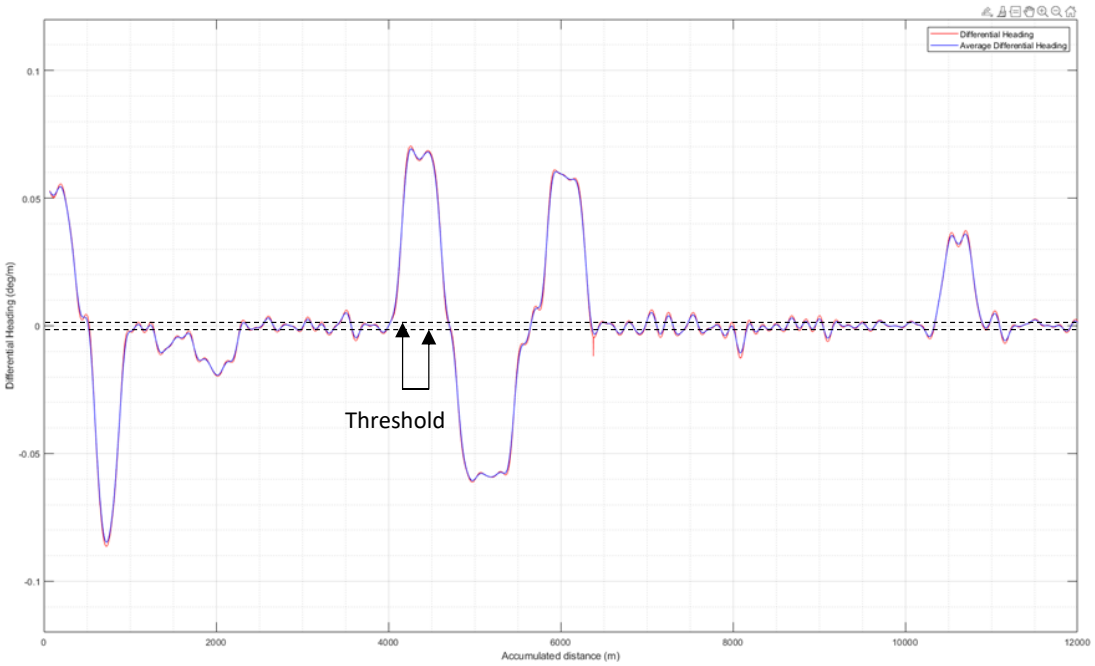


Figure 2.4 Differential (red) and average differential heading (blue) vs. distance. The horizontal (black dashed) lines indicate the threshold which decides the markings of the sections.

6) The above method of identifying the start and end points of the curve section tends to make the transition sections longer for those curves which have a higher value of PAHS. Therefore, for such curves which have a PAHS value of >0.02 deg/m, we apply selective thresholding to reduce the lengths of the transition sections and increase the lengths of the adjacent straight sections while keeping the curve section length the same. To achieve this, we increase the threshold of 0.002 deg/m (as in step 4) to 0.01 deg/m (5 times the original threshold) only for those curves which have a PAHS value of >0.02 deg/m.

7) Similarly, some of the curves with a small PAHS may be falsely accounted for as curve sections because their PAHS value is only slightly higher than the threshold (0.002 deg/m) to be identified as curve sections to begin with. To identify and eliminate such false curve sections, the PAHS value for each of the curve sections is recalculated by dividing the difference of the PAH of the two surrounding straight sections by the distance between them. If such a revised PAHS turns out to be smaller than 0.002 deg/m (original threshold), then that curve is absorbed in the surrounding straight sections.

8) The characterized parameters for each straight and curve section are then optimized to minimize the differential heading error *(24)* between the RRH and Google route heading over the entire portion of each of the straight and curve sections.

9) In the end, transition sections are identified and characterized. Any portion of the Google trajectory between a curve and a straight section is marked as a transition section. The heading at the

start point of the transition section becomes the IH of the transition section. The PAHS of the transition section is calculated from the difference in headings between the start and end points of the transition section divided by the distance between them.

2.2 Comparison between RRH Generated from a Google Route and a Past Trajectory

To see the effectiveness of the RRH generated from the Google route, we have plotted the Google route heading (green solid line) and the RRH (black solid line) with respect to distance as shown in Figure 2.5. To identify straight, curve and transition sections on the RRH, a mask with a black dotted line is also shown in Figure 2.5. The mask has fixed but different heading values for straight, curve and transition sections to distinguish them from each other. The straight sections have a fixed mask value of 252.5° (almost in the middle of Figure 2.5). For each of the curve and transition sections, two different mask values are attributed to each curve or transition section depending on the PAHS. The mask value for the curve section is either 275° or 230°, depending on whether the PAHS is positive or negative, respectively. Similarly, the mask value for the transition section is 263.75° or 241.25°, depending on whether the PAHS is positive or negative. Please note that specific mask values are chosen to mark the difference between the straight, curve and transition sections within the RRH for the Google route in Figure 2.5.

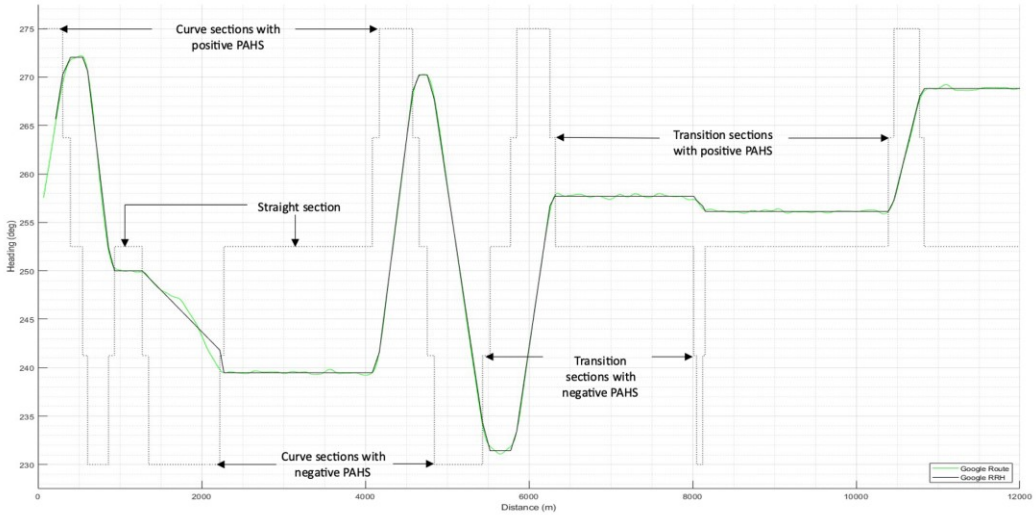


Figure 2.5 A graph plotting the Google route heading (green), RRH generated (black solid line) from the Google route, and a mask (black dotted line) to indicate the straight, curved, and transition sections.

The RRH values (black solid line) match pretty well with the Google route heading (green), especially for straight sections (Figure 2.5). The match between the RRH and the Google route varies for curve

sections, and we noticed that for one of the sharper curve sections between 1500 m and 2200 m, the match is not as good as for the other portions. (Figure 2.5).

This can be improved by tweaking the conditions of the RRH generation algorithm in the future. However, our main goal is that the RRH values for the straight sections are reliable so that unintentional lane departure on long stretches of the straight sections can be successfully detected.

We also wanted to compare the accuracy of the Google RRH with the RRH obtained from the past trajectories. Therefore, we applied the modified RRH generation algorithm to one of the past vehicle trajectories obtained on the Interstate I-35 southbound which is the same portion of I-35 for which the Google route was used to extract RRH. The trajectory length was a little shorter than 12 km covering almost the same portion as the Google route of Figure 2.5. The trajectory heading vs. distance is plotted in Figure 2.6. For reference, an RRH obtained from this trajectory is also shown in Figure 2.6. The same mask as in Figure 2.5 is also used in Figure 2.6 to differentiate among straight, curve, and transition sections of the RRH. Figure 2.6 shows that the generated RRH (black solid line) follows the vehicle trajectory heading (green) fairly well, indicating that the modified RRH generation algorithm also works well for the past trajectories. In fact, for a much sharper curve present in the trajectory (between 1500 m and 2200 m range), the match between RRH and the past trajectory is much better (Figure 2.6), as compared to the similar match for the Google route (Figure 2.5).

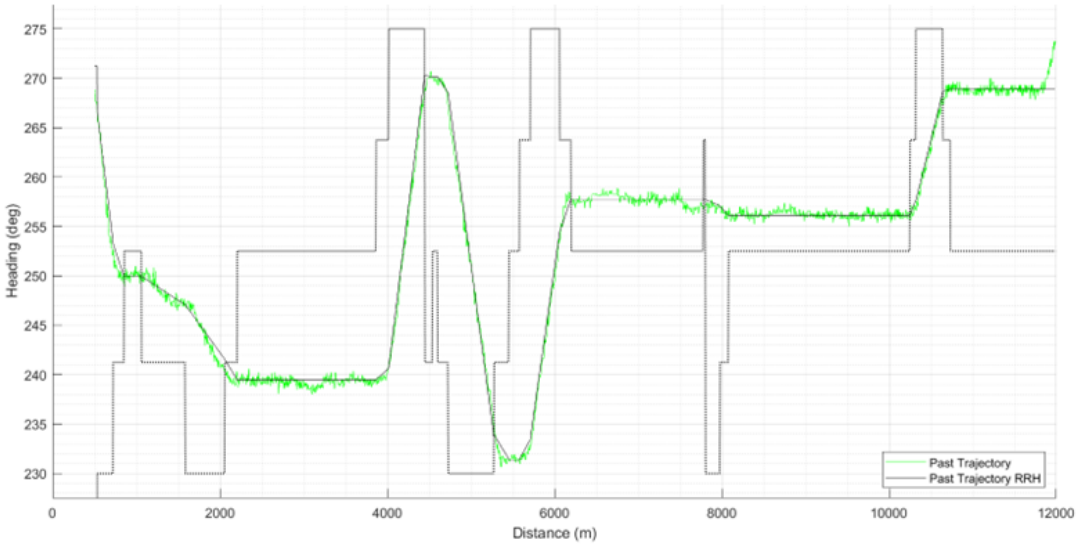


Figure 2.6 A past trajectory heading (green) and corresponding RRH (solid black) vs. distance. A mask (black dotted line) is drawn to indicate the straight, curve, and transition sections.

These discrepancies in Google RRH exist because sometimes the road width changes, i.e., the number of lanes cause abrupt heading deviations in the Google route. However, a more rigorous comparison of RRH obtained from the Google route and that of the past vehicle trajectory is made in the next chapter by detecting lane departures using our LDD algorithm.

Chapter 3: Field Tests and Results

This chapter highlights the field test results of the LDWS using the modified algorithm to generate RRH for any given road from Google routes as well as past trajectories. This chapter also compares the performance of the LDWS system when RRH is generated from Google routes to when generated from past trajectories.

3.1 Lane Departure Detection using an RRH Generated from a Google Route

We performed many field tests to evaluate the RRH obtained from the Google route using our LDD algorithm. All the field tests were performed by driving a test vehicle multiple times on the same 12 km segment of Interstate I-35 southbound, for which an RRH was extracted from the Google route, as discussed earlier. Each of the test runs covered a portion of the 12 km road segment to ensure that the test vehicle remained on the same portion of the road for which an RRH was extracted earlier from the Google route. The test vehicle was driven at about the speed limit (70 MPH) on the 4-lane freeway (2 lanes each way) and many lane changes were made intentionally during the field tests. For safety reasons, intentional lane changes were made to test the accuracy of lane departure detection. The vehicle trajectory data for each of the test runs were collected and evaluated with our LDD algorithm to detect the start and end of the lane changes present in each trajectory. Four such trajectories (red, blue, purple & orange colors) from the test runs are shown in Figure 3.1(a), where vehicle heading vs. vehicle traveled distance is plotted. For reference, Google RRH (black solid line) is also shown on the same scale in Figure 3.1(a).

In the first two trajectories (red and blue) shown in Figure 3.1(a), a total of 4 lane changes were made, and in the third trajectory (purple), only 2 lane changes were made. There was no lane change made for the fourth trajectory (orange) to ensure that no false alarm was detected by our LDD algorithm. All lane changes were made only on the straight portions of the road as unintentional lane departure is more relevant on longer stretches of straight portions of the road as discussed earlier.

Although, from the trajectories shown in Figure 3.1(a), lane changes cannot be visibly identified because there is a very small difference in the heading of a vehicle trajectory and the Google RRH. An example of a lane change is highlighted in a zoomed portion of Figure 3.1(a) between the 3000 m and 3500 m range to show that there is a lane change in two of the four trajectories (red and purple) in opposite directions indicated by the vehicle heading deviation in the opposite directions as compared to the Google RRH. Similarly, for the other two trajectories (blue and orange), there was no lane change for this portion, so no noticeable deviation was seen in vehicle heading as compared to the Google RRH.

(a) (b)

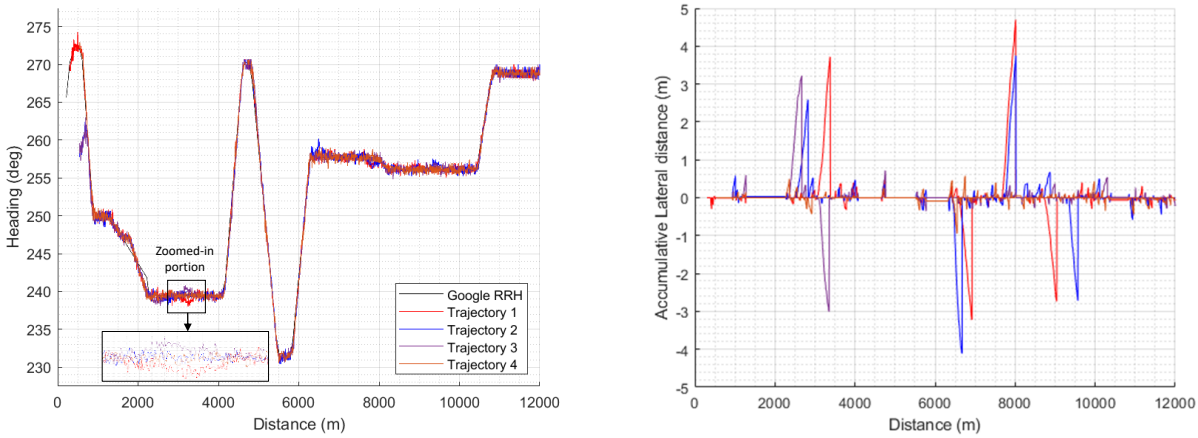


Figure 3.1 (a) RRH resulting from the Google route (black) along with the headings of four different trajectories (red, blue, purple & orange) on the same route vs. distance. A zoomed in portion of the Google RRH and 4 trajectories between 3000 to 3500 m distance range is shown as an in-set in the figure. (b) Vehicle accumulative lateral distance vs. traveled distance for four different driven trajectories (red, blue, purple & orange). The vehicle accumulative lateral distance is calculated using Google RRH.

To evaluate the accuracy of the lane change timing and duration, accumulative lateral distance of the vehicle for each trajectory vs. traveled distance is shown in Figure 3.1(b). The colors of the accumulated distance for each trajectory are kept the same as in Figure 3.1(a). When the accumulative lateral distance exceeds a certain threshold (~1m) value on either side, it is considered a lane departure. All lane changes were accurately identified by our algorithm indicating that the RRH obtained from Google route works well for detecting lane departures, either intentional (lane change) or unintentional.

The start and stop of each of the lane changes are clearly identifiable in Figure 3.1(b). For the orange trajectory, where there is no lane change, the accumulative lateral distance never exceeds the threshold of 1 m.

3.2 Lane Departure Detection using an RRH Generation from a Past Trajectory

We also wanted to compare the accuracy of lane changes being identified correctly when the RRH was generated from a Google route as well as a past trajectory. We have used Trajectory 4 (orange) of Figure 3.1 to generate an RRH from it because it does not have any lane changes. The remaining three trajectories (red, blue & purple) from the test runs were used to detect lane changes in them using the RRH extracted from one of the past trajectories (Trajectory 4).

(a) (b)

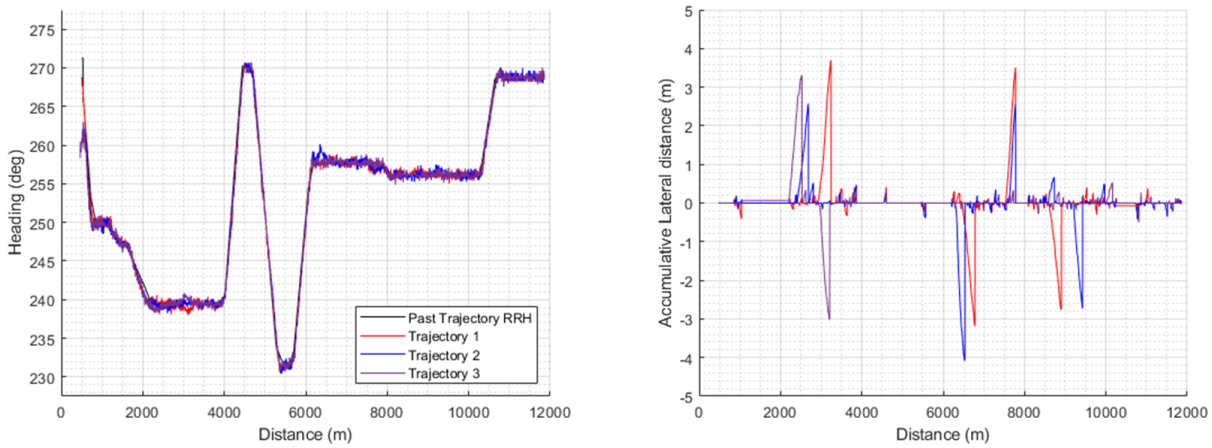


Figure 3.2 (a) RRH resulting from one of the past trajectories (black) along with the headings of three different trajectories (red, blue & purple) vs. distance. (b) Vehicle accumulative lateral distance vs. traveled distance for three different driven trajectories (red, blue & purple). The vehicle accumulative lateral distance is calculated using the RRH extracted from a past trajectory.

The heading of the three trajectories (red, blue and purple) are shown again in Figure 3.2(a), where vehicle heading is plotted with respect to traveled distance. As stated earlier, in two of these three trajectories (red and blue), a total of 4 lane changes were made in each, and in the third trajectory (purple), only 2 lane changes were made. For reference, RRH extracted from Trajectory 4 has also been shown in Figure 3.2(a).

To evaluate the accuracy of the lane change timing and duration, accumulative lateral distance of the vehicle for each of the three trajectories using the RRH from Trajectory 4 is calculated and is shown in Figure 3.2(b) vs. traveled distance. The colors of the accumulative distance for each trajectory are kept the same as in Figure 3.2(a). All lane changes were accurately identified by our algorithm showing the accuracy of the RRH obtained from a past trajectory using the modified RRH generation algorithm. The start and stop of each of the lane changes are clearly identifiable in Figure 3.2(b). These results indicate that lane departures can be accurately detected irrespective of whether the RRH is generated from a Google route or a past trajectory for straight portions of the road.

Due to the robustness and simplicity of our developed LDWS, we have also designed a smartphone app incorporating the idea that RRH can either be generated from Google routes or past trajectories to detect lane departures, which is explained in detail in the next chapter.

Chapter 4: Development of an App to Detect Lane Departure in Real Time

This chapter highlights the incorporation of the modified RRH generation (from both Google routes and past trajectories) and LDD algorithms in a smartphone app that can detect an unintentional lane departure to warn the driver in real-time using an audible warning. Both functionality of the smartphone app and the browser application depend upon each other as described below.

4.1 Interconnection between the Smartphone App and the Browser Application

The interdependence of the functionalities of the smartphone app and the browser application is shown in Figure 4.1. The backend browser has the ability to generate RRH both from a Google route or a past trajectory. The GPS receiver acquires the longitude and latitude of a moving vehicle in real-time and uploads the trajectory to the browser application to be later used by the RRH generation algorithm as shown in Figure 4.1.

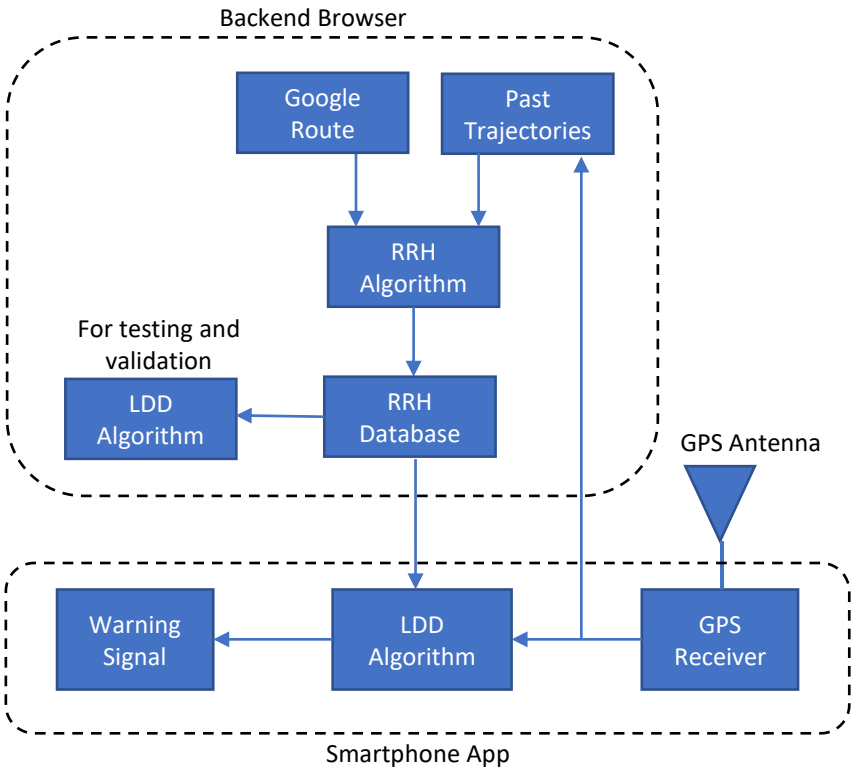


Figure 4.1 System architecture demonstrating the interconnection between the smartphone app and the backend browser-based application.

Please note that the RRH generation algorithm runs on a cloud server which will be explained later. The RRH generation algorithm uses either the past trajectories or Google routes to generate an RRH for a given road. While generating RRH from a past trajectory, the algorithm uses an adequate length of the GPS trajectory acquired by a smartphone for a given road. On the contrary, the user specifies the start and end location points of the Google route to be used to generate RRH using the backend browser application. Our backend browser application uses the Google Maps interface to provide the user with an easy method to choose the start and end location points of a desired route to generate RRH. In fact, our browser application can provide a user the opportunity to generate RRH for all potential roads in a particular area of interest so that an RRH is available on any road in that area, even if no prior past trajectory exists. Therefore, a smartphone riding in a vehicle can use our developed app to detect and warn about unintentional lane departures on any road, regardless of whether the vehicle is traveling on that road for the very first time.

All RRH generated by either past trajectories or by a user defined Google route are stored in a common database (Figure 4.1). This database is easily accessible to both the smartphone app and the browser application to extract the RRH of the desired road for lane departure detection. While lane departure detection is performed in real time in the smartphone app, we have also provided the lane departure detection ability in the browser application for testing and validation purposes (Figure 4.1).

4.2 Architecture of the Smartphone App and RRH Database

The architecture of the smartphone app and the RRH database residing in the cloud server of the browser application is shown in Figure 4.2. We have selected Google Cloud Platform (GCP) as our cloud server, which contains two databases. The first database is to store all uploaded past trajectories or a user defined Google route for RRH generation, and the second database accommodates the RRH of all available roads. Figure 4.2 describes the architecture of GCP containing the two databases and its interconnection with the smartphone app for RRH extraction and retrieval. These databases will be described in more detail later.

The smartphone app can upload a trajectory to the Google cloud server for RRH generation as well as retrieve the needed RRH of a given road to implement the LDD algorithm to detect a potential lane departure to warn the driver. In addition to that, the smartphone app must establish a secure connection with the cloud server to manage the extraction and retrieval of RRH for multiple roads (Figure 4.2). These three features of the smartphone app are described below.

4.2.1 Secure Connection

The first step for any smartphone running the app is to securely connect to GCP to successfully share the road trajectory and obtain the RRH of any road from the cloud database. In order to do that, the smartphone must exchange authentication credentials with the GCP each time it connects to it.

While launching the app on a smartphone for the first time, the app requires the user to create a username, which can be anonymous for privacy concerns. The authentication credentials are associated

with the username so that it can use those credentials to establish a connection with GCP for future use. The block showing “Firebase Auth” in the Google cloud server of Figure 4.2 is responsible for such an authentication process between the smartphone app and GCP.

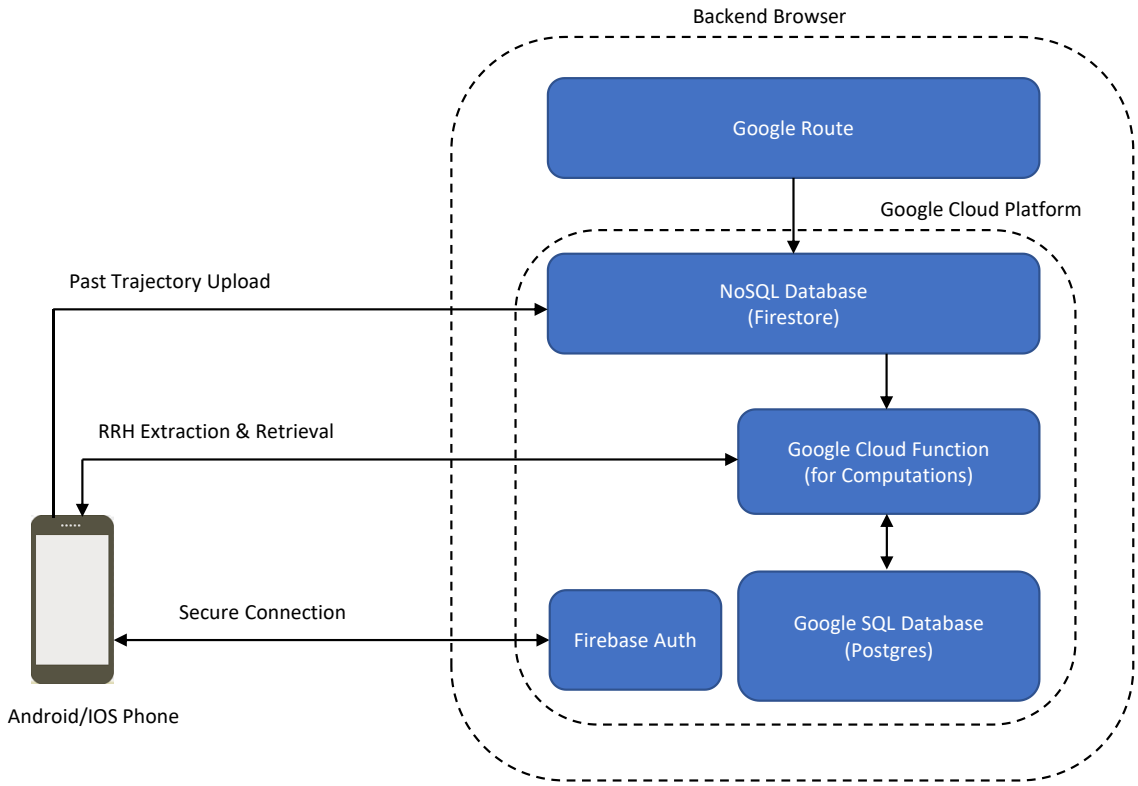


Figure 4.2 Schematic architecture of the smartphone app and its interconnection with GCP containing app database.

4.2.2 Trajectory Upload

Once a smartphone establishes a secure connection with GCP, it can upload its trajectory to GCP, which will reside in a “NoSQL” database called “Firestore” provided by the GCP.

A NoSQL database is appropriate for accommodating a large amount of data without much structure. Such a database is mainly used to house a large amount of raw data, especially if the data are temporary in nature. In our case, any road trajectory from even a short trip can contain a large volume of data which is only temporarily needed for RRH extraction and can be discarded later after successful extraction of each RRH. The nature of the data can be a good estimation tool for the amount of data present in a raw trajectory. Any trajectory data consists of a collection of GPS data points where each point is represented as a snippet of data, as shown in Figure 4.3. These snippets of data are stored in Google Firestore (NoSQL database). The GPS device can generate up to 10 such points or 10 data snippets every second while the smartphone travels along a road producing a large amount of data.

```
{
  // Provided by GPS chip on phone.
  "accuracy": 3,
  "heading": 347,
  "latitude": 38.74422308172889,
  "longitude": -77.19622757445973,
  // Provided by Google Roads API
  "googleLatitude":
38.74421031329726,
  "googleLongitude": -
77.19629289235164,
  // Generated by our app.
  "isManipulated": true,
  "timeStamp": "2021-06-19
00:03:42.022Z"
}
```


Figure 4.3: Snippet of data stored in JSON format.

4.2.3 RRH Extraction and Retrieval

As soon as a new trajectory for any given road is uploaded in the Firestore database, it is converted to RRH using the modified RRH generation algorithm. The piece of actual code for RRH extraction algorithm runs inside the cloud server using Google Cloud Function (Figure 4.2) which is a computation service provided by the Google cloud server. Google Could Function retrieves raw trajectory data from Firestore database and converts it into a useful RRH. It also saves the RRH in a structured database called "Postgres" (lower right block in Figure 4.2). Whenever a smartphone needs an RRH for any given road, it communicates with the Google Cloud Function, requesting the required RRH by sending its current location. If the relevant RRH is already present in the Postgres database, the cloud function retrieves it and sends it back to the smartphone, which is used for lane departure detection using the modified LDD algorithm.

The smartphone app database structure is discussed in detail in the following section.

4.3 App Database Structure

The app database structure development is the most crucial milestone toward the successful development of the smartphone app. The app needs to retrieve the corresponding RRH for any given road from the app database using the vehicle's current position to detect and warn the driver of any unintentional lane departure. This situation is analogous to the initial locking mechanism of a GPS receiver in which a vehicle (with a GPS receiver) is pinpointed at its current location on the map from an extensive mapping database. Similarly, for a vehicle traveling on a particular road for the very first time, our database structure has the ability to expand itself to accommodate the new RRH as well as to keep the provision for updating each existing RRH if and when more future trajectories are available for any given road which are already part of the database. Furthermore, the algorithm to extract RRH from past trajectories applies to large tracts of vehicle trajectories. However, sometimes trajectories include unnecessary portions of data like turns on highways and entrances/exits on freeways which require

exclusion from the RRH before making it a part of the database. This process demands developing a structured app database to format and store RRH for multiple roads in one place. Our database also has the provision of showing whether a Google RRH for that road is available or not.

Although the app database can reside in the memory of a smartphone, we opted for a cloud service, i.e., GCP, to accommodate the app database structure to allow multiple users to have access to the RRH generated by either any participating smartphone app user's past trajectory or by browser application from Google routes. The browser app is not intended to be run by individual users rather it should be used for planning purposes to create RRH for all roads in a given area.

Please note that the RRH generation algorithm will run on the cloud server, and the LDD algorithm will run on the smartphone. Therefore, an RRH generated by one vehicle can be potentially used by another vehicle for lane departure detection. On the other hand, if past trajectories are not available, RRH can also be generated from Google routes. The option of generating RRH from both past trajectories and Google routes is a good performance measure to evaluate how well RRH is being generated from Google routes when compared to past trajectories. As a result, the Google RRH can be further improved by fine-tuning the RRH generation algorithm so that RRH generated from both Google routes and past trajectories are comparable for all portions of the road.

The app database is structured into two separate databases, one for past trajectories upload (which has been described in Section 4.2.2) and one for RRH storage, which is described below.

4.3.1 RRH Database

The RRH database is the most crucial element of the app database structure because it accommodates the RRH for all the roads. As stated before, the location of the RRH database is chosen in the GCP instead of the smartphone's memory to allow multiple app users to have access to extract RRH either from past trajectories or Google routes. This feature will be efficient when multiple users will use and test the app. Additionally, this provides a platform for commercial apps like Google Maps to integrate the app feature within their environment. The structure also allows the database to expand itself to accommodate the new RRH as well as to keep the provision for updating an existing RRH of a given road when more future trajectories are available for that road. Our database also has the provision of showing whether a Google RRH for that road is available or not.

Due to its highly structured nature, a SQL database called Postgres (bottom right block in Figure 4.2) is used, which is accessible within the GCP. This database contains RRH for all road sections, along with the road name and other relevant parameters. The relevant parameters of each section of the road are the start and end points (longitude and latitude), path average heading (PAH) for the straight sections, and initial heading (IH), and path average heading slope (PAHS) for the curve or transition sections. Whenever a vehicle needs to retrieve an RRH for a given road, it sends a query to the Postgres database to retrieve the required RRH using the road name and the position of the two end points of various sections of any road present in the database. The RRH of each section of the road also contains a parameter called "degree of confidence," or DoC, which has an integer value indicating the number of

times a particular RRH has been previously updated. For example, a DoC value of “1” implies that only one trip generated the RRH for that road. A DoC value of 2 or 3 means that two or three past trajectories have been used for that RRH. Multiple trajectories can be either from the same vehicle traveling on that road at different times or by different vehicles traveling on the same road either at the same time or at different times.

A typical road structure containing two freeways (FW1 and FW2) is shown in Figure 4.4 to explain the RRH database (SQL database). Four different trajectories are also shown in Figure 4.4 using colored dashed lines.

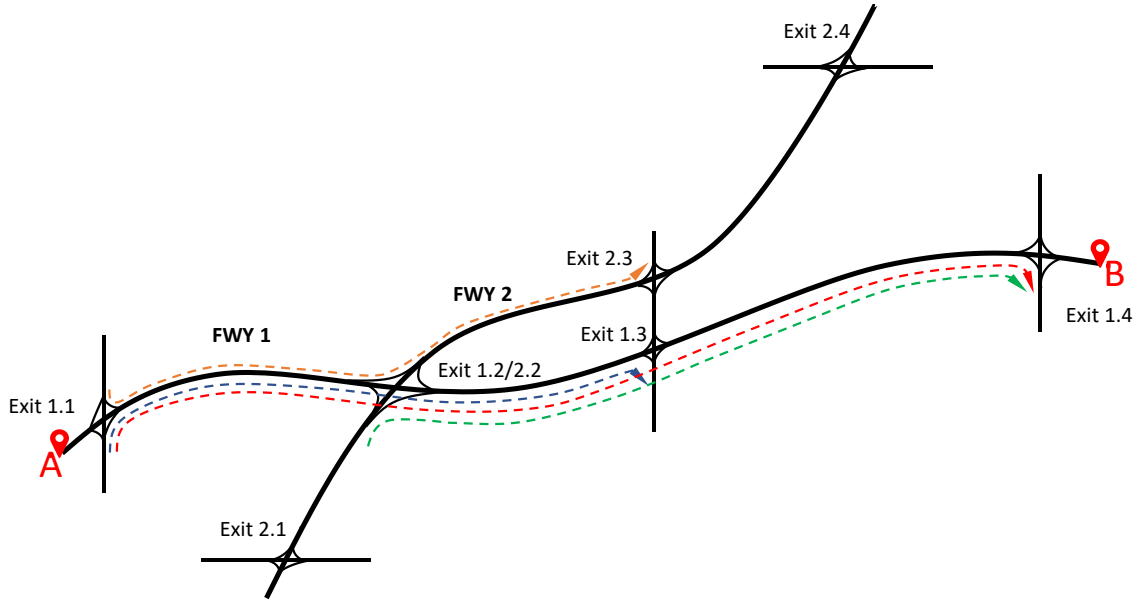


Figure 4.4 A typical road infrastructure containing two freeways to illustrate RRH database structure. The colored dashed lines are trajectories on that road obtained at different times. Points A and B on FW1 indicate the start and end points of a Google route for which an RRH has been generated.

The points A and B on FW1 (Freeway 1) indicate the start and end points of a Google route for which an RRH has been generated using the browser application.

After a trajectory is uploaded in the NoSQL database (Firestore), it is processed by the Google Cloud Function to generate RRH using the RRH generation algorithm for that road segment. After generating an RRH, the Google Cloud Function stores all sections of RRH in the SQL database (Postgress). The structure of this database is shown in Table 1, with a grey highlighted area. The first three columns of the table (not highlighted) are not part of the database but have been included to explain the process of updating the RRH in the database as more trajectories are available.

After the first trajectory (blue dashed line in Figure 4.4) is uploaded and converted to RRH, all the relevant parameters of each section of the RRH are stored in the database along with the road name and DoC value. In this case, the road name is FW1, and DoC is 1 for all sections. After the RRH resulting

from this trajectory is saved in the database, a total of 7 entries are made corresponding to 7 sections, as can be seen in Table 1.

When the second trajectory (red dashed line in Figure 4.4) is available and converted to an RRH, the database entries are updated (Table 1). As seen in Figure 4.4, the new trajectory (red dashed line) is simply an extension of the first trajectory (blue dashed line). Therefore, the new RRH contains some of the old sections for the part of the road from the first trajectory as well as some new sections for the new part of the road. In this case, the first seven sections (already present) are updated, and two more new sections are added, as shown in Table 1 (Trajectory #2). If a section is updated, its DoC value is increased by one and for the new sections, it will become 1. Please note that the last section or section #7 resulting from the first trajectory may get modified when RRH sections are generated from the second trajectory because the last section (either curve or straight) may get extended and the endpoint may differ from the previous point. For that purpose, this section will be either updated or remain the same depending upon how long the extension is. As a rule of thumb, if it is not extended by more than 20% in length, we will not consider this as a different section and increase the DoC value by one. However, if it gets extended by more than 20%, we will call it a different section and keep the Doc value to 1. In the future, we will also consider setting some margins for other parameters (PAH or IH and PAHS) of a given section of the RRH to decide on updating the existing section.

While the database is slowly populated with more RRH from past trajectories as more trajectories are available, the database can be augmented with RRH generated from Google routes for any or all roads in a given area using our browser application. The column “Google RRH” in Table 4.1 has values of “y”, “n” or blank. A value of “y” indicates that RRH from a Google route for that road segment exists and “n” indicates that it does not exist yet. The rows which are left blank are in fact, a subset of the road for which an RRH exists from the Google route. Therefore, it is not necessary to indicate that Google RRH exists for those. For example, Trajectories 1, 2 and 3 are all part of FW1, however, both Trajectories 1 and 3 are subset of Trajectory 2 (Figure 4.4), implying if an RRH from Google route is obtained for Trajectory 2, that will cover both Trajectories 1 and 3. Therefore, the entries in the Google RRH column in Table 4.1 for Trajectories 1 and 3 are left as blanks. In reality, RRH using the Google route can cover a much longer trajectory, e.g., even in the illustration of Figure 4.4, the Google route from which an RRH is extracted is longer than Trajectory 2.

Similarly, Trajectory 4 constitutes of a route that has portions from both FW1 and FW2 (Figure 4.4). However, in this example, an RRH from the Google route for FW2 is not obtained yet, therefore, the entries of Google RRH column in Table 4.1 for the Trajectory 4 has a “n” value.

Table 4.1 RRH database showing updated database after every new trajectory.

Reference Itenerary	Count	Comment	Road Name	Google RRH	DoC	Start Lat	Start Long	End Lat	End Long	PAH	IH	PAHS
	1	new	FW1		1	x	x	x	x		x	x
	2	new	FW1		1	x	x	x	x		x	x
	3	new	FW1		1	x	x	x	x	x		
	4	new	FW1		1	x	x	x	x		x	x
Trajectory #1	5	new	FW1		1	x	x	x	x		x	x
(blue dashed line)	6	new	FW1		1	x	x	x	x	x	x	x
from Exit 1.1. to 1.3	7	new	FW1		1	x	x	x	x	x		
	1	update	FW1	y	2	x	x	x	x		x	x
	2	update	FW1	y	2	x	x	x	x		x	x
	3	update	FW1	y	2	x	x	x	x	x		
	4	update	FW1	y	2	x	x	x	x		x	x
	5	update	FW1	y	2	x	x	x	x		x	x
Trajectory #2	6	update	FW1	y	2	x	x	x	x	x	x	x
(red dashed line)	7	update*	FW1	y	1 or 2	x	x	x	x	x		
from Exit 1.1 to Exit	8	new	FW1	y	1	x	x	x	x		x	x
1.4	9	new	FW1	y	1	x	x	x	x		x	x
	1		FW1		2	x	x	x	x		x	x
	2		FW1		2	x	x	x	x		x	x
	3	update*	FW1		2 or 3	x	x	x	x	x		
	4	update	FW1		3	x	x	x	x		x	x
	5	update	FW1		3	x	x	x	x		x	x
	6	update	FW1		3	x	x	x	x	x	x	x
Trajectory #3	7	update	FW1		2	x	x	x	x	x		
(green dashed line)	8	update	FW1		2	x	x	x	x		x	x
from Exit 1.2 to 1.4	9	update	FW1		2	x	x	x	x		x	x
	1	update	FW1		3	x	x	x	x		x	x
	2	update	FW1		3	x	x	x	x		x	x
	3	update*	FW1		3 or 4	x	x	x	x	x		
	4		FW1		3	x	x	x	x		x	x
	5		FW1		3	x	x	x	x		x	x
	6		FW1		3	x	x	x	x	x	x	x
	7		FW1		2	x	x	x	x	x		
	8		FW1		2	x	x	x	x		x	x
	9		FW1		2	x	x	x	x		x	x
Trajectory #4	10	new	FW2	n	1	x	x	x	x	x		
(orange dashed line)	11	new	FW2	n	1	x	x	x	x	x		
from Exit 1.1 to 2.3	12	new	FW1	n	1	x	x	x	x		x	x

4.4 App User Interface

Once our app is installed on a smartphone and started a pop-up screen will appear asking for permission whether the app can access the device's location using the device's GPS receiver. Figure 4.5a shows a screenshot where our app asks the user to choose if the user wants to share a precise or approximate location with the app. The user will have three options to choose from: "While using the app," "Only this time," or "Don't allow." The user's preferences will be saved once and will not be shown to the user again, although a user can change these settings anytime using the settings menu. To use the app, a user has to choose from the first two options.

After saving the location preferences, sign-up will be required to establish a secure connection to extract the RRH for a given road from GCP. The sign-up page is shown in Figure 4.5b, where users can register using a valid email and a password of their choice. These credentials will be required to sign into the app for subsequent use. The app must be active in the foreground to take the GPS data and for the lane departure detection algorithm to work. Once the user sign-in into the app and it is running in the foreground, the vehicle's current position will appear on a Google map on the smartphone app screen as a blue dot, as shown in Figure 4.5c. Typically, a comfortable view of about ± 100m around the vehicle's position is shown on the map of the app screen. Users can zoom in or zoom out according to their comfort level.

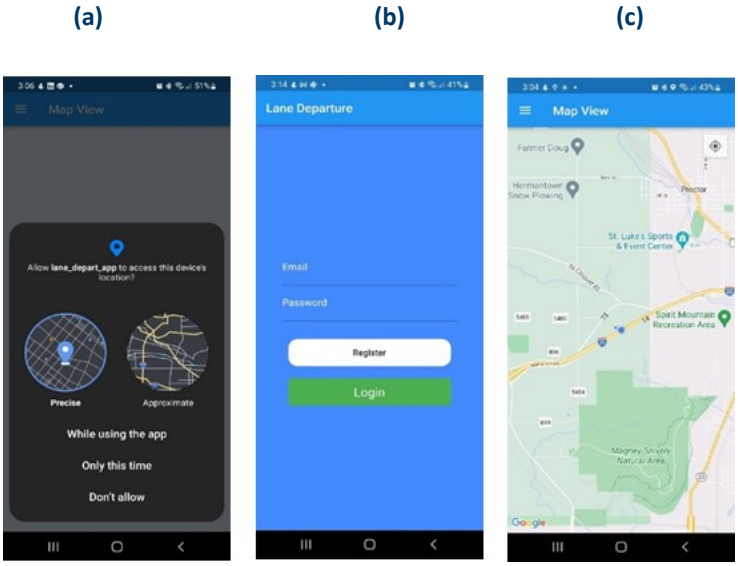


Figure 4.5 (a) Screenshot of the app seeking user permission to access the device's location. (b) Screenshot of the registration page of the app. (c) Screenshot of the app showing the device location on the map.

While traveling on the road with the app running in the foreground, the LDD algorithm will be active and be able to detect any lane departure and issue an audible warning. In the future, when the app gets integrated into a vehicle's navigation system, the app will be able to differentiate between intentional

and unintentional lane departures using the turn signal indicator. For now, our app will issue an audible warning for all lane departures, including intentional lane changes.

To demonstrate the app's functionality in the field, we drove the smartphone running our app in a vehicle on a 7 km segment of Interstate I-35 near Duluth, MN, for which an RRH was already generated from a previous trip. The test vehicle was driven within the speed limit (70 MPH) on the 4-lane freeway (2 lanes each way), and many back-and-forth lane changes were made intentionally in each test run. For safety reasons, intentional lane changes were made to test the accuracy of lane departure detection using the LDD algorithm. The app successfully identified the lane changes made during the test run and issued an audible warning. The detection is completed within a few GPS cycles, where each GPS cycle is 100 ms. Normally, the lane departure is detected for a little less than a second. The accuracy of our lane departure on any given road depends upon the accuracy of the RRH of that road. On curved portions, the accuracy may not be as much as on straight portions. However, unintentional lane departures occur mainly on long stretches of straight road portions as opposed to curved road sections where the driver must be attentive anyway.

(a)

(b)

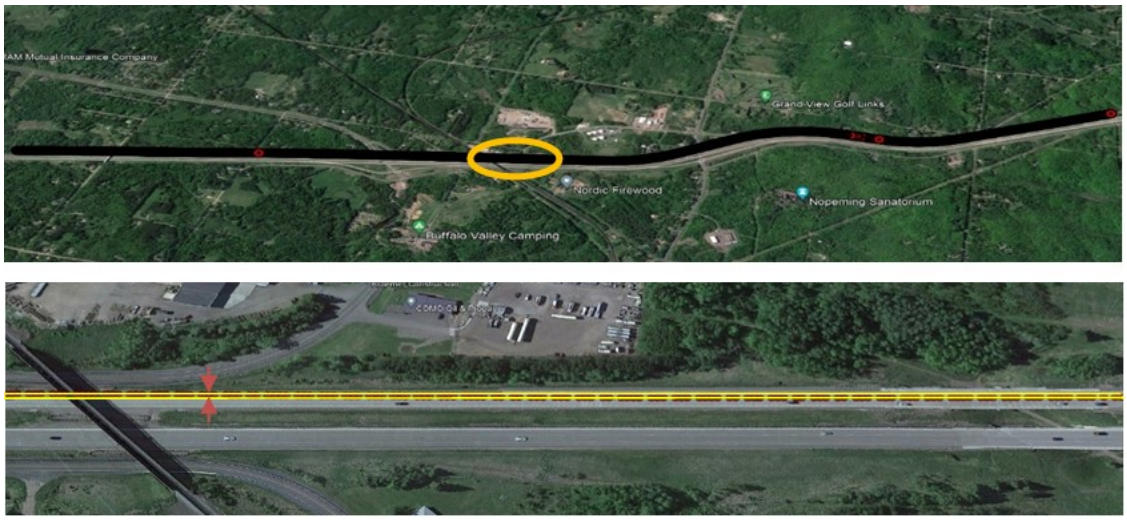


Figure 4.6 Picture from the Google Map of the 7 km route on Interstate I-35 with one lane departure highlighted with a yellow circle. Below is a zoomed in picture of the lane departure portion circled in yellow indicating the lane width with yellow lines. The lane change inside the yellow circle was made from left to right.

Figure 4.6a depicts a picture from Google Maps, showing a typical trajectory of the test run with a lane change circled in yellow. The zoomed-in portion of this part of the trajectory with a lane change (from left to right) is shown in Figure 4.6b. Please note that the trajectory with GPS points is shown as a continuum of small circles laterally shifting from left to right for a lane change. The lateral shift is approximately one lane width, as illustrated in Figure 4.6b, with two yellow lines. This example is only to demonstrate the smartphone app user interface, so an RRH using the Google route for that trajectory is not discussed here.

4.5 Scope of the App

We have written the code of the smartphone app in a language (Dart) which can be used to compile the code for both Android and iOS platforms. However, the current version of the developed app cannot be used for iOS platform (Apple devices) because we have to use an external GPS receiver for the app to work.

After we developed the initial app for collecting data for testing purposes, we faced a problem that a smartphone's GPS receiver switches to low frequency after a short while in any dataset (vehicle trajectory) we collect. The GPS receiver normally operates at 10 Hz i.e., location data can be accessed 10 times in a second or once every 100 msec. That is the case with every GPS receiver including the GPS receivers in any navigational device or a smartphone including both Android and iOS devices. However, when GPS receiver is integrated in a smartphone and is controlled with its operating system e.g., Android or iOS, it imposes extra restriction on the frequency of GPS operation to save battery life. Both Android and Apple iOS impose extra restrictions in controlling the frequency at which their GPS receivers can be accessed. They restrict it to 1 Hz as opposed to 10 Hz. This is not a fundamental restriction, but they do it only to prolong the battery life. They keep this restriction even if the battery is on a charger.

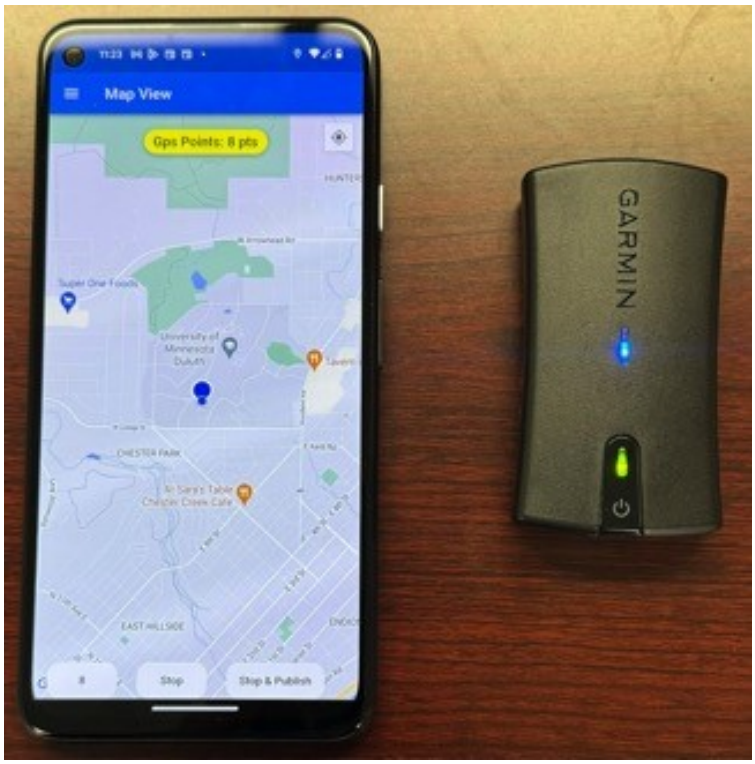


Figure 4.7 External GPS receiver device along with the Android phone.

For our algorithms to work properly, the GPS data should be accessed at 10 Hz frequency so when the smartphone forces the GPS data to run only at 1 Hz, that gives a very coarse trajectory data which

affects the performance and outcome of both of our algorithms (RRH and LDD). There are ways to set the frequency of GPS data acquisition to 10 Hz in Android phones and we have tried many such methods but soon after a few seconds of operation, it converts back to 1 Hz frequency. We have tried multiple ways to force the Android phone to give us 10 Hz data but every time, we only prolong it for a little more time and then it switches back to 1 Hz. Additionally, we artificially interpolated data to a higher frequency before we applied our algorithm of RRH generation. However, this solved only half of our problem i.e., only RRH generation algorithm works to some extent with this technique, but LDD algorithm is useless with the low frequency or coarse trajectory data.

After spending so much time and trying a variety of methods, we finally decided to use an external GPS device with the smartphone which can give us better control over the frequency of the GPS data. These external GPS devices can be interfaced with the smartphones either using a wireless (Bluetooth) or wired (USB) connection. We have identified and purchased one such wireless GPS device to use as external GPS receiver. This device is shown in Figure 4.7 along with the Android smartphone used for testing purposes.

This plan of action has two negative consequences. The main negative consequence is that we are not able to deploy our app in everyone's smartphone unless we provide that device as well. That means that our app cannot be downloaded and used by anyone as this device is necessary for the current version of our app to run. The second negative consequence is that our app does not work for Apple devices for now as iOS platform prohibits the use of accessing GPS data from an external GPS receiver. Although, we have written the smartphone app in a language (Dart) which can be compiled to run in both Android and iOS devices, we can't run it on Apple smartphones because of the need to access GPS data from an external device. Our initial version of the app which was prepared in Task 3 and used only to collect data was able to run for both Apple and Android smartphones but only collected data at 1 Hz frequency. However, now when we have LDD algorithm running in the smartphone app accessing GPS data at 10 Hz frequency from an external GPS receiver, it is only valid for Android devices.

In spite of the negative consequences, this does not change the end goal of the project. We are still able to develop a marketable app which can be used to secure future partnerships e.g., Google to adapt this as a feature in their Google Maps. If Google is on board, they could easily make changes in their operating system to let the GPS operate at 10 Hz eliminating the reliance on any external device. The use of external device is only an intermediate step for us to demonstrate the power/feature of our app in absence of not being able to control the operating system of the smartphones.

4.6 Demo Link

We have prepared a demo video to illustrate the functionality of the app. The demo video is a combination of the smartphone app screen recording while the app is running and a dashboard video recorded by another phone to see the correlation between actual lane change and the timing of the smartphone app warning. The test drive for the demo video was on about a 4-mile-long segment of I-35 SB near Duluth, MN. The test vehicle is shown as a moving blue dot on the smartphone app screen

recording. It changes to red when a lane is being changed. The smartphone app also emits a sound warning during the lane change. There are a total of 6 lane changes during the two-and-a-half-minute video. The start and end of each section of the road (straight, curved, or transitioning) is shown with a location symbol (Figure 4.8). The color of the location symbol indicates whether the vehicle is traveling on a straight (red), curved (blue), or transitioning (green) section. We also repeated the demo on I-35 NB which is captured in the second demo video. There are a total of 4 lane changes in the second demo video. Please note that both the demo videos also show a curve ahead warning for the upcoming curves which can be seen on the top of the smartphone app screen recording. For now, the demo app shows the curve ahead warning for all curves regardless of a curve's length and the degree of curvature. However, in the future, we will limit this warning to only those curves that are long enough and have a significant degree of curvature.

Figure 4.8 Location symbol used in the demo video.

Chapter 5: Expected Benefits, Future Work and Conclusions

This chapter highlights expected benefits of the app and future work followed by the conclusions.

5.1 Expected Benefits and Implementation Steps

The major goal of the project was to develop a marketable smartphone app using our two previously developed algorithms using a standard GPS receiver to detect and unintentional lane departure and warn the driver in real time. This goal has been largely achieved with some limitations. As explained earlier, our developed smartphone app depends upon an external GPS device to function. As a result, the current smartphone app works only for Android based devices. In spite of this, the concept of the app is very useful and the developed smartphone all could potentially benefit Minnesota drivers in many ways including the following:

1) **Cost Advantage:** Currently, almost all commercially available LDWSs use some kind of camera or vision-based sensor to detect an unintentional lane departure. Such systems periodically take pictures of lane markings and compare those pictures frame by frame to estimate unintentional lateral lane departure which makes systems quite complex and costly to implement. Therefore, such systems are usually present in the high-end trim versions of a vehicle and many economical vehicles do not have such systems. In comparison, our proposed system is GPS based and can run as a smartphone app incurring no additional cost to the user.

2) **Performance Advantage:** In addition to being costly, most commercially available LDWSs have some performance issues especially when lane markings are worn out, covered with snow, or if lighting conditions are not favorable. However, our solution does not depend upon vision sensor or lane markings, and therefore, performs the same in all conditions. Please note that our smartphone app may not perform well either around large buildings e.g., in an urban downtown area because the performance of a standard GPS is adversely affected by the vicinity of large buildings due to multipath signal reflection and interference. Please note that our smartphone app is not being recommended for an urban environment especially where the speed limit is less than 40 mph. The real advantage of our lane departure smartphone app can be reaped on a freeway or a long stretch of a rural road with a higher speed limit.

3) **Accessibility Advantage:** Another key advantage of our developed smartphone app is that it can be made accessible to anyone with a smartphone whereas a commercially available LDWS is locked to a certain vehicle. We do realize that many people may forget or simply may not prefer to have an extra smartphone app to remain open on their smartphones while driving. This problem can be alleviated if Google Inc. can adapt our smartphone app as a feature of their Google Maps app to truly unlock the accessibility advantage of our smartphone app. Similarly,

our smartphone app can be added as a feature in Apple Maps as well as in any GPS navigational device including Garmin and TomTom navigation devices. Furthermore, this can also be integrated in a vehicle's built in user interface to control the vehicle's other features.

Our future work will mainly consist of efforts towards unlocking the accessibility advantage of our developed smartphone app. Initial contacts regarding this have already been made with Google Inc. through personal contacts. In addition to that, the Office of Trademark and Commercialization (OTC) at the University of Minnesota is also aware of this approach and is willing to help. However, before securing a serious engagement with Google, there is some more work needed to make the smartphone app more robust so it can function seamlessly on more than a few exemplary roads. Although this project is formally ending, there is work in progress to bring the smartphone app to the level that can convince Google about the true benefits of our developed smartphone app.

5.2 Conclusions

We have successfully developed a backend browser application to extract RRH from either a Google route or a vehicle's past trajectory. Our backend browser stores RRH of various roads in a database residing in a cloud server. We have also developed a smartphone app using our LDD algorithm that uses RRH of a given road and can be accessed from the database of the cloud server. We have conducted field testing using the RRH generated by our backend browser and LDD algorithm running in our smartphone app. Our test results indicate that our smartphone app can accurately detect a lane departure irrespective of whether the RRH is generated from a Google route or a past trajectory, especially for the long stretches of straight portions of the road. We will continue to make some tweaks in the RRH algorithm to further improve it while we also try to secure future partnerships.

References

1. WHO. (2022, June). *Road traffic injuries*. Geneva: World Health Organization (WHO).

2. Singh, S. (2015). *Critical reasons for crashes investigated in the national motor vehicle crash causation survey* (Report No. DOT HS 812 115). Washington, DC: NHTSA.

3. Antony, M. M, & Whenish, R. (2021). Advanced driver assistance systems (ADAS). In *Automotive embedded systems (*pp 165–181). Location: Springer.

4. Sternlund, S., Strandroth, J., Rizzi, M., Lie, A., & Tingvall, C. (2017).The effectiveness of lane departure warning systems—A reduction in real-world passenger car injury crashes. *Traffic Injury Prevention Journal*, *18,* 225–229.

5. An, X., Wu, M., & He, H. (2006). A novel approach to provide lane departure warning using only one forward-looking camera. *International Symposium on Collaborative Technologies and Systems*, 356–362.

6. Baili, J., Marzougui, M., Sboui, A., Lahouar, S., Hergli, M., Bose, J. S. C., & Besbes, K. (2017). Lane departure detection using image processing techniques. *International Conference on Anti-Cyber Crimes* (ICACC), 238–241.

7. Chen, Y., & Boukerche, A. (2020). A novel lane departure warning system for improving road safety. *ICC 2020-2020 IEEE International Conference on Communications (ICC)*, 1–6.

8. Hsiao, P. & Yeh, C.-W. (2006). A portable real-time lane departure warning system based on embedded calculating technique. *IEEE 63rd Vehicular Technology Conference*, 2982–2986.

9. Jung, H., Min, J., & Kim, J. (2013). An efficient lane detection algorithm for lane departure detection. *IEEE Intelligent Vehicles Symposium (IV)*, 976–981.

10. Leng, Y. C., & Chen, C. L. (2010). Vision-based lane departure detection system in urban traffic scenes. *11th International Conference on Control Automation Robotics & Vision*, 1875–1880.

11. Yu, B., Zhang, W. & Cai, Y. (2008). A lane departure warning system based on machine vision. *IEEE Pacific-Asia Workshop on Computational Intelligence and Industrial Application*, 197–201.

12. Lindner, P., Richter, E., Wanielik, G., Takagi, K., & Isogai, A. (2009). Multi-channel lidar processing for lane detection and estimation. *12th International IEEE Conference on Intelligent Transportation Systems*, 1–6.

13. Chen, W., Wang, W., Wang, K., Li, Z., Li, H., & Liu, S. (2020). Lane departure warning systems and lane line detection methods based on image processing and semantic segmentation: A review. *Journal of Traffic and Transportation Engineering*, *7*(6), 748–774.

14. Gamal, I., Badawy, A., Al-Habal, A. M., Adawy, M. E., Khalil, K. K., El-Moursy, M. A., & Khattab, A. (2019). A robust, real-time and calibration-free lane departure warning system. *Microprocessors and Microsystems*, 71, 102874.

15. Dobler, G., Rothe, S., Betzitza, P., & Hartlieb, M. (2000). *Vehicle with optical scanning device for a lateral road area*. Google Patents. Retrieved from

16. McCall, J. C., & Trivedi, M. M. (2006). Video-based lane estimation and tracking for driver assistance: Survey, system, and evaluation. *IEEE Transactions on Intelligent Transportation Systems*, *7*(1), 20–37.

17. Son, J., Yoo, H., Kim, S., & Sohn, K. (2015). Real-time illumination invariant lane detection for lane departure warning system. *Expert Systems with Applications*, *42*(4), 1816–1824.

18. Sang, I. C., & Norris, W. R. (2024). A robust lane detection algorithm adaptable to challenging weather conditions. Retrieved from

19. Bajikar, S., Gorjestani, A., Simpkins, P., & Donath, M. (1997). Evaluation of in-vehicle GPS-based lane position sensing for preventing road departure. *Proceedings of Conference on Intelligent Transportation Systems*, 397–402.

20. Toledo-Moreo, R., & Zamora-Izquierdo, M. A. (2009). IMM-based lane-change prediction in highways with low-cost GPS/INS. *IEEE Transactions on Intelligent Transportation Systems*, *10*(1), 180–185.

21. Clanton, J. M., Bevly, D. M., & Hodel, A. S. (2009). A low-cost solution for an integrated multisensor lane departure warning system. *IEEE Transactions on Intelligent Transportation Systems*, *10*(1), 47–59.

22. Weon, I. S., Lee, S. G., & Woo, S. H. (2021). Lane departure detecting with classification of roadway based on Bezier curve fitting using DGPS/GIS. *Tehnički Vjesnik*, *28*(1), 248–255.

23. Rose, C., Britt, J., Allen, J., & Bevly, D. (2014). An integrated vehicle navigation system utilizing lane-detection and lateral position estimation systems in difficult environments for GPS. *IEEE Transactions on Intelligent Transportation Systems*, *15*(6), 2615–2629.

24. Shahnewaz Chowdhury, M. T. H., & Hayee, M. I. (2021). Generation of road reference heading using GPS trajectories for accurate lane departure detection.

25. Faizan, M., Hussain, S., & Hayee, M. I. (2019). Design and development of in-vehicle lane departure warning system using standard global positioning system receiver. *Transportation Research Record*, *2673*(8), 648–656

Appendix A:
Code Used for Implementation of RRH Generation Algorithm for Web Browser Tool

Please note that the entire code for the web browser tool is a few thousand lines because it has to accommodate data management and user interface in addition to the two algorithms, RRH generation algorithm and LDD algorithm. This code is written in Typescript (Angular) language. Below is the specific code used for RRH generation algorithm in web browser tool.

```
import { Snapshot } from "./Snapshot";
import { Section, SectionType, SectionRectangle } from "./Section";
import { headingTo, normalizeHeading, headingDistanceTo } from "geolocation-utils";

const EARTH_RADIUS = 6378137;

var dsp = require('digitalsignals');

export class ProcessedRouteWrapper {
	UserId: string;
	RouteId: string;
	SortedSnapshots: Snapshot[];

	Distances: number[] = [];
	AverageDistances: number[] = [];
	Latitudes: number[] = [];
	Longitudes: number[] = [];
	SmoothedHeading: number[] = [];
	OutHeadings: number[] = [];
	Slopes: number[] = [];
	Accuracies: number[] = [];
	AccumulativeDistances: number[] = [];
	AverageAccumulativeDistances: number[] = [];
	AverageHeadings: number[] = [];
	DifferentialHeadings: number[] = [];
	cutOffFrequency2: number;
	cutOffFrequency1: number;
	StraightSections: Section[] = [];
	StraightSectionsTH2: Section[] = [];
	AveragedDifferentialHeadings: number[] = [];
	PathAveragedHeadings: number[] = [];
	AllSections: Section[] = [];

	constructor(userId: string, routeId: string, cutOffFrequency1: number, cutOffFrequency2:
number, snapshots: Snapshot[], useGooglePoints: boolean = true) {
		this.UserId = userId;
		this.RouteId = routeId;
		this.cutOffFrequency1 = cutOffFrequency1;
		this.cutOffFrequency2 = cutOffFrequency2;
		this.SortedSnapshots = snapshots;

		this.ProcessRoute(useGooglePoints);
```

```
	}

	ProcessRoute(useGooglePoints: boolean)
	{
		// clean data abit (remove duplicated points etc)
		let sortedCompleteRoute: Snapshot[] = [];
		for (let i = 1; i < this.SortedSnapshots.length; i++)
		{
			if (AreSnapshotsOnSamePoint(this.SortedSnapshots[i - 1],
this.SortedSnapshots[i], useGooglePoints))
			{
				continue;
			}

			sortedCompleteRoute.push(this.SortedSnapshots[i]);
		}

		this.Distances.push(0);
		this.OutHeadings.push(0);
		this.Slopes.push(0);
		this.Accuracies.push(0);
		this.AccumulativeDistances.push(0);

		for (let index = 0; index < sortedCompleteRoute.length; index++)
		{
			// let index = i - 1;
			let currentSnapshot = sortedCompleteRoute[index];

			if (useGooglePoints)
			{
				this.Longitudes.push(currentSnapshot.GoogleLongitude);
				this.Latitudes.push(currentSnapshot.GoogleLatitude);
			}
			else
			{
				this.Longitudes.push(currentSnapshot.Longitude);
				this.Latitudes.push(currentSnapshot.Latitude);
			}

			if (this.Longitudes.length > 1) // Refactor later, basically make sure we have
atleast 2 clean points before calculating rest of the data.
			{
				let headingDistance = headingDistanceTo(
					{lat: this.Latitudes[index - 1], lon: this.Longitudes[index - 1] },
					{lat: this.Latitudes[index], lon: this.Longitudes[index]}
				)
```

```
                                        //this.Distances.push(headingDistance.distance)
                                        this.OutHeadings.push(headingDistance.heading + 360);
                                        this.Distances.push(headingDistance.distance)
                                        this.Accuracies.push(currentSnapshot.Accuracy);
                                        this.AccumulativeDistances.push(headingDistance.distance +
this.AccumulativeDistances[index - 1]);
                                }
                        }

                        const averageDistanceBetweenPoints =
this.AccumulativeDistances[this.AccumulativeDistances.length - 1] / (this.AccumulativeDistances.length
- 2);
                        for (let i = 0; i < this.AccumulativeDistances.length; i++) {
                                this.AverageDistances.push(averageDistanceBetweenPoints);
                                this.AverageAccumulativeDistances.push(averageDistanceBetweenPoints +
this.AccumulativeDistances[i - 1]);
                        }
                        this.Distances=this.AverageDistances;
                        this.AccumulativeDistances = this.AverageAccumulativeDistances;

                        for (let i = 0; i < this.OutHeadings.length; i++) {
                                if (i < 4) {
                                        this.SmoothedHeading.push(this.OutHeadings[i]); // smoothed heading
isn't averaged for first 4 points.
                                } else {
                                        this.SmoothedHeading.push((this.OutHeadings[i - 4] +
this.OutHeadings[i - 3] + this.OutHeadings[i - 2] + this.OutHeadings[i - 1] + this.OutHeadings[i] +
this.OutHeadings[i + 1] + this.OutHeadings[i + 2] + this.OutHeadings[i + 3] + this.OutHeadings[i + 4]) / 9 );
                                }
                        }

                        for (let i = 1; i < this.SmoothedHeading.length; i++) {
                                let slope = (this.SmoothedHeading[i] - this.SmoothedHeading[i - 1]) /
this.Distances[i];
                                this.Slopes.push(slope);
                        }

                        this.DifferentialHeadings.push(0);
                        for (let i = 1; i < this.SmoothedHeading.length; i++) {
                                this.DifferentialHeadings.push((this.SmoothedHeading[i] -
this.SmoothedHeading[i - 1])/ this.Distances[i]);
                        }

                        this.AveragedDifferentialHeadings =
CalculateAveragedDifferentialHeadings(this.DifferentialHeadings);
                        let threshold1 = 0.002;
```

```
		let straightSections =
GetStraightSections(this.AveragedDifferentialHeadings,threshold1);

		let threshold2 = 0.01;
		let straightSectionsth2 =
GetStraightSections(this.AveragedDifferentialHeadings,threshold2);

		const MinimumPointsBetweenStraightSections = 75;
		let previousStraightSectionPointer = 0;
		for (let i = 1; i < straightSections.length; i++) {
			const currentStraightSection = straightSections[i];
			const previousStraightSection =
straightSections[previousStraightSectionPointer];

			if (currentStraightSection.StartIndex - previousStraightSection.EndIndex <
MinimumPointsBetweenStraightSections) {
				previousStraightSection.EndIndex = currentStraightSection.EndIndex;
				// Remove last section from array
				delete straightSections[i];
			} else {
				// typescript deletes the object and sets the array[j] value with
undefined object so we need to handle that.
				for (let j = previousStraightSectionPointer + 1; j <
straightSections.length; j++) {
					if (straightSections[j] !== undefined) {
						previousStraightSectionPointer = j;
						break;
					}
				}
			}
		}

		straightSections.forEach(section => {
			let pathAveragedHeading = CalculatePathAveragedHeading(section,
this.SmoothedHeading, this.Distances, this.AccumulativeDistances);
			section.PathAveragedHeading = pathAveragedHeading;
			OptimizeStraightSection(section, this.SmoothedHeading, this.Distances); //
optimize straight sections
			this.AddSectionMetaData(section);
			this.StraightSections.push(section);
		});

		straightSectionsth2.forEach(section => {
			this.StraightSectionsTH2.push(section);
		});

		this.AllSections.push(this.StraightSections[0])
```

```
// Assume our path starts and ends at a straight section for now.
for (let i = 1; i < this.StraightSections.length; i++) {
    var currentStraightSection = this.StraightSections[i];
    var previousStraightSection = this.StraightSections[i - 1];

    let rawNonStraightSection = new Section(previousStraightSection.EndIndex,
currentStraightSection.StartIndex, SectionType.Unknown);

    // we need to process this section more to get the transient sections
    let pathAveragedDifferentialHeading1 =
CalculatePathAveragedDifferentialHeading(rawNonStraightSection, this.AveragedDifferentialHeadings,
this.Distances, this.AccumulativeDistances);
    let reSelectedSection1 =
PathAveragedDifferentialHeadingReselect(rawNonStraightSection, this.AveragedDifferentialHeadings,
pathAveragedDifferentialHeading1);
    let trueCurveSection= reSelectedSection1


    trueCurveSection.SectionType = SectionType.Curved;

    let pathAveragedSlopeForCurveSection =
CalculatePathAveragedSlope(trueCurveSection, this.Slopes, this.Distances, this.AccumulativeDistances);
    trueCurveSection.PathAvergaedSlope = pathAveragedSlopeForCurveSection;
    trueCurveSection.InitialHeading =
this.SmoothedHeading[trueCurveSection.StartIndex];
    this.AddSectionMetaData(trueCurveSection);

    //changing threshold of straight sections depending on slope of curve
    if (Math.abs(trueCurveSection.PathAvergaedSlope) > 0.02) {
        for(let j = 1; j < this.StraightSectionsTH2.length; j++) {
            let countend=0;
            if (this.StraightSectionsTH2[j].EndIndex >=
previousStraightSection.EndIndex) {

previousStraightSection.EndIndex=this.StraightSectionsTH2[j].EndIndex;

currentStraightSection.StartIndex=this.StraightSectionsTH2[j+1].StartIndex;
                countend++;
                if (countend == 1) {
                    break;
                }
            }
        }
    }

    //combining straight sections if curve length is less than 75m
    if (trueCurveSection.TotalSectionLength <=
0.8*MinimumPointsBetweenStraightSections
```

```
                    || Math.abs((currentStraightSection.OptimizedPathAveragedHeading-
previousStraightSection.OptimizedPathAveragedHeading)/(currentStraightSection.StartIndex-
previousStraightSection.EndIndex)) < threshold1) {

                        previousStraightSection.EndIndex = currentStraightSection.EndIndex;
                        this.StraightSections[i] = previousStraightSection;
                        this.AddSectionMetaData(this.StraightSections[i]);
                        continue;
                    }

                    OptimizeCurveSection(trueCurveSection, this.SmoothedHeading, this.Distances,
this.AccumulativeDistances); //optimize curve section
                    this.AddSectionMetaData(trueCurveSection);

                    let pathAveragedHeadingP =
CalculatePathAveragedHeading(previousStraightSection, this.SmoothedHeading, this.Distances,
this.AccumulativeDistances);
                    previousStraightSection.PathAveragedHeading = pathAveragedHeadingP;
                    OptimizeStraightSection(previousStraightSection, this.SmoothedHeading,
this.Distances); // optimize straight sections
                    this.AddSectionMetaData(previousStraightSection);
                    this.AllSections.pop();
                    this.AllSections.push(previousStraightSection);

                    let pathAveragedHeading =
CalculatePathAveragedHeading(currentStraightSection, this.SmoothedHeading, this.Distances,
this.AccumulativeDistances);
                    currentStraightSection.PathAveragedHeading = pathAveragedHeading;
                    OptimizeStraightSection(currentStraightSection, this.SmoothedHeading,
this.Distances); // optimize straight sections
                    this.AddSectionMetaData(currentStraightSection);


                    // now we have curve section, the sections to the right and left are transient
sections.
                    let leftTransientSection = new Section(previousStraightSection.EndIndex,
trueCurveSection.StartIndex, SectionType.Transient);
                    leftTransientSection.InitialHeading =
previousStraightSection.PathAveragedHeading;
                    var sectionLengthLeftTransient =
this.AccumulativeDistances[leftTransientSection.EndIndex] -
this.AccumulativeDistances[leftTransientSection.StartIndex];
                    let pathAveragedSlopeForLeftTransientSection =
(trueCurveSection.InitialHeading-leftTransientSection.InitialHeading)/sectionLengthLeftTransient;
                    leftTransientSection.PathAvergaedSlope =
pathAveragedSlopeForLeftTransientSection;
                    leftTransientSection.OptimizedInitialHeading =
previousStraightSection.OptimizedPathAveragedHeading;
```

```
                    let OptimizedpathAveragedSlopeForLeftTransientSection =
(trueCurveSection.OptimizedInitialHeading-
leftTransientSection.OptimizedInitialHeading)/sectionLengthLeftTransient;
                    leftTransientSection.OptimizedPathAvergaedSlope =
OptimizedpathAveragedSlopeForLeftTransientSection;

                    this.AddSectionMetaData(leftTransientSection);

                    let rightTransientSection = new Section(trueCurveSection.EndIndex,
currentStraightSection.StartIndex, SectionType.Transient);

                    var sectionLength = this.AccumulativeDistances[trueCurveSection.EndIndex] -
this.AccumulativeDistances[trueCurveSection.StartIndex];
                    rightTransientSection.InitialHeading = trueCurveSection.InitialHeading +
(trueCurveSection.PathAvergaedSlope * sectionLength);
                    var sectionLengthRightTransient =
this.AccumulativeDistances[rightTransientSection.EndIndex] -
this.AccumulativeDistances[rightTransientSection.StartIndex];
                    let pathAveragedSlopeForRightTransientSection =
(currentStraightSection.PathAveragedHeading-
rightTransientSection.InitialHeading)/sectionLengthRightTransient;
                    rightTransientSection.PathAvergaedSlope =
pathAveragedSlopeForRightTransientSection;
                    rightTransientSection.OptimizedInitialHeading =
trueCurveSection.OptimizedInitialHeading + (trueCurveSection.OptimizedPathAvergaedSlope *
sectionLength);
                    let OptimizedpathAveragedSlopeForRightTransientSection =
(currentStraightSection.OptimizedPathAveragedHeading-
rightTransientSection.OptimizedInitialHeading)/sectionLengthRightTransient;
                    rightTransientSection.OptimizedPathAvergaedSlope =
OptimizedpathAveragedSlopeForRightTransientSection;

                    this.AddSectionMetaData(rightTransientSection);

                    this.AllSections.push(leftTransientSection);
                    this.AllSections.push(trueCurveSection);
                    this.AllSections.push(rightTransientSection);
                    this.AllSections.push(currentStraightSection);
                }



                // create section meta data of bounding boxes.
                this.AllSections.forEach(section => {
                    try {
                        CalculateRectangleOfSection(section);
                    }
                    catch(e) {}
```

```
                });
        }

        private AddSectionMetaData(section : Section)
        {
                section.StartLatitude = this.Latitudes[section.StartIndex];
                section.StartLongitude = this.Longitudes[section.StartIndex];
                section.EndLatitude = this.Latitudes[section.EndIndex];
                section.EndLongitude = this.Longitudes[section.EndIndex];

                // Also calculate the total length (accumulative distance) of section
                section.TotalSectionLength = this.AccumulativeDistances[section.EndIndex] -
this.AccumulativeDistances[section.StartIndex];
                section.AccumulativeDistanceAtStart = this.AccumulativeDistances[section.StartIndex];

                // rectangles start 1 point after the actual section starts and end 1 point before the
actual section ends
                section.RectangleStartLatitude = this.Latitudes[section.StartIndex + 1];
                section.RectangleStartLongitude = this.Longitudes[section.StartIndex + 1];
                section.RectangleEndLatitude = this.Latitudes[section.EndIndex - 1];
                section.RectangleEndLongitude = this.Longitudes[section.EndIndex - 1];

                // if we have a curved or transient section then we need the mid-point of the section
too but for now we will calculate the midpoint for all sections
                var midpoint = section.StartIndex + Math.floor((section.EndIndex -
section.StartIndex)/2);
                section.MidLatitude = this.Latitudes[midpoint];
                section.MidLongitude = this.Longitudes[midpoint];
        }
}

export function AreSnapshotsOnSamePoint(a:Snapshot, b: Snapshot, useGooglePoints: boolean) :
boolean {
        return useGooglePoints ?
                                a.GoogleLatitude == b.GoogleLatitude && a.GoogleLongitude ==
b.GoogleLongitude :
                                a.Latitude == b.Latitude && a.Longitude == b.Longitude;
}

export function CalculatePathAveragedHeading(section: Section, headings: number[], distances:
number[], accumulativeDistances: number[]): number
{
        let sumOfHeadingMultiplyDistance: number[] = []
        sumOfHeadingMultiplyDistance.push(0);

        let j = 1;
        for (let i = section.StartIndex + 1; i <= section.EndIndex; i++)
        {
```

```
		sumOfHeadingMultiplyDistance[j] = sumOfHeadingMultiplyDistance[j - 1] + (headings[i]
* distances[i]);
		j++;
	}

	let sectionLength = accumulativeDistances[section.EndIndex] -
accumulativeDistances[section.StartIndex];
	let pathAveragedValue = sumOfHeadingMultiplyDistance[sumOfHeadingMultiplyDistance.length
- 1] / sectionLength;

	return pathAveragedValue;
}

// TODO: combine CalculatePathAveragedDifferentialHeading and CalculatePathAveragedHeading
export function CalculatePathAveragedDifferentialHeading(section: Section, differentialHeadings:
number[], distances: number[], accumulativeDistances: number[]): number
{
	let sumOfDifferentialHeadingMultiplyDistance: number[] = []
	sumOfDifferentialHeadingMultiplyDistance.push(0);

	let j = 1;
	for (let i = section.StartIndex + 1; i <= section.EndIndex; i++)
	{
		sumOfDifferentialHeadingMultiplyDistance[j] =
sumOfDifferentialHeadingMultiplyDistance[j - 1] + (differentialHeadings[i] * distances[i]);
		j++;
	}

	let distanceInSection = accumulativeDistances[section.EndIndex] -
accumulativeDistances[section.StartIndex];
	let pathAveragedDifferentialHeading =
sumOfDifferentialHeadingMultiplyDistance[sumOfDifferentialHeadingMultiplyDistance.length - 1] /
distanceInSection;

	return pathAveragedDifferentialHeading;
}

export function CalculatePathAveragedDifferentialHeadings(sections: Section[], differentialHeadings:
number[], distances: number[], accumulativeDistances: number[]): number[]
{
	let pathAveragedDifferentialHeadings: number[] = [];

	sections.forEach(section => {
		let pathAveragedDifferentialHeading =
CalculatePathAveragedDifferentialHeading(section, differentialHeadings, distances,
accumulativeDistances);
		pathAveragedDifferentialHeadings.push(pathAveragedDifferentialHeading);
	});
```

```
	return pathAveragedDifferentialHeadings;
}

export function CalculatePathAveragedHeadings(sections: Section[], headings: number[], distances:
number[], accumulativeDistances: number[]): number[]
{
	let pathAveragedHeadings: number[] = [];

	sections.forEach(section => {
		let pathAveragedHeading = CalculatePathAveragedHeading(section, headings, distances,
accumulativeDistances);
		pathAveragedHeadings.push(pathAveragedHeading);
	});

	return pathAveragedHeadings;
}

export function PathAveragedDifferentialHeadingReselect(section: Section, differentialHeadings:
number[], currentPathAveragedDifferentialHeading: number): Section {
	let sb: number[] = [];
	for (let i = section.StartIndex; i < section.EndIndex; i++) {
		if (Math.abs(differentialHeadings[i]) >=
Math.abs(currentPathAveragedDifferentialHeading)) {
			sb.push(i)
		}
	}

	return new Section(sb[0], sb[sb.length - 1], SectionType.Unknown);
}

export function CalculatePathAveragedSlope(section: Section, Slopes: number[], distances: number[],
accumulativeDistances: number[]): number
{
	let distanceInSection = accumulativeDistances[section.EndIndex] -
accumulativeDistances[section.StartIndex];
	let sumOfSlopeMultiplyDistance: number[] = []
	sumOfSlopeMultiplyDistance.push(0);

	let j = 1;
	for (let i = section.StartIndex + 1; i <= section.EndIndex; i++)
	{
		sumOfSlopeMultiplyDistance[j] = sumOfSlopeMultiplyDistance[j - 1] + (Slopes[i] *
distances[i]);
		j++;
	}
```

```
	let pathAveragedDifferentialHeading =
sumOfSlopeMultiplyDistance[sumOfSlopeMultiplyDistance.length - 1] / distanceInSection;

	return pathAveragedDifferentialHeading;
}

export function CalculatePathAveragedSlopeOfTransitionSection(section: Section, SmoothedHeadings:
number[], distances: number[], accumulativeDistances: number[]): number
{
	let distanceInSection = accumulativeDistances[section.EndIndex] -
accumulativeDistances[section.StartIndex];
	let pathAveragedSlopeOfTransientSection = (SmoothedHeadings[section.EndIndex] -
SmoothedHeadings[section.StartIndex])/ distanceInSection;
	return pathAveragedSlopeOfTransientSection;
}

export function OptimizeStraightSection(straightSection: Section, headings: number[], distances:
number[]) {
	// we will try to find the optimum PAH value between 0.9PAH to 1.1PAH. PAH which produces
the min ALS
	// is the optimum PAH value.
	let headingsInSection = headings.slice(straightSection.StartIndex, straightSection.EndIndex + 1);
	const pathAveragedHeadingMaxValue = Math.max.apply(null, headingsInSection);
	const pathAveragedHeadingMinValue = Math.min.apply(null, headingsInSection);
	straightSection.MaxHeadingInSection = pathAveragedHeadingMaxValue;
	straightSection.MinHeadingInSection = pathAveragedHeadingMinValue;
	const delataPathAveragedHeading = (pathAveragedHeadingMaxValue -
pathAveragedHeadingMinValue) / 100;

	let currentOptimimPathAveragedHeadingValue = straightSection.PathAveragedHeading;
	let currentMinimumAlsValue = Number.MAX_VALUE;

	let potentialPathAveragedHeading = pathAveragedHeadingMinValue;
	while (potentialPathAveragedHeading <= pathAveragedHeadingMaxValue) {
		let als = 0;
		for (let index = straightSection.StartIndex + 1; index <= straightSection.EndIndex;
index++) {
			let headingAtPoint = headings[index];

			// WARN: there is a sign difference between the doc and code implementation
			let theta = headingAtPoint - potentialPathAveragedHeading; // whats the unit
here? degrees?
			let thetaInRadians = theta * Math.PI / 180;
			als = als + distances[index] * Math.sin(thetaInRadians);
		}

		if (Math.abs(als) < Math.abs(currentMinimumAlsValue)) {
			currentMinimumAlsValue = als;
```

```
                    currentOptimimPathAveragedHeadingValue = potentialPathAveragedHeading;
                }

                potentialPathAveragedHeading = potentialPathAveragedHeading +
delataPathAveragedHeading;
        }

        // straightSection.PathAveragedHeading = currentOptimimPathAveragedHeadingValue;
        straightSection.OptimizedPathAveragedHeading = currentOptimimPathAveragedHeadingValue;
}

export function OptimizeCurveSection(curveSection: Section, headings: number[], distances: number[],
accumulativeDistances: number[]) {
        // we will try to find the optimum PAS value between 0.9PAS to 1.1PAS. PAS which produces the
min ALS
        // is the optimum PAH value.

        const pathAveragedSlopeMaxValue = curveSection.PathAvergaedSlope + Math.abs(0.3 *
curveSection.PathAvergaedSlope);
        const pathAveragedSlopeMinValue = curveSection.PathAvergaedSlope - Math.abs(0.3 *
curveSection.PathAvergaedSlope);
        const deltaPathAveragedSlope = Math.abs((pathAveragedSlopeMaxValue -
pathAveragedSlopeMinValue) / 100);

        const initialHeadingsRange = headings.slice(curveSection.StartIndex - 5, curveSection.StartIndex
+ 5); // 2 points on each side of original IH
        const initialHeadingMaxValue = Math.max.apply(null, initialHeadingsRange);
        const initialHeadingMinValue = Math.min.apply(null, initialHeadingsRange);
        const deltaInitialHeading = (initialHeadingMaxValue - initialHeadingMinValue) / 100;

        let currentOptimimInitialHeadingValue = curveSection.InitialHeading;
        let currentOptimimPathAveragedSlopeValue = curveSection.PathAvergaedSlope;

        let currentMinimumAlsValue = Number.MAX_VALUE;
        let allAlsValues: number[] = [];
        let potentialPathAveragedSlope = pathAveragedSlopeMinValue;
        while (potentialPathAveragedSlope <= pathAveragedSlopeMaxValue) {

                let potentialInitialHeadingValue = initialHeadingMinValue;
                while (potentialInitialHeadingValue <= initialHeadingMaxValue) {
                        let als = 0;
                        for (let index = curveSection.StartIndex + 1; index <= curveSection.EndIndex;
index++) {
                                let headingAtPoint = headings[index];

                                // WARN: there is a sign difference between the doc and code
implementation
```

```
                                let distanceTillCurrentPoints = accumulativeDistances[index] -
accumulativeDistances[curveSection.StartIndex];
                                let href_k = potentialInitialHeadingValue +
distanceTillCurrentPoints*potentialPathAveragedSlope;
                                // let href_k = potentialInitialHeadingValue +
distances[index]*potentialPathAveragedSlope;
                                let theta = headingAtPoint - href_k;
                                let thetaInRadians = theta * Math.PI / 180;
                                //als = als + distances[index] * Math.sin(thetaInRadians);
                                if (als==0)
        {als = als + distances[index] * Math.cos(thetaInRadians);
        } else
        {als = als + distances[index] * Math.sin(thetaInRadians);}

                        }

                        if (Math.abs(als) < Math.abs(currentMinimumAlsValue)) {
                                currentMinimumAlsValue = als;
                                currentOptimimPathAveragedSlopeValue =
potentialPathAveragedSlope;
                                currentOptimimInitialHeadingValue = potentialInitialHeadingValue;
                        }

                        allAlsValues.push(als);
                        potentialInitialHeadingValue = potentialInitialHeadingValue +
deltaInitialHeading;
                }

                potentialPathAveragedSlope = potentialPathAveragedSlope + deltaPathAveragedSlope;
        }


        curveSection.OptimizedInitialHeading = currentOptimimInitialHeadingValue;
        curveSection.OptimizedPathAvergaedSlope = currentOptimimPathAveragedSlopeValue;
}

export function OptimizeTransientSection(curveSection: Section) {
        // In transient sections there is no optimization done so the optimized parameters are equal to
non-optimized parameter/

        curveSection.OptimizedInitialHeading = curveSection.InitialHeading;
        curveSection.OptimizedPathAvergaedSlope = curveSection.PathAvergaedSlope;
}

export function GetStraightSections(averagedHeadings: number[],threshold: number): Section[] {
        let scanWindow = 3;
        // let threshold1 = 0.002;
        // let threshold2 = 0.02;
```

```
	let straightSectionHelperArray: number[] = [];
	for (let i = 1; i < averagedHeadings.length - 1 - scanWindow; i++) {
		// if all points in our scanWindow are above threshold then
		let currentWindow = averagedHeadings.slice(i, i + scanWindow);

		if (currentWindow.every(val => val > threshold)) {
			straightSectionHelperArray.push(100);
		}
		else if (currentWindow.every(val => val < -threshold)) {
			straightSectionHelperArray.push(-100);
		} else {
			straightSectionHelperArray.push(0);
		}
	}

	let sections: Section[] = [];

	// now all consecutive '0' points in our helper array are straight sections.
	for (let i = 0; i < straightSectionHelperArray.length; i++) {
		if (straightSectionHelperArray[i] === 0) {
			let startStraightSectionIndex = i;
			let endStraightSectionIndex = -1;
			for (let j = i; j < straightSectionHelperArray.length; j++) {
				if (straightSectionHelperArray[j] !== 0 || j ===
straightSectionHelperArray.length - 1) {
					endStraightSectionIndex = j;

					sections.push(new Section(startStraightSectionIndex,
endStraightSectionIndex, SectionType.Straight));
					i = j + 1;
					break;
				}
			}
		}
	}

	return sections;
}

export function GetAllNonStraightSections(straightSections: Section[]): Section[]
{
	let nonStraightSections: Section[] = [];
	// Assume our path starts and ends at a straight section for now.
	for (let i = 1; i < straightSections.length; i++) {
		const currentSection = straightSections[i];
		const previousSection = straightSections[i - 1];
```

```
                let nonStraightSection = new Section(previousSection.EndIndex,
currentSection.StartIndex - 1, SectionType.Unknown);
                nonStraightSections.push(nonStraightSection);
        }

        return nonStraightSections;
}

export function CalculateAveragedDifferentialHeadings(differentialHeadings: number[]): number[] {
        let numberOfHeadingsToAverage = 40; // 20 points ahead and 20 points behind
        let averagedHeadings: number[] = Array(differentialHeadings.length).fill(0);

        // We can't average start and of array on both sides so just copy over original values
        for (let i = 0; i < numberOfHeadingsToAverage; i++) {
                averagedHeadings[i] = differentialHeadings[i];

                let lastIndex = differentialHeadings.length - 1;
                averagedHeadings[lastIndex - i] = differentialHeadings[lastIndex - i];
        }

        for (let i = numberOfHeadingsToAverage; i < differentialHeadings.length - 1 -
numberOfHeadingsToAverage; i++) {
                // slice the array into 20 points around current point.
                let startIndex = i - numberOfHeadingsToAverage;
                let endIndex = i + numberOfHeadingsToAverage + 1; // endIndex is excluded in slice.
                var slice =  differentialHeadings.slice(startIndex, endIndex);
                var sumHeading = slice.reduce((a, b) => a + b);
                let averagedHeading = sumHeading / (endIndex - startIndex)
                averagedHeadings[i] = averagedHeading;
        }

        return averagedHeadings;
}


function GetNextPowerOfTwoNumber(input: number): number {
        let power = Math.ceil(Math.log2(input));
        return Math.pow(2, power);
}

export function ApplySmoothingfilter(input: number[], cutOffFrequency1: number, cutOffFrequency2:
number): number[] {
        console.log(`input: ${input}`);

        var clonedInput  = [...input];

        const fs = GetNextPowerOfTwoNumber(clonedInput.length);
        for (let i = clonedInput.length + 1; i <= fs; i++) {
```

```
            clonedInput.push(0);
        }

        var fft = new dsp.FFT(fs, fs);
        fft.forward(clonedInput);
        var fftReal = fft.real;
        var fftImag = fft.imag;

        for (let i = cutOffFrequency2; i <= fs/2; i++) {
            fftReal[i] = 0;
            fftReal[fs - i + 1] = 0;

            fftImag[i] = 0;
            fftImag[fs - i + 1] = 0;
        }

        for (let i = cutOffFrequency1; i <= cutOffFrequency2 - 1; i++) {
            var f_f = (cutOffFrequency2-1-i)/(cutOffFrequency2-cutOffFrequency1+1);
            fftReal[i] = fftReal[i]*f_f;
            fftImag[i] = fftImag[i]*f_f;

            fftReal[fs-i+1] = fftReal[fs-i+1]*f_f;
            fftImag[fs-i+1] = fftImag[fs-i+1]*f_f;
        }

        var smoothedData: number[] = fft.inverse(fftReal, fftImag);
        for (let index = 0; index < smoothedData.length; index++) {
            smoothedData[index] = Math.abs(smoothedData[index]);
        }

        return smoothedData.slice(0, input.length);
}

export function CalculateRectangleOfSection(section: Section)
{
        var width = 10;
        let rectangle: SectionRectangle;

        if (section.SectionType === SectionType.Straight)
        {
            let headingFromStartToEndOfRectangle = normalizeHeading(headingTo( //
normalizeHeading makes heading to be in 0 : 360 range
                {lat: section.RectangleStartLatitude, lon: section.RectangleStartLongitude },
                {lat: section.RectangleEndLatitude, lon: section.RectangleEndLongitude }
            ));

            // we need angle wrt east so we need to add 90 degrees to the above answer
            var headingWrtEast = headingFromStartToEndOfRectangle + 90;
```

```
                var distanceRatio = (width / 2) / EARTH_RADIUS
                rectangle = {
                        StartMaxLatitude: section.RectangleStartLatitude + (distanceRatio *
Math.cos(headingWrtEast)),
                        StartMaxLongitude: section.RectangleStartLongitude + (distanceRatio *
Math.sin(headingWrtEast)),

                        StartMinLatitude: section.RectangleStartLatitude - (distanceRatio *
Math.cos(headingWrtEast)),
                        StartMinLongitude: section.RectangleStartLongitude - (distanceRatio *
Math.sin(headingWrtEast)),

                        EndMaxLatitude: section.RectangleEndLatitude + (distanceRatio *
Math.cos(headingWrtEast)),
                        EndMaxLongitude: section.RectangleEndLongitude + (distanceRatio *
Math.sin(headingWrtEast)),

                        EndMinLatitude: section.RectangleEndLatitude - (distanceRatio *
Math.cos(headingWrtEast)),
                        EndMinLongitude: section.RectangleEndLongitude - (distanceRatio *
Math.sin(headingWrtEast))
                }
        } else
        {
                // curve or transient section
                let headingDistanceFromStartToMidOfRectangle = headingDistanceTo(
                        {lat: section.RectangleStartLatitude, lon: section.RectangleStartLongitude },
                        {lat: section.MidLatitude, lon: section.MidLongitude }
                );

                // we need angle wrt east so we need to add 90 degrees to the above answer
                var headingWrtEast =
normalizeHeading(headingDistanceFromStartToMidOfRectangle.heading) + 90; // normalizeHeading
makes heading to be in 0 : 360 range
                var minDistanceToMidPoint = headingDistanceFromStartToMidOfRectangle.distance; //
min distance from P1 to Pm
                section.PerpendicularDistanceToMidPoint = minDistanceToMidPoint *
Math.sin(headingWrtEast); // d

                // now our width of ractangle will be d + w
                var distanceRatio = (Math.abs(section.PerpendicularDistanceToMidPoint) + width) /
EARTH_RADIUS;
                var distanceRatio2 = (width - section.PerpendicularDistanceToMidPoint) /
EARTH_RADIUS;
                var coefficient = section.PathAvergaedSlope >= 0 ? 1 : -1;

                rectangle = {
```

```
                        StartMaxLatitude: section.RectangleStartLatitude + (coefficient * (distanceRatio
* Math.cos(headingWrtEast))),
                        StartMaxLongitude: section.RectangleStartLongitude + (coefficient *
(distanceRatio * Math.sin(headingWrtEast))),

                        StartMinLatitude: section.RectangleStartLatitude,
                        StartMinLongitude: section.RectangleStartLongitude,

                        EndMaxLatitude: section.RectangleEndLatitude + (coefficient * (distanceRatio *
Math.cos(headingWrtEast))),
                        EndMaxLongitude: section.RectangleEndLongitude + (coefficient *
(distanceRatio * Math.sin(headingWrtEast))),

                        EndMinLatitude: section.RectangleEndLatitude,
                        EndMinLongitude: section.RectangleEndLongitude
                    }
                    //overwriting the minimum lat and long according to the condition
    if (section.PerpendicularDistanceToMidPoint < width)
    {
       rectangle.StartMinLatitude = section.RectangleStartLatitude - (coefficient * (distanceRatio2 *
Math.cos(headingWrtEast)));
       rectangle.StartMinLongitude = section.RectangleStartLongitude - (coefficient * (distanceRatio2 *
Math.sin(headingWrtEast)));

       rectangle.EndMinLatitude = section.RectangleEndLatitude - (coefficient * (distanceRatio2 *
Math.cos(headingWrtEast)));
       rectangle.EndMinLongitude = section.RectangleEndLongitude - (coefficient * (distanceRatio2 *
Math.sin(headingWrtEast)))

    } else
    {
       rectangle.StartMinLatitude = section.RectangleStartLatitude;
       rectangle.StartMinLongitude = section.RectangleStartLongitude;

       rectangle.EndMinLatitude = section.RectangleEndLatitude;
       rectangle.EndMinLongitude = section.RectangleEndLongitude

    }

        }

        section.SectionRectangle = rectangle;
}
```

Appendix B:
Code used for Implementation of LDD Algorithm for Web Browser Tool

Please note that the entire code for the web browser tool is a few thousand lines because it has to accommodate data management and user interface in addition to the two algorithms, RRH generation algorithm and LDD algorithm. This code is written in Typescript (Angular) language. Below is the specific code used for LDD algorithm in web browser tool.

```
import { BoundingBox, distanceTo, headingDistanceTo, insideBoundingBox } from "geolocation-utils";
import { PrintLdwSnapshotsAsCsv } from "./FileSaver";
import { LaneDepartureSnapshot } from "./LaneDepartureSnapshot";
import { MapService } from "./map.service";
import { Section, SectionType } from "./Section";
import { Snapshot } from "./Snapshot";

export class LaneDepartureRoutine {

	// TODO: The DataStructure probably should be a Queue since we only use latest `X` number of elements and we need a mechanism to
	// remove the older items. For now I am using arrays since JS/TS doesn't have a built-in queue data structure.
	DataSnapshots: LaneDepartureSnapshot[] = [];
	Counter: number = 0;

	constructor(allSections: Section[], gpsSnapshots: Snapshot[], mapService: MapService) {

		mapService.initMap();
		mapService.setMap(gpsSnapshots[0].Latitude, gpsSnapshots[1].Longitude);
		gpsSnapshots.forEach(gpsSnapshot => {
			this.ProcessNewGpsSnapshot(gpsSnapshot, allSections);
			this.PredictLaneDeparture();
		});

		// The below code is used only in Development/Matlab scripts and should not be ported over to mobile.
		this.getFirstPointInFirstAndSecondValidSectionsAndUpdateSnapshots(allSections);
		PrintLdwSnapshotsAsCsv(this.DataSnapshots);
	}

	// This will basically be the GPS event callback
	ProcessNewGpsSnapshot(gpsSnapshot: Snapshot, allSections: Section[]) {
		let startDate: Date = this.DataSnapshots.length > 0 ? this.DataSnapshots[0].TimeStamp : new Date(gpsSnapshot.TimeStampAsString);
		let currentDatasnapshot: LaneDepartureSnapshot = new LaneDepartureSnapshot(
			gpsSnapshot.Latitude,
			gpsSnapshot.Longitude,
			gpsSnapshot.SnapshotNumber,
			gpsSnapshot.TimeStampAsString,
			startDate);
```

```
		if (this.DataSnapshots.length < 2) {
			this.DataSnapshots.push(currentDatasnapshot);
			return;
		}

		let previousDataSnapshot = this.DataSnapshots[this.DataSnapshots.length - 1];
		let headingDistanceFromPreviousSnapshot = headingDistanceTo(
			{lat: previousDataSnapshot.Latitude, lon: previousDataSnapshot.Longitude },
			{lat: gpsSnapshot.Latitude, lon: gpsSnapshot.Longitude }
		)

		currentDatasnapshot.Distance = headingDistanceFromPreviousSnapshot.distance;
		currentDatasnapshot.AccumulativeDistance =
previousDataSnapshot.AccumulativeDistance + currentDatasnapshot.Distance; // not really used?
		currentDatasnapshot.Heading = headingDistanceFromPreviousSnapshot.heading + 360;

		if (this.DataSnapshots.length < 5) {
			this.DataSnapshots.push(currentDatasnapshot);
			return;
		}

		let previousToPreviousDataSnapshot = this.DataSnapshots[this.DataSnapshots.length -
2];
		currentDatasnapshot.AveragedHeading = (previousToPreviousDataSnapshot.Heading +
previousDataSnapshot.Heading + currentDatasnapshot.Heading) / 3;

		const [currentVahicleSection, sectionInfo] =
this.GetSectionOfVehicle(currentDatasnapshot.Latitude, currentDatasnapshot.Longitude, allSections);
		currentDatasnapshot.SectionInfo = sectionInfo;
		if (currentVahicleSection === undefined) {
			// what to do when we can't find point in any section?
			this.Counter++;
			return;
		}

		currentDatasnapshot.SectionStartIndex = currentVahicleSection.StartIndex;
		currentDatasnapshot.PerpendicularDistanceToMidPoint =
currentVahicleSection.PerpendicularDistanceToMidPoint;
		currentDatasnapshot.PathAvergaedSlope = currentVahicleSection.PathAvergaedSlope;
		currentDatasnapshot.OptimizedPathAvergaedSlope =
currentVahicleSection.OptimizedPathAvergaedSlope;
		currentDatasnapshot.InitialHeading = currentVahicleSection.InitialHeading;
		currentDatasnapshot.OptimizedInitialHeading =
currentVahicleSection.OptimizedInitialHeading;
		// We need to figure out if this is the first point in a 'new section`. (this should check
SectionId in final version)
		if (currentDatasnapshot.SectionStartIndex === previousDataSnapshot.SectionStartIndex)
{
```

```
                currentDatasnapshot.DistanceFromStartOfSection =
previousDataSnapshot.DistanceFromStartOfSection + currentDatasnapshot.Distance;
        } else {
                currentDatasnapshot.IsFirstPointInSection = true;
                currentDatasnapshot.DistanceFromStartOfSection = distanceTo(
                        {lat: currentVahicleSection.StartLatitude, lon:
currentVahicleSection.StartLongitude },
                        {lat: currentDatasnapshot.Latitude, lon: currentDatasnapshot.Longitude
}
                );
        }

        // Confirm DistanceFromStartOfSection
        currentDatasnapshot.OptimizedRefrenceHeading = currentVahicleSection.SectionType
=== SectionType.Straight
                ? currentVahicleSection.OptimizedPathAveragedHeading
                : currentVahicleSection.OptimizedInitialHeading +
(currentVahicleSection.OptimizedPathAvergaedSlope *
currentDatasnapshot.DistanceFromStartOfSection);

        currentDatasnapshot.RefrenceHeading = currentVahicleSection.SectionType ===
SectionType.Straight
                ? currentVehicleSection.PathAveragedHeading
                : currentVahicleSection.InitialHeading +
(currentVahicleSection.PathAvergaedSlope * currentDatasnapshot.DistanceFromStartOfSection);

        // Lateral Distance and Accumulated Lateral Distance Calculation (we use these to
decide if lane departure happened)
        currentDatasnapshot.Theta = currentDatasnapshot.OptimizedRefrenceHeading -
currentDatasnapshot.AveragedHeading; // rename to currentAveragedHeading

        // We want to find Perpendicular field of a right triangle
        currentDatasnapshot.LateralDistance = currentDatasnapshot.Distance *
Math.sin(currentDatasnapshot.Theta * Math.PI / 180) // Sin(Theta) = P / H
        currentDatasnapshot.AccumulativeLateralDistance =
previousDataSnapshot.AccumulativeLateralDistance + currentDatasnapshot.LateralDistance;

        // Although we probably will only care about absolute value of lateral distances but to
help debug stuff I will create separate properties
        // for Absolute lateral distances
        currentDatasnapshot.AbsoluteLateralDistance =
Math.abs(currentDatasnapshot.LateralDistance)
        currentDatasnapshot.AbsoluteAccumulativeLateralDistance =
Math.abs(currentDatasnapshot.AccumulativeLateralDistance)
        currentDatasnapshot.SectionType = currentVahicleSection.SectionType
        currentDatasnapshot.PathAveragedHeading=
currentVahicleSection.PathAveragedHeading
```

```
                currentDatasnapshot.OptimizedPathAveragedHeading =
currentVahicleSection.OptimizedPathAveragedHeading
                this.DataSnapshots.push(currentDatasnapshot);


                let startofFirstSectionLat = allSections[0].StartLatitude;
                let startofFirstSectionLon = allSections[0].StartLongitude;
    currentDatasnapshot.dist2StartofAllsections = distanceTo({lat: startofFirstSectionLat, lon:
startofFirstSectionLon },{lat: currentDatasnapshot.Latitude, lon: currentDatasnapshot.Longitude });

                let endofLasttSectionLat = allSections[allSections.length-1].EndLatitude;
                let endofLastSectionLon = allSections[allSections.length-1].EndLongitude;
    currentDatasnapshot.dist2EndofAllsections = distanceTo({lat: endofLasttSectionLat, lon:
endofLastSectionLon },{lat: currentDatasnapshot.Latitude, lon: currentDatasnapshot.Longitude });
  }

        private getFirstPointInFirstAndSecondValidSectionsAndUpdateSnapshots(allSections: Section[]) {
                // The code in this function is used only in Development/Matlab scripts and should not
be ported over to mobile.
                let firstSnapshotIndex = -1;
                let firstSectionIndex = -1;
                let secondSnapshotIndex = -1;
                let secondSectionIndex = -1;

                for (let J = 0; J < this.DataSnapshots.length - 1; J++) {
                        const [VehicletestSection, _] =
this.GetSectionOfVehicle(this.DataSnapshots[J].Latitude, this.DataSnapshots[J].Longitude, allSections);

                        if (VehicletestSection === undefined) {
                                continue;
                        }

                        if (firstSectionIndex === -1) {
                                firstSectionIndex = VehicletestSection.StartIndex;
                                firstSnapshotIndex = J;
                        } else if (VehicletestSection.StartIndex > firstSectionIndex) {
                                secondSectionIndex = VehicletestSection.StartIndex;
                                secondSnapshotIndex = J;
                                break;
                        }
                }

                for (let index = 0; index < this.DataSnapshots.length - 1; index++) {
                        const currentDatasnapshot = this.DataSnapshots[index];
                        currentDatasnapshot.firstSnapshotIndex = firstSnapshotIndex;
                        currentDatasnapshot.firstSectionIndex = firstSectionIndex;
                        currentDatasnapshot.secondSnapshotIndex = secondSnapshotIndex;
                        currentDatasnapshot.secondSectionIndex = secondSectionIndex;
```

```
                        if (secondSectionIndex !== -1){
                                currentDatasnapshot.sum =
this.DataSnapshots[secondSnapshotIndex].AccumulativeDistance-
this.DataSnapshots[firstSnapshotIndex].AccumulativeDistance;
                                currentDatasnapshot.section =
allSections[firstSnapshotIndex].TotalSectionLength;
                        }

                        currentDatasnapshot.dist = distanceTo({lat: allSections[0].StartLatitude, lon:
allSections[0].StartLongitude },{lat: this.DataSnapshots[0].Latitude, lon: this.DataSnapshots[0].Longitude
});
                }
        }

        PredictLaneDeparture() {
                if (this.DataSnapshots.length < 5) {
                        return;
                }

                let lastDataSnaphotIndex = this.DataSnapshots.length - 1;

                this.DataSnapshots[lastDataSnaphotIndex].Alarm =
this.DataSnapshots[lastDataSnaphotIndex - 1].Alarm;
                if (Math.abs(this.DataSnapshots[lastDataSnaphotIndex].AccumulativeLateralDistance)
== 0 ||
                   Math.abs(this.DataSnapshots[lastDataSnaphotIndex -
1].AccumulativeLateralDistance) == 0 ||
                        Math.abs(this.DataSnapshots[lastDataSnaphotIndex -
2].AccumulativeLateralDistance) == 0 ) {
                                return;
                }

                // Alarm here. For debugging/testing I will update a field in the object.
                this.DataSnapshots[lastDataSnaphotIndex].Alarm = true;
                // If previous state was in alarm then this isn't start of alarm.
                this.DataSnapshots[lastDataSnaphotIndex].StartOfAlarm =
this.DataSnapshots[lastDataSnaphotIndex - 1].Alarm == true ? false: true;

                // We now need to calculate if Accmulative Lateral Distance needs to be reset or not.
Once a vehicle completes
                // a lane departure we need to set the Accumulative lateral distance to 0 so we can
catch the next lane departure.
                let countOfSnapshotsGreaterThanZero = 0;
                let countOfSnapshotsLessThanZero = 0;

                for (let index = 0; index < 6; index++) {
                        if (this.DataSnapshots[lastDataSnaphotIndex - index].LateralDistance > 0) {
```

```
                countOfSnapshotsGreaterThanZero++;
            } else {
                countOfSnapshotsLessThanZero++;
            }
        }

        if (countOfSnapshotsGreaterThanZero >= 2 && countOfSnapshotsLessThanZero >= 2) {
            this.DataSnapshots[lastDataSnaphotIndex].AccumulativeLateralDistance = 0;
        }
    }

    GetSectionOfVehicle(latitude: number, longitude: number, allSections: Section[]): [Section |
undefined, string]
    {
        for (let index = 0; index < allSections.length; index++) {
            // TODO: currentSection should start from the previous point's section instead
of starting from beginning of all sections.
            const currentSection = allSections[index];
            var pointIsInCurrentSection = this.IsLocationInSection(latitude, longitude,
currentSection);

            if (pointIsInCurrentSection)
            {
                // At this point we also want to find out if this point is `owned` by
multiple sections. So, we will also check the next section.
                const isLastSection = index === allSections.length - 1;
                if (isLastSection)
                {
                    // We are in the last section so no point in checking the next
section since there is none.
                    return [currentSection, 'Single'];
                }

                const nextSection = allSections[index + 1];
                var pointIsInNextSection = this.IsLocationInSection(latitude, longitude,
nextSection);

                if (pointIsInNextSection)
                {
                    // We have a point that exists in 2 sections so we will have to
figure out which section is closer to the point.
                    let distanceFromEndOfCurrentSection = Math.abs(distanceTo(
                        {lat: currentSection.RectangleEndLatitude, lon:
currentSection.RectangleEndLongitude },
                        {lat: latitude, lon: longitude }
                    ));

                    let distanceFromStartOfNextSection = Math.abs(distanceTo(
```

```
                                                {lat: nextSection.RectangleStartLatitude, lon:
nextSection.RectangleStartLongitude },
                                                {lat: latitude, lon: longitude }
                                        ));

                                        return distanceFromEndOfCurrentSection <=
distanceFromStartOfNextSection ? [currentSection, 'Double'] : [nextSection, 'Double']
                                }

                                return [currentSection, 'Single-DistanceBased'];
                        }
                }

                // We haven't been able to find point in any section, we will now check distances to all
sections and asking this point to the section with min distance.
                let indexOfSectionWithMinDistance: number = 0;
                let currentMinDistance = Number.MAX_VALUE;
                for (let index = 0; index < allSections.length; index++)
                {
                        const currentSection = allSections[index];
                        let distanceFromStartOfCurrentSection = Math.abs(distanceTo(
                                {lat: currentSection.RectangleStartLatitude, lon:
currentSection.RectangleStartLongitude },
                                {lat: latitude, lon: longitude }
                        ));

                        if (distanceFromStartOfCurrentSection < currentMinDistance)
                        {
                                currentMinDistance = distanceFromStartOfCurrentSection;
                                indexOfSectionWithMinDistance = index;
                        }
                }

                // we now have the section with Min distance, now we should check if point should be
assigned to section before the selected section or not.
                // We have a point that exists in 2 sections so we will have to figure out which section is
closer to the point.
                var selectedSection = allSections[indexOfSectionWithMinDistance]
                if (indexOfSectionWithMinDistance === 0)
                {
                        // first section, we will start assigning sections when points atleast reach first
section.
                        return [undefined, 'PointBeforeFirstSection'];
                }

                var previousSection = allSections[indexOfSectionWithMinDistance - 1];

                let distanceFromEndOfPreviousSection = Math.abs(distanceTo(
```

```
                        {lat: previousSection.RectangleEndLatitude, lon:
previousSection.RectangleEndLongitude },
                        {lat: latitude, lon: longitude }
                ));

                return distanceFromEndOfPreviousSection <= currentMinDistance ? [previousSection,
'Double-DistanceBased'] : [selectedSection, 'Double-DistanceBased']
        }

        IsLocationInSection(latitude: number, longitude: number, currentSection: Section): boolean
        {
                var sectionRectangle = currentSection.SectionRectangle;
                if (sectionRectangle != undefined) {
                        var boundingBox: BoundingBox = {
                                topLeft:    {latitude: sectionRectangle.StartMinLatitude, longitude:
sectionRectangle.StartMinLongitude},
                                bottomRight: {latitude: sectionRectangle.EndMaxLatitude, longitude:
sectionRectangle.EndMaxLongitude}
                        }

                        return insideBoundingBox({latitude: latitude, longitude: longitude},
boundingBox)
                }

                throw new Error("no bounding rectangle defined for section.");
        }
}
```

Appendix C:
Code used for Implementation of LDD Algorithm in Smartphone App

Please note that the entire code for the smartphone app several thousand lines because it has to accommodate data management and user interface in addition to implementation of LDD algorithm. This code is written in Dart language so that it can be compiled for both Android and iOS devices. Below is the specific code used to implement LDD algorithm in the smartphone app.

```
ProcessNewGpsSnapshot(Snapshot gpsSnapshot, Section currentVahicleSection) {
  DateTime startDate = this.DataSnapshots.length > 0
     ? this.DataSnapshots[0].TimeStamp!
     : DateTime.parse(gpsSnapshot.TimeStampAsString!);

  // create a new data object which will be used from now on for LaneDeparture Routine.
  LaneDepartureSnapshot currentDatasnapshot = new LaneDepartureSnapshot(
     gpsSnapshot.Latitude!,
     gpsSnapshot.Longitude!,
     gpsSnapshot.SnapshotNumber,
     gpsSnapshot.TimeStampAsString!,
     startDate);

  // We need atleast 2 snapshots to calculate heading.
  if (this.DataSnapshots.length < 2) {
   this.DataSnapshots.add(currentDatasnapshot);
   return;
  }

  // calculate heading, distance and accumulative distance wrt previous point.
  var previousDataSnapshot =
     this.DataSnapshots[this.DataSnapshots.length - 1];

  var headingFromPreviousSnapshot = Geolocator.bearingBetween(
     previousDataSnapshot.Latitude!,
     previousDataSnapshot.Longitude!,
     gpsSnapshot.Latitude!,
     gpsSnapshot.Longitude!);

  var DistanceFromPreviousSnapshot = Geolocator.distanceBetween(
     previousDataSnapshot.Latitude!,
     previousDataSnapshot.Longitude!,
     gpsSnapshot.Latitude!,
     gpsSnapshot.Longitude!);

  currentDatasnapshot.Distance = DistanceFromPreviousSnapshot;
  currentDatasnapshot.AccumulativeDistance =
     previousDataSnapshot.AccumulativeDistance +
        currentDatasnapshot.Distance; // not really used?
  currentDatasnapshot.Heading = headingFromPreviousSnapshot + 360;

  // we need atleast 5 snapshots for rest of the logic to work.
```

```
  if (this.DataSnapshots.length < 5) {
    this.DataSnapshots.add(currentDatasnapshot);
    return;
  }

  // Average heading is calculated between 3 snapshots (current and previous 2 snapshots)
  var previousToPreviousDataSnapshot =
      this.DataSnapshots[this.DataSnapshots.length - 2];
  currentDatasnapshot.AveragedHeading =
      (previousToPreviousDataSnapshot.Heading +
          previousDataSnapshot.Heading +
          currentDatasnapshot.Heading) /
        3;

  // what to do when we can't find point in any section?
  if (currentVahicleSection == null) {
    this.Counter++;
    return;
  }

  // update snapshot data object with section attributes (some fields were added just for debugging
purposes)
  currentDatasnapshot.SectionStartIndex = currentVehicleSection.startIndex!;
  currentDatasnapshot.PerpendicularDistanceToMidPoint =
      currentVehicleSection.perpendicularDistanceToMidPoint;
  currentDatasnapshot.PathAvergaedSlope =
      currentVehicleSection.pathAvergaedSlope;
  currentDatasnapshot.OptimizedPathAvergaedSlope =
      currentVehicleSection.optimizedPathAvergaedSlope;
  currentDatasnapshot.InitialHeading = currentVehicleSection.initialHeading;
  currentDatasnapshot.OptimizedInitialHeading =
      currentVehicleSection.optimizedInitialHeading;

  // We need to figure out if this is the first point in a 'new section`.
  if (currentDatasnapshot.SectionStartIndex ==
      previousDataSnapshot.SectionStartIndex) {
    currentDatasnapshot.DistanceFromStartOfSection =
        previousDataSnapshot.DistanceFromStartOfSection +
            currentDatasnapshot.Distance;
  } else {
    currentDatasnapshot.IsFirstPointInSection = true;
    currentDatasnapshot.DistanceFromStartOfSection =
        Geolocator.distanceBetween(
            currentVehicleSection.startLatitude!,
            currentVahicleSection.startLongitude!,
            currentDatasnapshot.Latitude!,
            currentDatasnapshot.Longitude!);
  }
```

```
  // calculate reference heading
  currentDatasnapshot.OptimizedRefrenceHeading =
    currentVehicleSection.sectionType == SectionType.Straight.name
      ? currentVehicleSection.optimizedPathAveragedHeading!
      : currentVehicleSection.optimizedInitialHeading! +
        (currentVehicleSection.optimizedPathAvergaedSlope! *
          currentDatasnapshot.DistanceFromStartOfSection);

  currentDatasnapshot.RefrenceHeading =
    currentVehicleSection.sectionType == SectionType.Straight.name
      ? currentVehicleSection.pathAveragedHeading!
      : currentVehicleSection.initialHeading! +
        (currentVehicleSection.pathAvergaedSlope! *
          currentDatasnapshot.DistanceFromStartOfSection);

  // Lateral Distance and Accumulated Lateral Distance Calculation (we use these to decide if lane
departure happened)
  currentDatasnapshot.Theta = currentDatasnapshot.OptimizedRefrenceHeading -
    currentDatasnapshot.AveragedHeading;

  // LateralDistance is distance traveled in the perpendicular direction to the road.
  currentDatasnapshot.LateralDistance = currentDatasnapshot.Distance *
    sin(currentDatasnapshot.Theta * pi / 180); // Sin(Theta) = P / H
```