

Map-based Lane Keeping with HERE HD Live Map

Contents

Introduction	2
Map-based Lane Keeping	2
System Prerequisites	3
Procedure	4
Lane Keeping System	4
Parsing HERE HD Map Data – Lane Geometry Data	5
Receiving Geometric Pose Messages from a Localization Algorithm	8
Calculating Orientation Error and Lateral Velocity Error	8
Overall Process of Error Calculation	9
Finding the Closest Lane Point	10
Finding the Look Ahead Point	11
Calculating Error Main Function	13
Observations & Results	14
Test #1: Lane Lines Expanding into a Turn or Exit Lane	14
Test #2: An Intersection Without Lane Lines	16
Test #3: A Widening in the Lane	17
Challenges	19
Alternative Solutions	19
Conclusion	20
Conditions of this Report	21



Introduction

There are many challenging tasks when developing autonomous driving features to cope with the various changes to the environment. Often lane markings are faded or are covered with snow or dirt and can be difficult for a camera-based detection system. In this report, VSI addresses the application of HD map assets to improve the safety and performance of automated vehicle features within the context of lane keeping and trajectories.

VSI has been examining applications of HD maps in our test vehicle. In a previous report, we discussed map-based Adaptive Cruise Control (ACC) using the advised speed attributes from HERE's HD map data. In this report, we apply HERE's HD map data to a lane keeping application and examine performance of lane keeping with a map-based approach compared to a camera and computer vision-based approach.

Map-based Lane Keeping

Map-based lane keeping improves the performance and safety of automated driving systems, even Level 2 systems. Most Level 2 solutions available today localize and plan their trajectories based on cameras used to detect the lane markers on the roadway. This can be somewhat problematic since these systems are totally dependent on the presence of those lane markings.

An example of a problem that occurs when AV systems are performing computer vision only lane keeping is when there is a divergence in the lane lines as illustrated in Figure 1. The left image shows an example of when the vehicle momentarily changes its path to account for a widening in the lane. The result is annoying, although not particularly dangerous. In the second example (right), the AV system has

no understanding of the correct path to follow and under certain conditions the vehicle may veer to the right and follow the path of the exit ramp.

Mapping Assets Solve This Common Problem

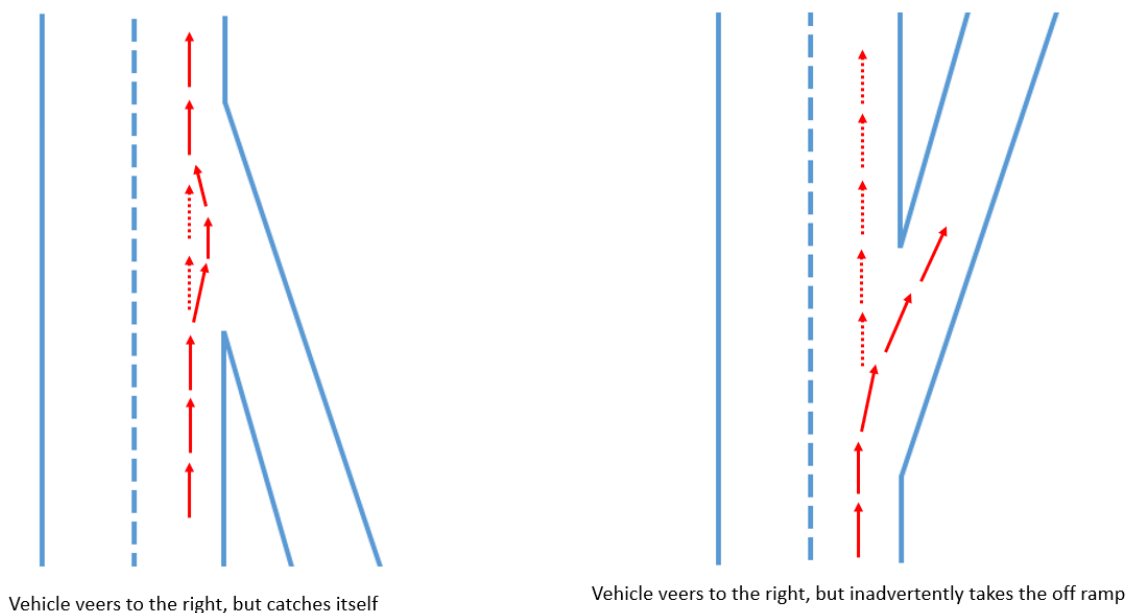


Figure 1: These illustrations show examples of common lane keeping problems with computer vision only AV systems. Map assets can solve these common problems.

Using a lane model from HERE's HD Live Map can solve these common issues involved in computer vision only lane keeping. Also, if cars are expected to self-drive in inclement weather, they will have to stay in their lanes even when the vision-based sensors can no longer see the lane lines. Using a lane model and localizing the vehicle in real-time enables the AV system to perform accurate and reliable lane keeping functions regardless of lane marking visibility.

System Prerequisites

VSI uses the 2016 Kia Soul as our base vehicle and Nvidia's Drive PX2 as a host AV computer. Prior to this project, we installed a GPS module, [Honeywell 1120 IMU](#), and [Velodyne 16 Puck LiDAR](#) in the test vehicle. In this project VSI applies all critical elements of the hardware stack coupled with HERE's HD Live Map for this lane keeping project.

Table 1: Basic Tools and Components

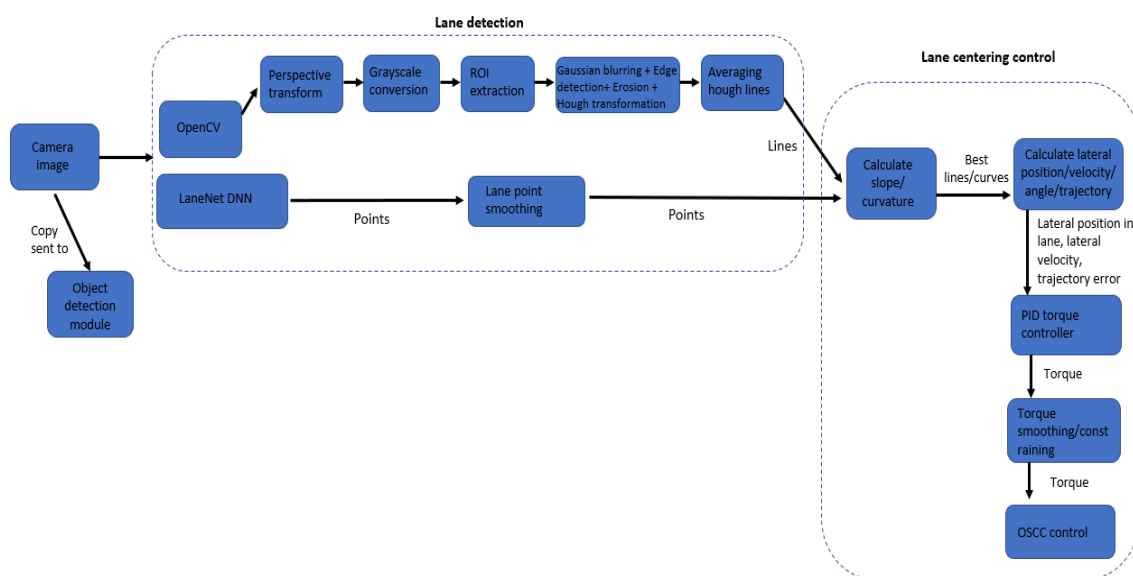
Basic Tools and Components		
Component/Tool	Partner/Supplier	Note
2014 Kia Soul	Kia	Base vehicle
Drive PX2	Nvidia	AV computer
Open Source Car Control (OSCC)	Polysync	Hardware interface
OSCC API	Polysync	Software interface
Kvaser leaf light	Kvaser	CAN bus interface
socketcan	Open source	Linux library for writing CAN messages
Ubuntu OS	Open source	AV ECU OS
GMSL camera	Leopard	Mono-camera
GPS	Adafruit	GPS module
HD Live Map	HERE	Digital maps
LiDAR	Velodyne	VLP-16 Puck
Inertial Measurement Unit (IMU)	Honeywell	HG1120 MEMS-based IMU

Copyright 2018 – Vision Systems Intelligence, LLC.

Procedure

Lane Keeping System

As an initial part of our AV build process, we created a lane detection and lane centering stack which implements [computer vision only lane keeping](#) on our research vehicle. The following diagram shows the overall architecture of our computer vision only lane keeping system.



Copyright 2018 – Vision Systems Intelligence, LLC.

Figure 2: This diagram shows the overall data flow and functions of VSI’s computer vision only lane keeping system.

For the project, VSI developed a new lane keeping system based on HERE’s HD lane model from the HD map. The following diagram shows the overall architecture of our map-based lane keeping system.

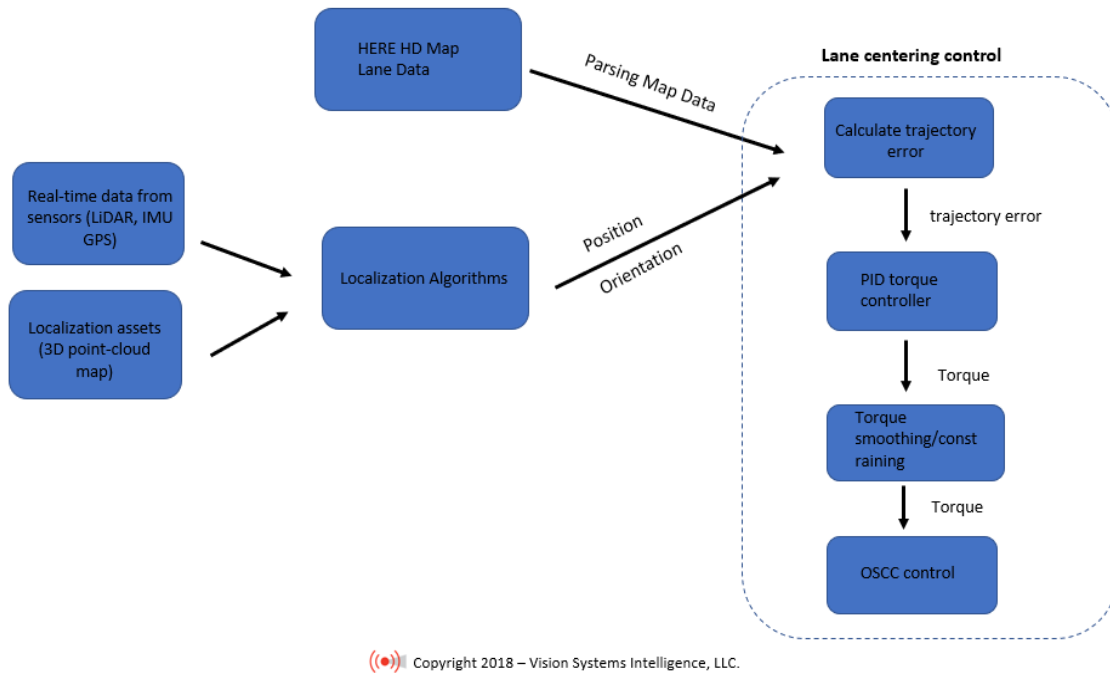


Figure 3: This diagram shows the overall architecture of VSI’s map-based lane keeping system

Parsing HERE HD Map Data – Lane Geometry Data

HERE’s HD Live Map consists of multiple layers of data delivered in a map-tile format. These layers are categorized into three main layers including Road Model, HD Lane Model, and HD Localization Model. There are also many sub-layers under these main categories. The Road Model sub-layers contain general road topology, road centerline geometry, and road-level attributes. The HD Lane Model sub-layers provide lane topology data and lane-level attributes. The HD Localization Model supports different localization approaches by providing various localization data.

To utilize the HD map to enhance ADAS or automated driving functionalities, we first need to understand what parts or layers of the map we need to use for a target application. For lane keeping, we use the layer called **lane-geometry-polyline** that has the precise lane geometry.

After identifying which parts and layers are to be used, the next process is downloading the map data. Since HERE’s HD Live Map is divided into many tiles covering different square areas, we manually

download the data from the tiles that we plan to test drive on. We only downloaded the layer lane-geometry polyline that we needed for this project.

The next process is developing a function for parsing the tile data in real-time. HERE uses the **Google Protobuf** format for storing the tile data. There are many options for storing and parsing this map data to be useable quickly by the AV system. We conducted this process by reading the tile data from the Protobuf into C++ vectors or custom structures with vectors, including only the layers and tiles required for this application. While parsing the map data, we decode **Morton codes** and perform bitwise exclusive-or on each subsequent value with the previous value in the sequence.

The Lane Geometry Tile that we process contains the geometry of lane boundaries and lane centerline paths. For our study, we only extract the lane centerline paths. Lane paths are stored as a sequence of 3D points that represent the center of the lane.

The following C++ pseudo-code represents how we parse HERE Protobuf Tile data that comes from their HD Live Map Lane Geometry Polyline layer.

```
#include <cstdint>
#include <iostream>
#include <fstream>
#include <string>
#include <algorithm>
#include "com/here/pb/hdmap/external/v1/lanes/layer-lane-geometry-polyline.pb.h"
#include "com/here/pb/hdmap/external/v1/lanes/layer-lane-attributes.pb.h"
#include "vsi/map_parsing/decode_morton_2d.h"

using namespace std;
using namespace com::here::pb::hdmap::external::v1::geometry;
using namespace com::here::pb::hdmap::external::v1::common;
using namespace com::here::pb::hdmap::external::v1::lanes;
using namespace com::here::pb::hdmap::shared::v1::lanes;

struct Coordinate {
    long double latitude;
    long double longitude;
};

struct Processed_lane_geometry {
    vector<Coordinate> coordinates;
    int id;
};

vector<Processed_lane_geometry> tileLanes;
```

```
//Parse Link Layer Data
void parseLaneTileData(const LaneGeometryLayerTile& tile) {

    ///creating our own ID to differentiate each lane centerline in a lane group
    int lane_id = 0;

    int64_t current_2d_encoding_int = tile.tile_center_here_2d_coordinate();
    pair<double, double> decoded_tile_center = decode_morton_2d(current_2d_encoding_int);

    //Loop through each Lane Group
    for (int i = 0; i < tile.lane_group_geometries_size(); i++) {
        LaneGroupGeometry groupGeometry = tile.lane_group_geometries()[i];

        /////Loop Through Each Lane Path
        for (int j = 0; j < groupGeometry.lane_geometries_size(); j++) {

            double lat = decoded_tile_center.second; double lon = decoded_tile_center.first;
            current_2d_encoding_int = tile.tile_center_here_3d_coordinate().here_2d_coordinate();
            LineString3dOffset theLanePathGeometry =
            groupGeometry.lane_geometries()[j].lane_path_geometry();

            //Creating a new Lane and adding it an ID
            Processed_lane_geometry new_path;
            new_path.id = lane_id;

            ///Loop through all the points in the path
            for (int k = 0; k < theLanePathGeometry.here_2d_coordinate_diffs().size(); k++) {

                int64_t here_2d_coordinate_diff = theLanePathGeometry.here_2d_coordinate_diffs()[k];
                int64_t next_coordinate_int = (here_2d_coordinate_diff ^ current_2d_encoding_int);
                std::pair<double, double> decoded = decode_morton_2d(next_coordinate_int);

                /// Saving for next iteration
                current_2d_encoding_int = next_coordinate_int;

                //Creating a new coordinate and adding it to the link
                coordinate new_coordinate;
                new_coordinate.latitude = decoded.second;
                new_coordinate.longitude = decoded.first;
                new_path.coordinates.push_back(new_coordinate)
            }
            //Adding the new Lane path to the list/vector of processed Lane geometry
            tileLanes.push_back(new_path);

            lane_id++;
        }
    }
}
```

The next step is to use this Lane Geometry data along with the vehicle's precise position and orientation to calculate how the ego vehicle should adjust its steering (steering wheel) to stay aligned with the center of the lane. We compare the difference between the trajectory we want the vehicle to take and the vehicle's current trajectory which we call the *trajectory error*.

Receiving Geometric Pose Messages from a Localization Algorithm

In order for sensor data and other important information to be communicated among the nodes involved in this project, we encapsulate them as **ROS nodes**. ROS, which stands for **Robotic Operating System**, provides a robust and efficient communications framework. With our localization algorithm engaged, we begin receiving a ROS pose message from the node for localization; this message includes our current position and orientation values. We translate these into usable formats, namely latitude and longitude for our position and Euler angles for our orientation.

```
/* Receive geometric pose messages from ROS, translate position and orientation into usable
formats and pass them on */
void localizationCallback(const geometry_msgs::PoseStamped msg) {

    //extract geometric position and orientation
    geometry_msgs::Pose pose = msg.pose;
    geometry_msgs::Point position = pose.position;
    geometry_msgs::Quaternion orientation = pose.orientation;

    //translate position into Latitude and Longitude
    coordinate current_position;
    translateIntoCoordinate(position.x, position.y, current_position);

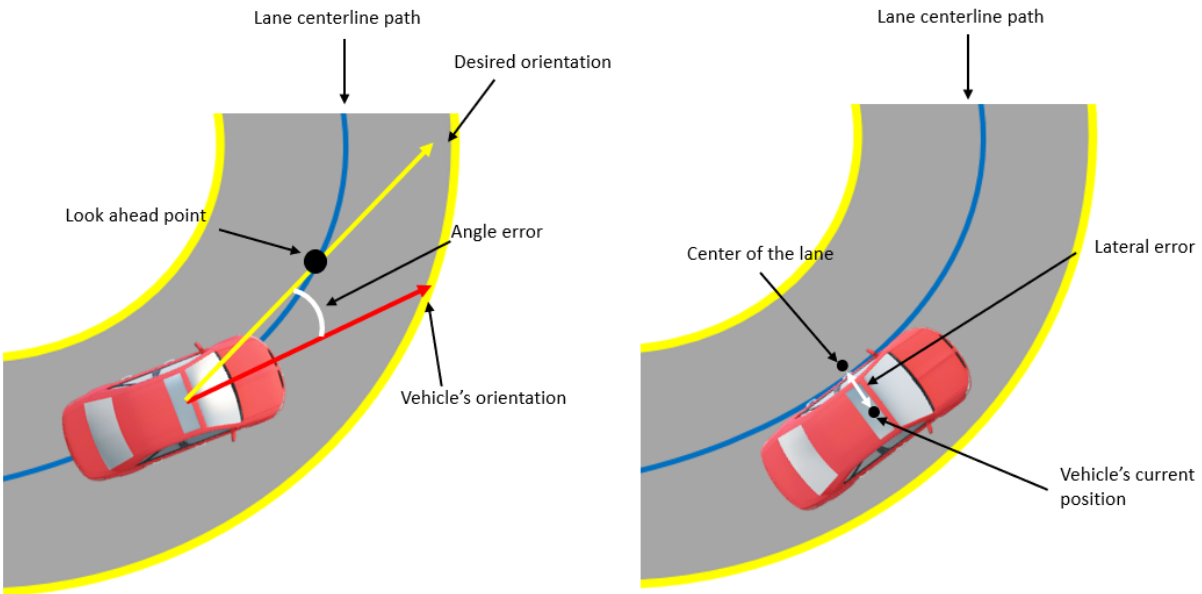
    //translate orientation into Euler Angle
    double yaw;
    double pitch;
    double roll;
    toEulerAngle(orientation, yaw, pitch, roll);
    double vehicle_orientation = yaw;

}
```

Calculating Orientation Error and Lateral Velocity Error

We developed a function called *calcError* which calculates the difference in the vehicle's orientation from the orientation which would keep the vehicle on the lane (*angle_error*), as well as calculating the vehicle's lateral displacement from the center of the lane (*lateral_error*). This function takes in and uses a list of coordinates that represent the center of the lane the vehicle is following (*lane_centerline_path*), the vehicle's current position (*current_position*), and the vehicle's orientation (*vehicle_orientation*).

Calculating Angle Error and Lateral Error




 Copyright 2018 – Vision Systems Intelligence, LLC.

Figure 4: These images show how to calculate angle error and lateral error values for VSI’s lane keeping system

These output error values are passed to **PID controllers**, through a processing pipeline and eventually to vehicle control systems. In our PID control algorithms, VSI uses a variety of techniques to track trajectory error over time and to calculate proper torque commands with the received information.

The following subsections describe details of the error calculation process.

Overall Process of Error Calculation

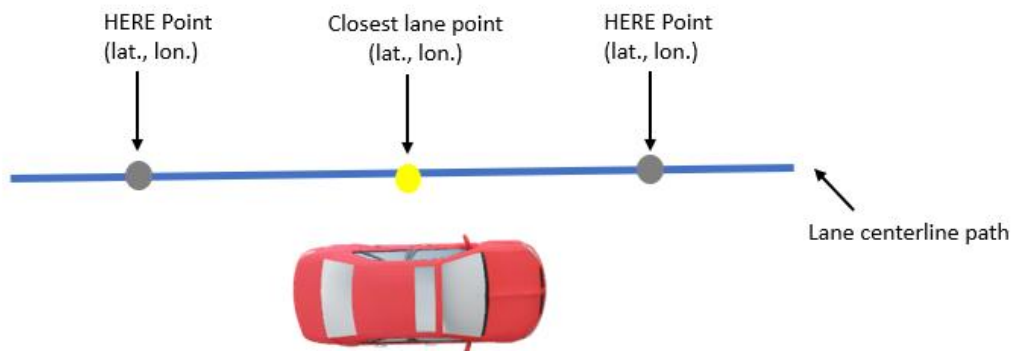
- To find the *angle_error*, we must first find the *desired_angle* which would keep the vehicle closely following the center of the lane.
- To find this angle, we find a point on the path that represents the center of the lane we are following some fixed distance ahead (*look_ahead*). We call this the *lookAheadPoint*.
- We find this *lookAheadPoint* by traversing the lane centerline path by the fixed distance (*look_ahead*) starting at the point on the centerline path that is closest to the vehicle’s current position. It is important to note that the coordinates in the HERE map that represent the centerline path are distributed sparsely for minimal data size. Thus, VSI calculates the true closest point on the lane (called *closestPoint*) by using nearby lane points in HERE’s dataset.

- By using the *lookAheadPoint* and the vehicle's position, we calculate the *desired_angle* necessary to keep on the lane.
- Comparing the *desired_angle* and the current vehicle orientation we get the *angle_error* from the desired path.

Finding the Closest Lane Point

To find the closest point on the lane centerline, two coordinates from HERE's data are used: The closest point on the centerline path preceding the vehicle, and the closest point on the centerline path following the vehicle (*nearest_HERE_point* and *second_nearest_HERE_point*).

We loop through the lane points and store the point with the minimum distance to the vehicle's current position. Once we have found the closest lane point, we use the vehicle's current orientation to find the point on the opposite side of the vehicle by looking at the difference in latitude and longitude. These two points will be used to project a point that lies between these two points and is closest to the vehicle's position.



 Copyright 2018 – Vision Systems Intelligence, LLC.

Figure 5: This image shows how to calculate the closest lane point by using HERE's data

Most of the time, the closest point on the lane will lie between two lane points rather than being a lane point itself. In the *findClosestProjectedPoint* method, we use linear algebra to project a line between the two points found in the *findClosestHEREPoints* method, as well as a line emanating from the vehicle's position that is perpendicular to the lane centerline where it intersects. The intersection point of these

lines represents the *closestPoint* on the lane to the vehicle. This point is returned in a coordinate called *closestPoint*, which is passed in by reference.

```
void findClosestProjectedPoint (coordinate point1, coordinate point2, coordinate
current_position, coordinate& closestPoint) {
    long double lon1 = point1.longitude;
    long double lat1 = point1.latitude;

    long double lon2 = point2.longitude;
    long double lat2 = point2.latitude;

    long double current_lon = current_position.longitude;
    long double current_lat = current_position.latitude;

    //using equation[  $y = m(\text{slope}) + b$  ]
    long double m = (lat2 - lat1) / (lon2 - lon1);
    if (lon2 - lon1 == 0) { // if m is infinite
        m = 1000000;
    }
    long double b = lat1 - m * lon1;
    long double m_perpendicular = -1 * (lon2 - lon1) / (lat2 - lat1);
    if (lat2 - lat1 == 0) { // if m_perpendicular is infinite
        m_perpendicular = -1000000;
    }
    long double b_perpendicular = current_lat - m_perpendicular * current_lon;

    closestPoint.longitude = (b_perpendicular - b) / (m - m_perpendicular);
    closestPoint.latitude = m * lon1 + b;
}
```

Finding the Look Ahead Point

When steering a vehicle to follow a lane, steering is not based solely on minimizing the distance between the vehicle and the center of the lane at the current moment but is also based on orienting the vehicle so that it will be centered in the lane at a point shortly in the future, which is called the *lookAheadPoint*. Finding a point further ahead in the lane is especially crucial when the lane curves, so that the vehicle can make proper steering adjustments ahead of time. In order to find this *lookAheadPoint*, we use the *look_ahead* value in-conjunction with the *closestPoint* coordinate to find two HERE coordinates in the *lane_centerline_path*, one coordinate that has a distance slightly less than the *look_ahead* value, *preLookAheadPoint*, and one coordinate that has a distance slightly greater than the *look_ahead* value, *pastLookAheadPoint*. We use these two coordinates and the *look_ahead* distance value to calculate the *lookAheadPoint*.

```
void findLookAheadProjectedPoint(coordinate closestPoint, int forward_point_index,
vector<Coordinate> lane_centerline_path, coordinate &lookAheadPoint) {
    //find a lane point past the look_ahead distance which we'll use as a triangulation point
    double overflowDistance;
    coordinate preLookAheadPoint = closestPoint;
    coordinate pastLookAheadPoint; // used for triangulating look ahead projected pt
    for(int i = forward_point_index; i < lane_centerline_path.size(); i++){
        double dist = getDistanceAlongPath(lane_centerline_path.at(i), closestPoint);
        if(dist > look_ahead) {
            pastLookAheadPoint = lane_centerline_path.at(i);
            preLookAheadDistance = getDistanceAlongPath(lane_centerline_path.at(i-1),
closestPoint);
            overflowDistance = look_ahead - preLookAheadDistance;
            break;
        } else { preLookAheadPoint = lane_centerline_path.at(i); }
    }

    // calculate LookAheadPoint using linear algebra
    long double end_lon = pastLookAheadPoint.longitude;
    long double end_lat = pastLookAheadPoint.latitude;

    long double start_lon = preLookAheadPoint.longitude;
    long double start_lat = preLookAheadPoint.latitude;

    long double m = (end_lat - start_lat) / (end_lon - start_lon);
    if (end_lon - start_lon == 0) {
        if (end_lat - start_lat > 0) m = 1000000;
        else if (end_lat - start_lat < 0) m = -1000000;
    }

    long double lon_diff = sqrt(pow(look_ahead, 2) / (pow(m, 2) + 1)); // always positive
    long double lat_diff = m * lon_diff;

    long double look_ahead_lon;
    long double look_ahead_lat;
    if (end_lon - start_lon >= 0 && end_lat - start_lat >= 0) { // Q1
        look_ahead_lon = start_lon + lon_diff;
        look_ahead_lat = start_lat + lat_diff; // lat_diff is positive
    }
    else if (end_lon - start_lon <= 0 && end_lat - start_lat >= 0) { // Q2
        look_ahead_lon = start_lon - lon_diff;
        look_ahead_lat = start_lat - lat_diff; // lat_diff is negative
    }
    else if (end_lon - start_lon <= 0 && end_lat - start_lat <= 0) { // Q3
        look_ahead_lon = start_lon - lon_diff;
        look_ahead_lat = start_lat - lat_diff; // lat_diff is positive
    }
    else if (end_lon - start_lon >= 0 && end_lat - start_lat <= 0) { // Q4
        look_ahead_lon = start_lon + lon_diff;
        look_ahead_lat = start_lat + lat_diff; // lat_diff is negative
    }

    lookAheadPoint.longitude = look_ahead_lon;
    lookAheadPoint.latitude = look_ahead_lat;
}
```

Calculating Error Main Function

This is the code snippet of the whole process of orientation and lateral error calculations. After calculating errors, these error values are passed to PID controllers for final control processing.

```
//calculate angle_error - difference in vehicle orientation from orientation which would
keep on lane
//calculate lateral_error - difference in lateral position required to re-encounter the lane
long double angle_error;
long double lateral_error;
calcError(current_position, vehicle_orientation, &angle_error, &lateral_error,
lane_centerline_path);

//These error values are then passed to PID controllers for final control processing
//Calculate angular and lateral error necessary to follow the lane
void calcError(coordinate vehicle_position, long double vehicle_orientation, long double*
angle_error, long double* lateral_error, vector<Coordinate> lane_centerline_path) {

    //find closest HERE points to calculate the true closestPoint
    coordinate nearest_HERE_point;
    coordinate second_nearest_HERE_point;
    findClosestHEREPoints(lane_centerline_path, vehicle_position, &nearest_HERE_point, &
second_nearest_HERE_point);

    //calculated true closest point along lane centerline
    coordinate closestPoint;
    findClosestProjectedPoint (nearest_HERE_point, second_nearest_HERE_point,
vehicle_position, &closestPoint);

    //find a coordinate on a lane segment at a distance of Look_ahead
    coordinate lookAheadPoint;
    findLookAheadProjectedPoint(closestPoint, lane_centerline_path, &lookAheadPoint);

    //find desired angle
    long double diff_lat = lookAheadPoint.latitude - vehicle_position.latitude;
    long double diff_lon = lookAheadPoint.longitude - vehicle_position.longitude;
    long double desired_angle = atan(diff_lat/diff_lon);

    //convert desired_angle to a Euler angle from the coordinate space
    if(diff_lat >= 0 && diff_lon >= 0); //Quadrant 1 - Do Nothing
    else if(diff_lat >= 0 && diff_lon <= 0) // Q2 - pi + currentAngle
        desired_angle = M_PI + desired_angle;
    else if (diff_lat <= 0 && diff_lon <= 0) // Q3 - (-pi) + currentAngle
        desired_angle = -1 * M_PI + desired_angle;
    else if (diff_lat <= 0 && diff_lon >= 0); //Quadrant 4 - Do Nothing

    //calculate angular error
    if(desired_angle > M_PI_2 && vehicle_orientation < 0)
        angle_error = abs(desired_angle + vehicle_orientation) * -1.0;
    else if(desired_angle < 0 && vehicle_orientation > M_PI_2)
        angle_error = abs(desired_angle + vehicle_orientation);
    else angle_error = desired_angle - vehicle_orientation;
    //calculate lateral_error
    lateral_error = getDistance(vehicle_position, closestPoint);
    if (angle_error < 0) lateral_error *= -1;
}
```

Observations & Results

VSI tested our lane keeping system based on HERE's HD Live Map data at various locations. In this section, we discuss our test results by comparing how lane keeping performs when using an HD map versus a computer vision only approach. We also address some challenges with map-based ADAS in this section.

Test #1: Lane Lines Expanding into a Turn or Exit Lane

VSI tested our lane keeping system with and without map data on a local road near our office. The left picture in Figure 6 shows the satellite image of the test location. The picture in the middle shows lane information of the road from HERE's HD map dataset. It shows the lane markings (yellow lines) and centerline (green lines) information of each lane. The right image shows a point-cloud map overlaid with the lane centerline the vehicle is following from HERE's HD map data. Our test vehicle approached the intersection shown on the top part of the picture from the south.

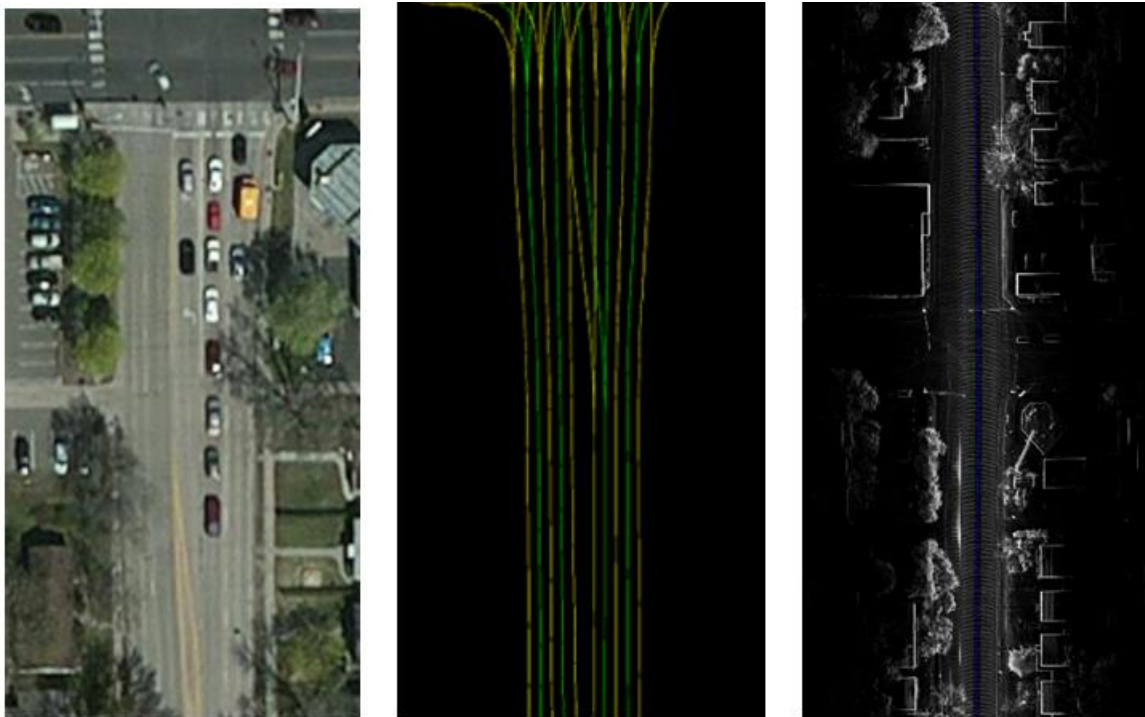


Figure 6: These pictures show different map images of the test location. The left picture shows the satellite image of the test location. The picture in the middle shows lane information of the road from HERE's HD map dataset. The right picture shows a point-cloud map overlaid with the lane centerline the vehicle is following from HERE's HD map data.

The following images show the recorded paths the vehicle traveled with the lane keeping system engaged. From these images you can see that the map-based lane keeping system (left) continued to stay in the center of its non-turn lane as the yellow lane line expanded into the left turn only lane. However, the computer vision only lane keeping system (right) veered into the turn lane requiring the human driver to disengage the system and direct the vehicle back into the correct lane. This is a common behavior of camera-only lane keeping systems when approaching turn lanes or exit lanes. The algorithm finds it difficult to distinguish which lane markings are meant for it to follow in order to maintain the current lane when the lane line has expanded into a completely new and undesirable lane to be in. This behavior can be annoying for passengers and could lead to a potential accident as the vehicle begins following an undesirable path.

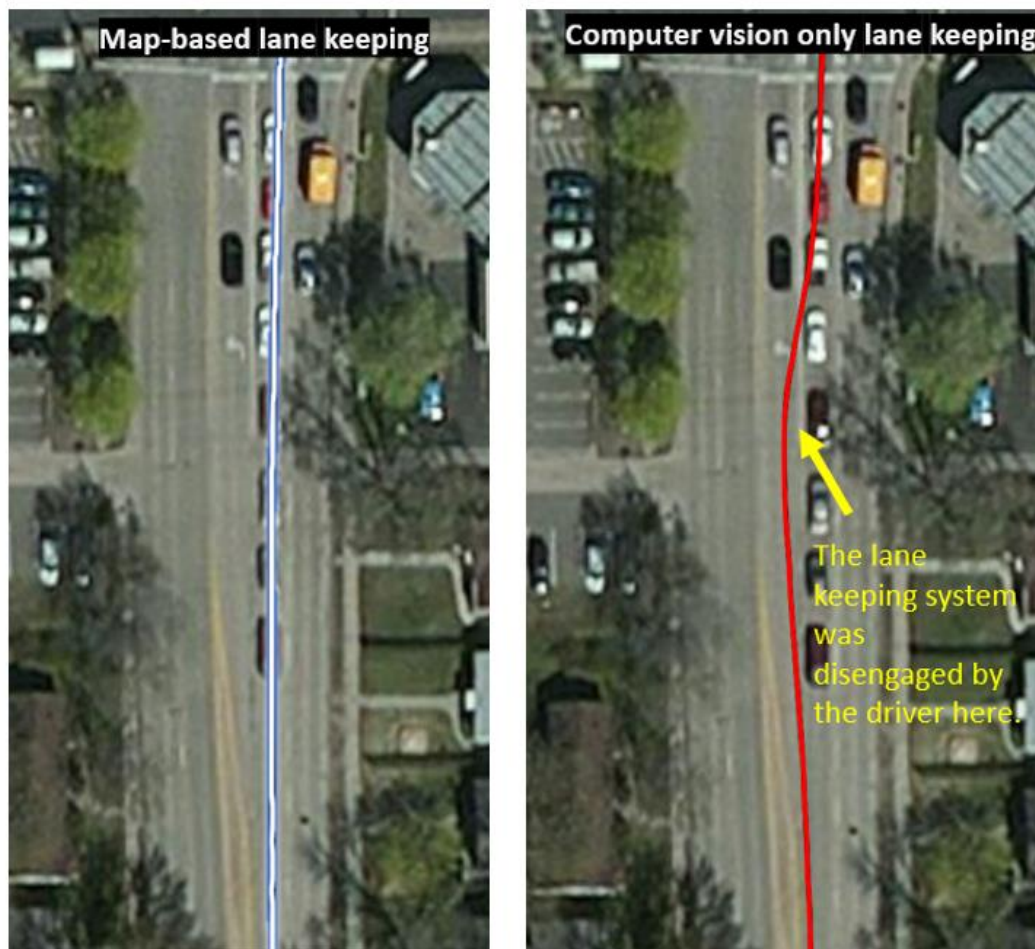


Figure 7: These images show a comparison of the two test results. The left picture shows the test result of the map-based lane keeping system by using HERE's HD map data. The right picture shows the test result of the computer vision only lane keeping system.

Test #2: An Intersection Without Lane Lines

In this scenario, we tested our lane keeping system at an intersection where there is a break in lane lines. Figure 9 shows a comparison of the recorded paths the vehicle traveled with the computer vision only lane keeping system (red) and the map-based lane keeping system (green). The vehicle approached the intersection shown on the bottom part of the image from the east (left).



Figure 8: This picture shows the satellite image of the test location. VSI tested our lane keeping system at an intersection where there is a break in lane lines.

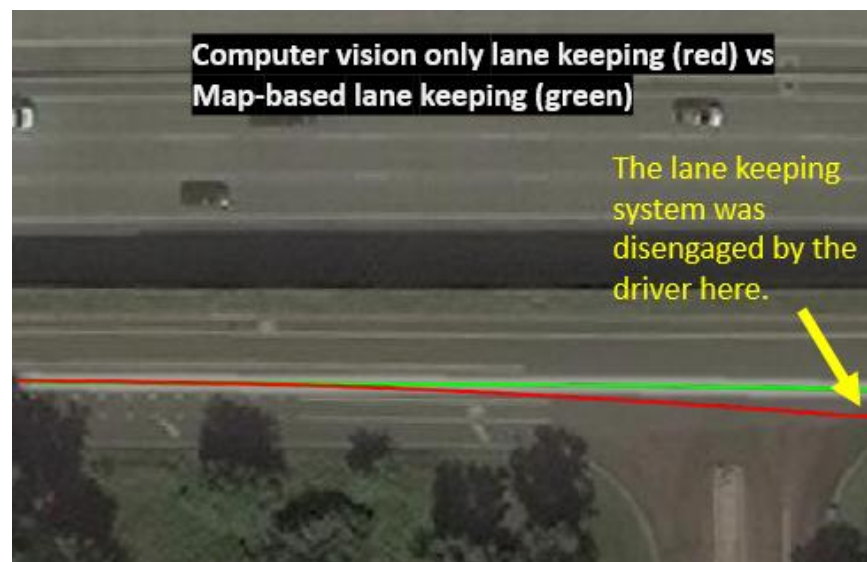


Figure 9: This image shows a comparison of the recorded paths the vehicle traveled with the computer vision only lane keeping system (red) and the map-based lane keeping system (green).

The map-based lane keeping system stayed in the desired trajectory which is driving straight and proceeding through the intersection. On the other hand, the computer vision only lane keeping system

gradually deviated when the system could not detect any lane lines between the end of the lane lines that the vehicle was following and the beginning of the lane lines in the opposite side of the intersection. The vehicle proceeded through the intersection, but it veered into the right resulting in the human driver disengaging the system.

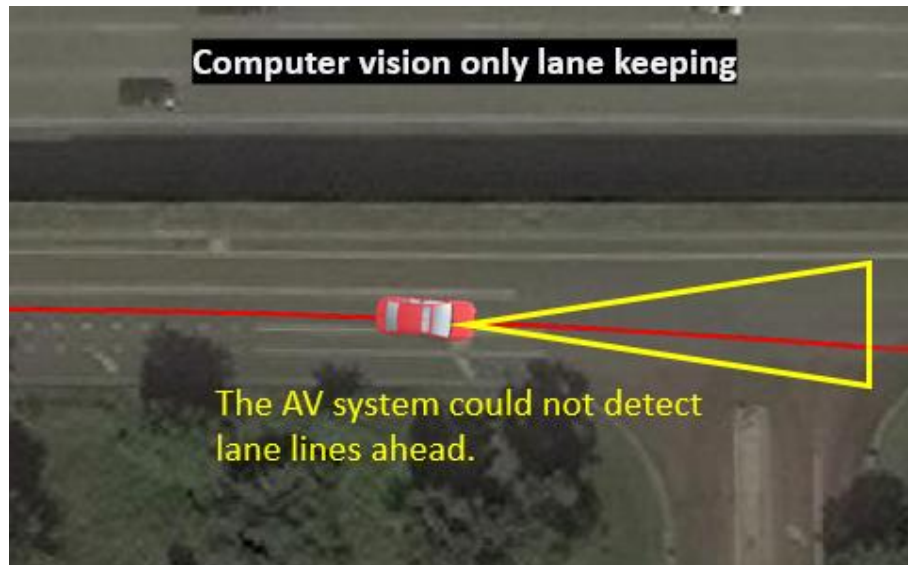


Figure 10: This image illustrates that a computer vision only lane keeping system cannot provide proper steering control without any lane lines ahead.

This example illustrates that a computer vision only lane keeping system does not know the correct path to follow when it cannot see the lane lines on the road. The system becomes blind and cannot provide proper steering control even in a simple intersection like this test location. This will become problematic to use computer vision only keeping systems especially on local roads.

Test #3: A Widening in the Lane

In this location, we tested our lane keeping system on a local road where there is a widening in the lane. Our vehicle traveled from the east to the west (left to right) and intended to drive straight and pass the intersection shown on the bottom part of the image. Figure 12 shows a comparison of the recorded paths the vehicle traveled with the computer vision only lane keeping system (red) and the map-based lane keeping system (green).



Figure 11: This picture shows the satellite image of the test location. VSI tested our lane keeping system on a local road where there is a widening in the lane.



Figure 12: This image shows a comparison of the recorded paths the vehicle traveled with the computer vision only lane keeping system (red) and the map-based lane keeping system (green).

As the vehicle approached the intersection, the right lane line which the vehicle was following, extended towards the outside of the curb creating a space for a right-turn. While the map-based lane keeping stayed in the desired trajectory regardless the changes of the lane markings, the computer vision only lane keeping system followed the right lane line and slightly changed its path. The vehicle veered into the right side of the road resulting in the human driver disengaging the system to prevent an accident.

This behavior of computer vision only lane keeping systems can be also seen in other situations like merging lanes shown in Figure 1. When a computer vision only system follows the misleading lane markings, it changes its path momentarily even though that is not where the vehicle should be driving.

In some cases, the system fixes its path on its own by finding new lane markings to follow and returns to

a desired trajectory. However, that is still not an ideal behavior and would not be comfortable for passengers.

Challenges

One of the key challenges with HD maps is to keep map data up-to-date. The map needs to be updated frequently and accurately. Also, it is important for AVs to understand when the map data has been updated and how reliable the data is.

HERE's HD Map Live offers a technique called the **Quality Index** to provide information about how recent and certain the data is. For example, there is recent construction and changes to the lane boundaries on a given road segment (called a Link). If HERE has received enough confirmations about the change from the fleet vehicles that have recently driven through this Link, then the Link is marked with a higher value in the quality index. On the other hand, if there have been no fleet vehicles that have recently driven through this Link, the Link's quality is shown as a low-quality index. It is useful for AV systems to receive such dynamic information so that the systems can be programmed to use their on-board sensors or alert the human driver to retake control in certain parts of the map that are not up-to-date or have a low-quality index.

In addition, in order to make use of map data more accurately and effectively with AV systems, it requires accurate and reliable localization. However, to achieve high precision localization, it could require expensive hardware or additional processing power and software components for AV systems which might not be realistic for series production consumer vehicles.

Alternative Solutions

As an alternative, a different kind of localization method could have been used for this map-based lane keeping project such as an RTK-based solution or a visual landmark matching solution.

Additionally, VSI designed our map-based lane keeping system to follow lane centerline paths, but a system could also be designed to use the lane boundaries and steer accordingly to stay centered between the lane boundaries.

It is also important to note that this VSI research project demonstrates the potential and advantages of using HD maps with LiDAR to enhance lane-level functions. However, there are alternative or supplementary approaches to achieve similar results that may differ from our approach.

- For example, common approaches using correction service rely on GNSS/RTK/IMU solutions against a lane model.
- While VSI used LiDAR, it is possible to perform localization with cameras instead of LiDAR.
- Alternative approaches likely use a combination and fusion of both map-based lane keeping and computer vision/real-time sensing-based lane keeping rather than lane keeping based solely on map data.
- These vehicles' map-based lane keeping systems could be linked to a more advanced routing engine for path planning and following which enables the AV system to understand what lane segments the vehicle is driving on and what segments the vehicle is going to be driving on next.
- These vehicles would be most likely to feature some form of connectivity such as a cellular connection and a system that dynamically downloads new map updates and new map tile data.

Conclusion

- In the context of automated driving, precision maps are used for various tasks such as path planning and localization. Map data can also improve the performance and safety of automated driving systems.
- Computer vision only lane keeping systems tend to get confused and make errors in a vehicle's trajectory when lane lines are out of the ordinary or invisible. Using a lane model from an HD map can solve common issues involved in computer vision only lane keeping.
- VSI observed that our map-based lane keeping system provided more reliable and consistent performance over our computer vision only lane keeping system during the tests. The map-based system operated properly in situations like a lane with misleading lane markings and an intersection without any lane lines.
- One of the key challenges with HD maps is to keep map data up-to-date. Also, it is important for AV systems to know how recent and reliable the map data is in order for these systems to make proper judgments such as using the data from on-board sensors instead of the map data or requesting the human driver to take over in areas with a low-quality index.



Conditions of this Report

In no event shall Vision Systems Intelligence, LLC. (aka. VSI, or VSI Labs) including partners, owners employees or agents, be liable to you or any other party for any damages, losses, expenses or costs whatsoever (including without limitation, any direct, indirect, special, incidental or consequential damages, loss of profits or loss opportunity) arising in connection with your use of this information. The information was conducted on a best efforts basis. No guarantee can be made about the completeness or accuracy of the information appearing in this report. VSI shall be held harmless for claims against HERE resulting from these experiment results, or this paper.