Part1: Controller

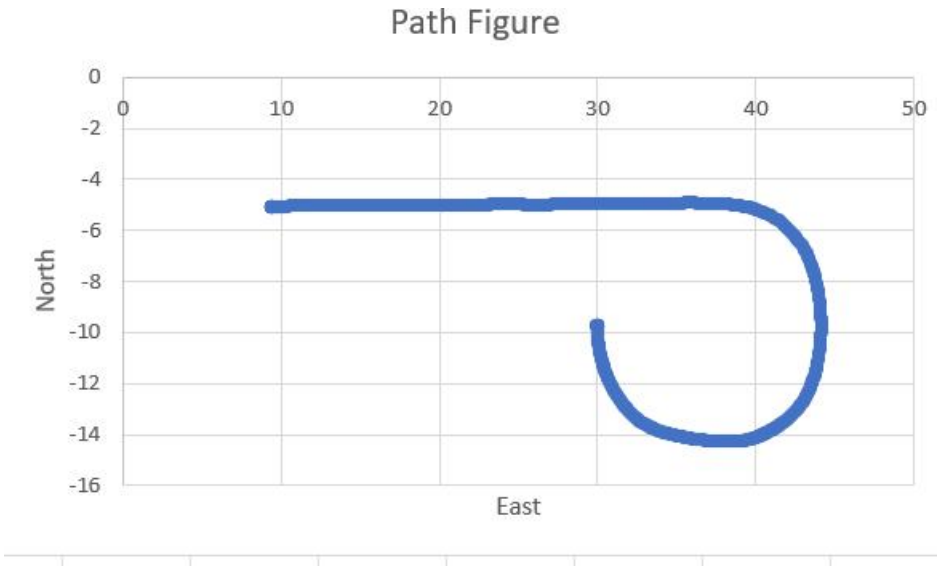
Video Link: https://youtu.be/h6O3qAGtVFc

What we want to do:

1. Design better control methods, decrease the cross-track error in lane keeping, and compare the performance of different controllers.

What we did:

1. Generate the lane path from A point to B point
    - Drive the car to follow the lane along a straight line and a circle. Use GPS to record the position, heading, and velocity of the vehicle every control period. (Every 10 ms)
    - The figure of the path looks like this



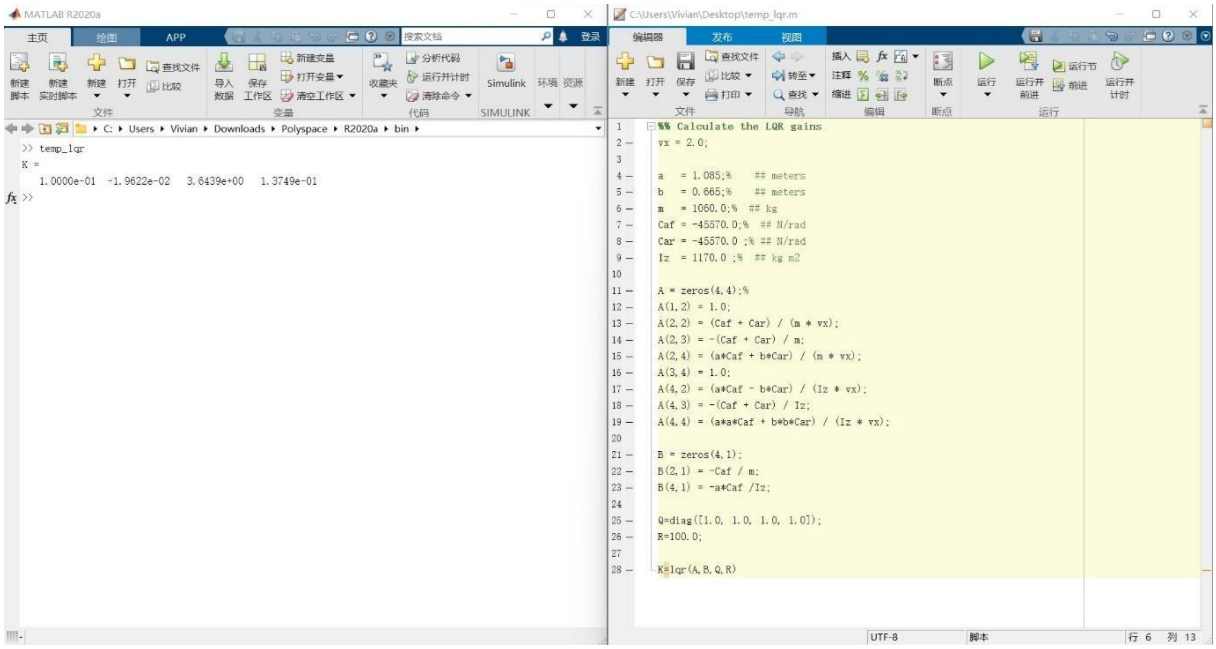
2. Use the demo code, pure pursuit controller to control the vehicle following the waypoints.

- Performance: The average cross-track error when making a turn is around 3.1 meters.
- Analysis: PP controller is just a kind of proportional control method. If the coefficient of proportionality is set as a relatively small value to ensure the stability of the system, the stable error will be large and cannot be eliminated.

3. Design another control method: LQR controller based on the vehicle kinematics model
    - Build the vehicle kinematics model.
    - Design the LQR solver to calculate the discrete Riccati Equation.
    - Adjust parameters in the Q matrix and R matrix. Test the performance with different Q/R matrices and identify a set of parameters, by which we can get the best performance.
    - Compute the LQR gain matrix, K, and state matrix, X, to get the front wheel angle, which is the input of the system.

    - Performance: The average cross-track error when making a turn is around 1.9 meters.
    - Analysis: LQR is a kind of optimal control method, using state feedback. The performance is better than the Pure pursuit controller. But we cannot decouple the lateral control from the longitudinal control, just using vehicle kinematics model. So, lateral control and longitudinal control influence on each other. This coupling increases the cross-track error.

4. Look for another control method: LQR controller based on the vehicle dynamics model
    - Build the vehicle dynamics model: we choose the model based on errors
    - Decide the parameters in the model:
        - Collect data of the vehicle states, control input and path kappa when following waypoints to compute the errors and the rate of errors every control period. (Every 10 ms)
        - Fit the data by the Least Square method to estimate the dynamics parameters in the model
    - Implement feedforward control to compensate for the disturbance matrix, which is caused by the path kappa, in the vehicle model.
    - Compute LQR gains offline using MATLAB, the code and result are shown below.



- Performance: The average cross-track error when making a turn is around 0.5 meters.
- Analysis: The performance is better than the Pure pursuit controller and LQR based on the kinematics model. We use the vehicle dynamics model to decouple the lateral control from the longitudinal control. Also, the response is faster because the gains are computed offline. So, the cross-track error can be smaller.

Code:

The projectlane.csv contains the path waypoints we recorded.
The Pure_pursuit_tracker_pid.py, LQR_kinematics.py, LQR_dynamics.py are the pure pursuit controller, LQR controller based on the vehicle kinematics model, LQR controller based on the vehicle dynamics model respectively. The guideline to run the python files is shown below.

$ cd ~/group1/
$ catkin_make

$ source devel/setup.bash
$ roslaunch basic_launch gem_sensor_init.launch

$ source devel/setup.bash
$ roslaunch basic_launch gem_dbw_joystick.launch

Take GNSS-based waypoints follower with LQR_kinematics as an example

$ source devel/setup.bash

$ rosrun gem_gnss_control LQR_kinematics.py

Some afterthoughts:

1. Compensate delay: The GEM vehicle unavoidably exists delay like the time that the steering wheel turns, which influences the accuracy of the controller. So, we would like to design a prediction module to compensate the delay caused by the physical parts of the vehicle.
2. Performance at different levels of speed: The performance of the controller would be worse when the vehicle moves faster. So, we would like to improve our controller to adapt a wide range of speed, getting good performance as speed increases.

Part2: Lane Detector

We utilize two different methods to realize the lane detector and show the results on the recorded rosbag.

Code Location:
The relevant code is located in lane_detector_part.

The first method:
Hough Transform based lane detector

Description:
The first method is based on hough transform. It could detect the straight line well but has no ability to detect the curve.

Video Link:
https://youtu.be/hXn8bDd7wPk

Steps:

1. Convert the Image from RGB to HSL color space.
2. Generate the mask by thresholding the white color(lane)
3. Get the image with only white color by mapping the mask to original image
4. Use Gaussian Blur to fine tune image
5. Use Canny edge to generate edges of white lanes
6. Set region of interest, only take the region which is in front of the car in consideration
7. Use hough transform to find straight lines in region of interest
8. Separate the left and right lines
9. Generate left and right lanes by using two points of detected lines to fit the linear equation by Linear regression

10. Sample points on left and right lanes and get the average points which should be in front of the car to indicate the next move.
11. Get the depth information of detected middle point from the depth image.
12. Calculate the distance of the middle points and send it as the topic for controller.

Results:

The proposed method detects the straight line well. But it can't detect the curve.

The second method:
The second method is based on UKFKalman to fit a cubic plane curve.

Description:
The proposed method uses the same preprocess method as method one to get the white lines. But it utilizes the DCSACN cluster algorithm to identify the left and right lanes. It utilizes detected points to fit the cubic plane curve to the left and right lanes respectively. It utilizes UKFKalman to keep the stability of the detection.

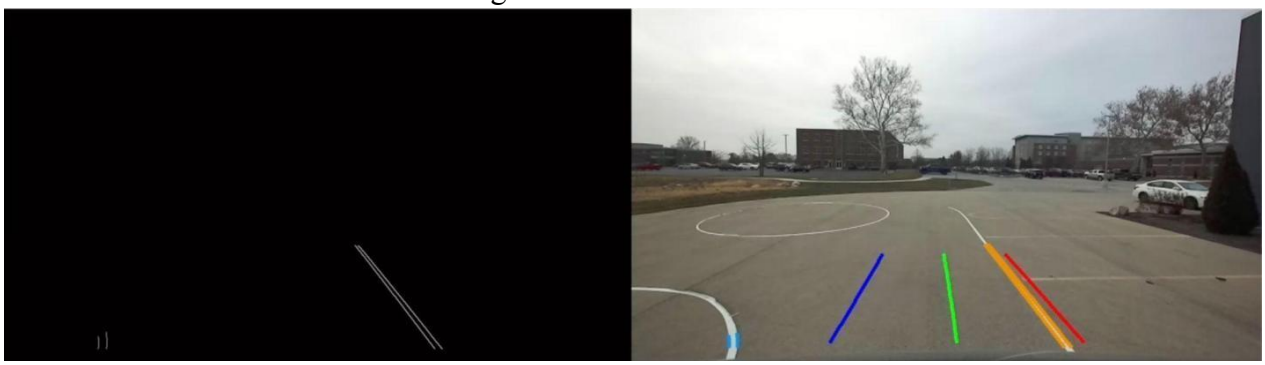
Video Link:
https://youtu.be/4RBllkrjGfQ

Steps:
1. Convert the Image from RGB to HSL color space.
2. Generate the mask by thresholding the white color(lane)
3. Get the image with only white color by mapping the mask to original image
4. Use Gaussian Blur to fine tune image
5. Use Canny edge to generate edges of white lanes
6. Set region of interest, only take the region which is in front of the car in consideration
7. Use DCSCAN cluster algorithm to identify the left and right lanes
8. If detected points of the left and lanes is not NULL
    a. Use detected points to fit cubic plane curve
    b. Use UKFKalman to adjust the fit cubic plane curve incase of the dramatic fluctuations between former and current frames.
9. If one of left or right lanes is NULL
    a. Fit cubic plane curve for detected lane
    b. Made a horizontal shift to generate undetected lane
10. Sample points on left and right lanes and get the average points which should be in front of the car to indicate the next move.
11. Get the depth information of the detected middle point from the depth image.
12. Calculate the distance of the middle points and send it as the topic for the controller.

Results:
In Fig 1, The left image is the image in step 6 where the image has undergone processes including the threshold of white color, canny edge, and region of interest, which represents the information used to detect lanes.
The right image is the final result. The Green lane is the generated middle lane which is used to direct the orientation of the car. The white blue and the origin represent the left and right lanes identified by the DCSAN algorithm. The dark blue and red represent the cubic plane curve we fit for the left and right lanes according to detected points. We calculated the average of left and right lanes to generate the middle lane.
When the lanes are complete and there are no noises in the preprocess image. The generated result is good. Both left and right lanes fit well and the final middle lane is good too.

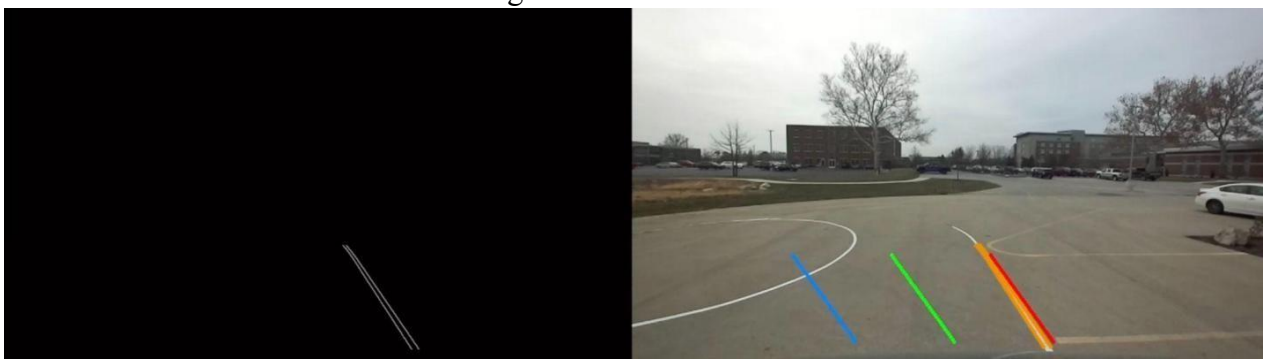
(Fig 1)

In Fig 2, With the help of UKFKalman, Even if one of the lanes is not complete, the lane detector could still keep stability according to the information of the last frame and fuse the current information with different weights.



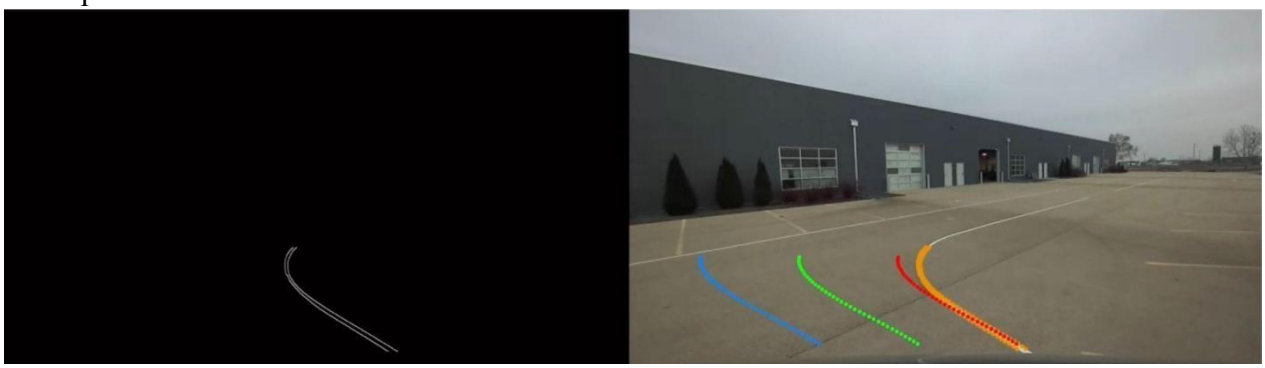
(Fig 2)

In Fig 3, With trick of the step 9, when one of the lanes is completely missing. The lane detector will make a horizontal shift to generate an undetected lane.



(Fig 3)

In Fig 4, Instead of fitting the linear equation in method one, the proposed method fits the cubic plane curve. Therefore the detector could detect curves.

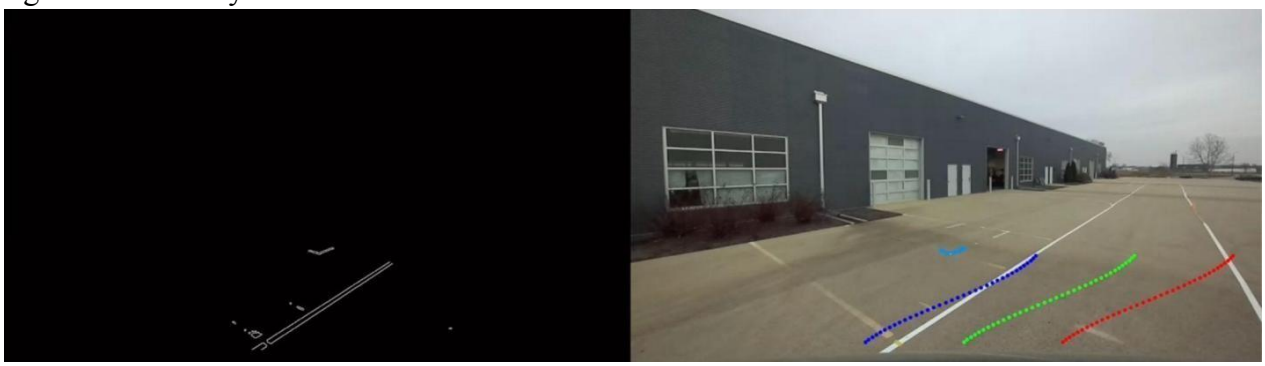


(Fig 4)

In Fig 5, when both lanes are not complete in the preprocessed image, the lane detector failed. Because it can't identify complete left and right points to fit curves.

(Fig 5)

In Fig 6, When the noise is too much, the DCSAN cluster algorithm will miss up and treat the noise as the lane. In the right image below, a tiny noise is detected as the left lane and the right lane is totally missed.



(Fig 6)

Conclusion:

The proposed method works well when the lane is complete and there are no other noises. The proposed image determines whether the detector successfully detects lanes. But the real situation is changing, we can't set a fixed parameter like threshold of white color, region of interest for all situations. The better way is to utilize Neural Network as a preprocess method and use the traditional way as the proposed method to generate lanes.