IMPLEMENTATION OF PATH TRACKING ALGORITHMS AND TRAJECTORY OPTIMIZATION BASED ON THE EXTENDED KALMAN FILTER

By

JIN-HYUK YU

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

2021

To my family and friends

ACKNOWLEDGMENTS

First, I would like to thank my advisor Dr. Carl Crane III for bringing me on Indy Autonomous Challenge Project at Center for Intelligent Machines and Robotics (CIMAR). It was a great experience for me to enlarge my sight in Autonomous Vehicle research are. And I also appreciate for guiding me through my research and studies. I also thank Dr. Michael Griffis, my thesis committee member, for his help and advice. Thanks to his help, I could successfully write my thesis well.

I also thank my colleagues and friends at CIMAR. As working on the Indy Autonomous Project together, they gave their first steps and guidance on my research journey.

I also thank my family. Thanks to my family's warm concern and support, I could end my research well.

TABLE OF CONTENTS

LIST OF FIGURES

Abstract of Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Science

IMPLEMENTATION OF PATH TRACKING ALGORITHMS AND TRAJECTORY OPTIMIZATION BASED ON THE EXTENDED KALMAN FILTER

By

Jin-hyuk Yu

MAY 2021

Chair: Carl D. Crane, III
Major: Mechanical Engineering

Autonomous vehicles, also known as Unmanned Ground Vehicles, is a state-of-the-art research trend in Mechanical, Computer, and Automobile Engineering research. After the concept of Autonomous vehicles was first introduced in 1939 Futurama exhibition, many researchers have been researching to establish fully autonomous driving vehicles.

Path tracking algorithm is one of the essential parts of autonomous driving. It allows the vehicle to track the trajectory generated by localization and navigation, and also drive without human interaction in a road environment. In this research, four popular algorithms for the path tracking were introduced: pure pursuit, lateral Stanley, PID feedback, and Model Predictive Control. Each algorithm has their own advantages, and many researchers have been utilizing in autonomous driving are. However, it is hard to remain robust in the complicated road environment only with one path tracking algorithm due to continuously changing environments such as obstacles, other vehicles, and humans. And this problem caused vehicles' collision, failing to track the path, etc. Finding the alternative resolution to solve this problem becomes a hot issue for many researchers.

In this research, we are going to implement four path tracking algorithms introduced above. To establish better performance in trajectory tracking, we are going to apply the extended Kalman filter, a common filtering method for positioning. Then by comparing results of four path tracking algorithms, we are going to seek how the performance is increased and what path tracking algorithm shows better performance and robustness in trajectory tracking.

CHAPTER 1
INTRODUCTION

1.1 History of Autonomous Vehicles

The concept of the autonomous vehicle first came out Futurama, an exhibit at the 1939 New York World’s Fair. In General Motors’ 1939 exhibition, the first self-driving vehicle was created by Norman Bel Geddes. It was an electric self-driving vehicle guided by radio-controlled electromagnetic fields generated with magnetized metal spikes embedded in the roadway. By 1958, General Motors actually had made this concept a reality. Sensors called ‘pick-up coils’ that could detect the current flowing through a wire embedded in the road were embedded on the car’s front end. To tell the vehicle to move the steering wheel left or right, the current could be manipulated.

Since then, many countries and companies have been trying to create a fully autonomous vehicle. GM, Google, Tesla and other companies developed their own autonomous vehicles to establish fully self-driving cars. With their endeavors, it is likely to expect that fully self-driving vehicles would come out in the near future.

At present, many autonomous vehicles developed and manufactured are considered to be semi-autonomous vehicles due to safety features, such as assisted parking and braking systems, and a few have the capability to drive, steer, brake, and park themselves. Autonomous vehicle technology relies on several parts: sensors, IPS, GPS, and automobile software. Sensors, such as visual sensors, Radar and Lidar, detect lane boundaries, signs and signals, obstacles, and environments surrounding. IPS and GPS help the vehicle to find its position and location on the road. Information from these two parts goes to automobile software, which is the center for controlling vehicle movements. These parts are considered as the necessary parts that all self-driving vehicles should contain, and many companies invest their time and money on developing these parts' technologies for better autonomous driving.

1.2 Path Tracking Controller

In the area of automobile software, path tracking controller is the essential part that autonomous vehicle should have. Path tracking controller is the bridge to connect the navigation to the execution mechanism of the autonomous vehicle. The goal of the path tracking algorithm is to generate actuator commands that minimize the cross-track error or simply the lateral distance between the vehicle's center point and the geometric path estimated by the motion planner. Apart from this, the controller should minimize the difference between the vehicle's orientation and path's heading while also limiting the rate of steering input for overall lateral stability and smooth driving experience. Since the advent of autonomous vehicles, path tracking controllers have existed. And recent developments have made them very extensive and robust system assumptions.

Since path tracking controller is essential for fully autonomous driving, many researchers have been trying to devise proper algorithms for the control of path tracking. There are several representative path tracking controllers used for autonomous driving: Pure pursuit controller, PID controller, Model Predictive controller, and Lateral Stanley controller. Each controller is considered to be good methods for autonomous driving control, and has been used for the development of better control for autonomous vehicles.

But they also have their own limits on tracking the trajectory for driving. This is because the driving environment has lots of variables that can make an effect on finding and tracking the optimal trajectory for autonomous vehicles. Especially, in complicated and curved roads, it is hard for autopilots to create the optimal trajectory and track it with using only a single path tracking controller because the driving environment changes continuously, and self-driving cars should detect these changes, find the optimal path and track the planned optimal path robustly. This problem causes many car accidents on the road, and it makes people realize the need for the enhancement of existed path tracking controllers.

To solve this problem, many researchers utilized various methods to improve the performance and robustness of path tracking algorithms. Some tries to improve the safety of controlling by extending the existed path tracking algorithms or combining more than two path tracking controllers. [Chen et al., 2018] Yuangpeng combined the pure pursuit controller with PID controller and low-pass filter so that the model based on these combinations can promise a better tracking performance.

Other researchers tried to solve the problem by adjusting other mechanisms like probabilistic models. [Kim et al., 2019] In Dae Jung Kim’s research, his resolution targeted the enhancement of robustness in steering control for vehicles by applying probabilistic models, such as Kalman Filter, Particle Filter, and Bayesian probability to model the trajectory, localization, and minimizing the tracking error.

There are many other approaches on solving the path tracking problem in complicated driving environments. But the point of all these approaches goes to the same point: minimize the path tracking error and maximize the accuracy and robustness of autonomous driving.

1.3 Research Goal

The goal of this research is to search for the way to improve the performance of the path tracking algorithm by applying probability filtering methods that can reduce the tracking error. Popular path tracking algorithms will be implemented following the driving simulation on a closed, complicated and arbitrary closed track first. Path tracking algorithms that are going to be used in this research are: Pure pursuit controller, Proportional-integral-derivative controller (PID controller), and Model predictive control (MPC). This is to compare the performance of each path tracking algorithms and to check the results of adjusting the Kalman Filter at the process of path tracking algorithms.

CHAPTER 2
BACKGROUND

In this section, we are going to review literatures that have been studied for autonomous vehicles. Ackermann steering mechanism that composes the vehicle model of this research will be reviewed. Path tracking algorithms that determine speed and steering settings at each instant of time in order for vehicles to follow a certain path will be reviewed. And Kalman filter applied for the trajectory optimization will also be review.

2.1 Ackermann Steering Based Vehicle Model

The main point of the autonomous vehicle driving in the driving simulation is to design the steering mechanism that we want to analyze. Steering is the main component of the vehicle that can have an effect on the result of driving. There are various kinds of modeling the steering mechanism for the driving simulation. In this chapter, we are going to look at Ackermann Steering model, the most common steering model for autonomous driving simulation.

Ackermann Steering model is the most commonly used and simplified Ackermann Steering Vehicle model used for geometric path tracking. It is a front-wheel-steering model (FWS) based on Kinematic analysis. [Thomas, 1992] The Ackermann Steering model is described at Figure 2-1.

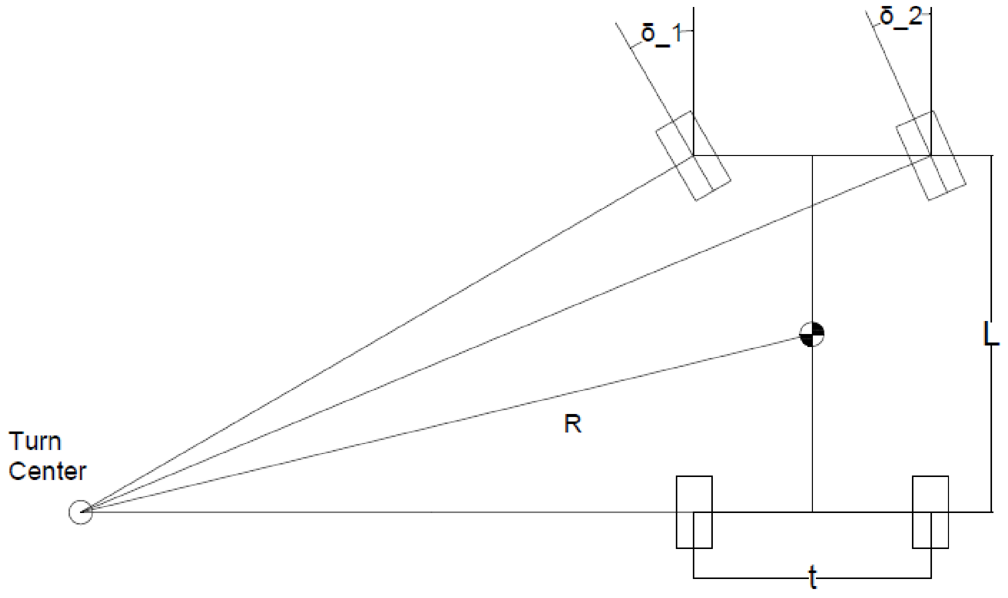

Figure 2-1. Simple Ackermann Steering Model

In Figure 2-1, the ideal geometry conditions for an automobile to turn right when the four wheels is shown. Ideally, there is a kinematic condition between the inner and outer wheels that allows wheels to turn slip-free at very low speed. The kinematic condition is called the Ackerman condition and is expressed by

$$\cot\delta_2 - \cot\delta_1 = \frac{t}{L} \tag{2-1}$$

where δ_1 and δ_2 are the steer angles of the wheel number 1 of the front left and the wheel number 2 of the front right wheel. [Jazar, 2019]

At Equation 2-1, the steering angles δ_1 and δ_2 can be written as

$$\begin{cases} \tan\ \delta_1 = \dfrac{L}{R_1 - \dfrac{t}{2}} \\ \tan\ \delta_2 = \dfrac{L}{R_1 + \dfrac{t}{2}} \end{cases} \tag{2-2}$$

This FWD Ackermann Steering vehicle model can be also simplified into Geometric Bicycle model. [Vu, 2012] For the center steer angle, δ, it can be written as

$$\delta = \tan^{-1}(\frac{L}{R_1}) \tag{2-3}$$

2.2 Path Tracking Algorithm

Path tracking algorithm is the main source of autonomous driving system that enables Unmanned Ground Vehicles (UGV) to automatically follow the track without any human interactions. This algorithm has been researched for the autonomous system in Geometric, Dynamic and Kinematic analysis. These analyses make various ways of path tracking algorithms for autonomous driving.

Geometric path tracker is one of the most popular classes of path tracking methods found in robotics. Geometric path tracking algorithm uses geometric relationships between the vehicle and the path resulting in control law solutions to the path tracking problems. [Snider, 2009]. In this paper, we will review four popular geometric path tracking controllers and a filtering algorithm.

2.2.1 Pure Pursuit Algorithm

Pure pursuit algorithm is the traditional way of path tracking methods for autonomous driving. The pure pursuit algorithm was originally developed as a method to calculate the arc necessary to get a robot back onto a path. The Terragator, six-wheeled and skid-steered robot that was used for outdoor vision experimentation in the early 1980s, was the first application of this method. [Wallace et al., 1985]

The basic procedure is depicted well in Craig’s research. [Craig, 1992] In his research, this tracking algorithm works by calculating the curvature that will move a vehicle from its source to a goal position. The whole point of the algorithm is to choose some goal position, where the goal is chosen as some point along the path to be tracked. The algorithm itself iterates continuously, with the goal point sliding along the path, forming a smooth tracking trajectory. Consider Figure 2-2. We can see the Geometry of the Pure pursuit algorithm showing the path to be tracker(left) and the calculated steering curvature(right). It operates in a fashion similar to the way humans drive, in which we fixate on a point some distance ahead on the road and attempt to follow it. [Giesbrecht, 2004]

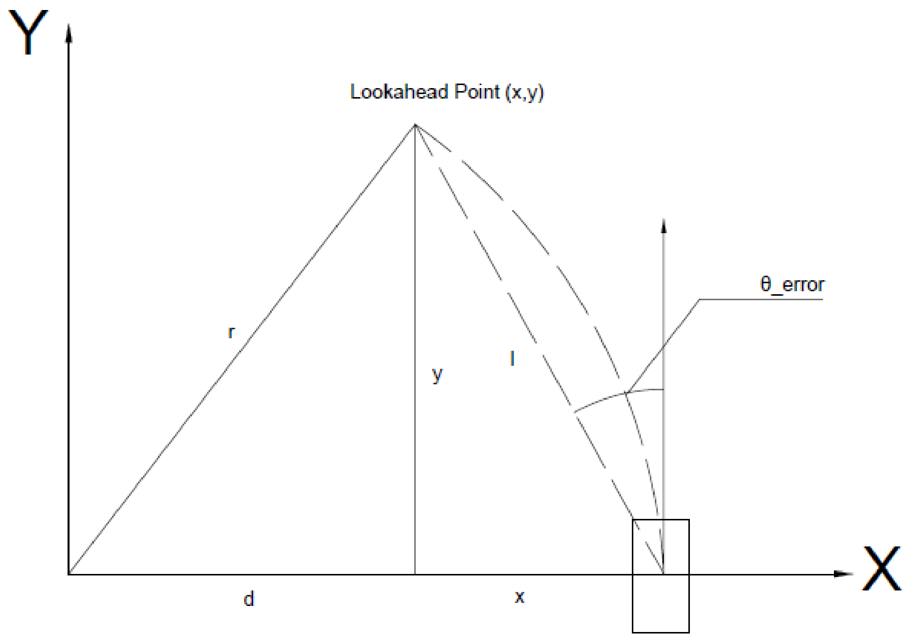

Figure 2-2. Pure Pursuit Geometry Model

To find the formula that works for Pure pursuit algorithm, let the point (x,y) be one lookahead distance 1 from the origin. The point (x,y) is constrained to be on the path. The objective of this pure pursuit algorithm is to find the curvature of the arc that joins the origin to (x,y) and whose chord length is 1. The lookahead distance is consider to be analogous to the distance to a spot in front of a car that a human driver might look toward to track the roadway.

The following two equations can be derived from the Figure 2-2. The first equation derives from the geometry of the smaller right triangle.

$$x^2 + y^2 = l^2 \tag{2-4}$$

The second derives from the summing of line segments on the x axis.

$$x + d = r \tag{2-5}$$

Equation 2-4 shows the circle of radius l about the origin. This is the locus of possible goal points for the vehicle. Equation 2-5 shows the relationship between the radius of the arc that joins the origin and the goal point, and the x offset of the goal point from the vehicle. Equation 2-4 also states that the radius of the arc and the x offset are independent and differ by d.

The next step is to find the relation between the curvature of the arc and the lookahead distance. From the Figure 2-2, we have

$$d^2 + y^2 = r^2 \tag{2-6}$$

Change Equation 2-5 into

$$d = r - x$$

And inserting this equation into Equations 2-6 results in

$$(r - x)^2 + y^2 = r^2$$

And it can be written as

$$x^2 + y^2 = 2rx \tag{2-7}$$

Using this result, we can find the curvature of an arc 'γ', which is given as

$$\gamma = \frac{1}{r} \tag{2-8}$$

Substituting Equation 2-7 into 2-4 yields

$$2rx = l^2$$
$$r = \frac{l^2}{2x}$$

The curvature of an arc can be written as

$$\gamma = \frac{2x}{l^2} \tag{2-9}$$

This formula has been related to the x offset of the goal point from the origin by the inverse square of the lookahead distance. And Pure pursuit algorithm is a proportional controller which operates on the error(θ_{err}) between the current vehicle heading and the heading to the goal point on the path.[Giesbrecht et al., 2004] It is shown in Figure 2-2.

From Figure 2-2, we can see that the error can be written as

$$\sin(\theta_{err}) = \frac{x}{l}$$

So, for small heading errors, $\theta_{err} \cong \frac{x}{l}$, and the curvature can be written as

$$\gamma = \frac{2\theta_{err}}{l} \tag{2-10}$$

From this formula, we can see that the only variance in the curvature is the lookahead distance, l. This makes the algorithm easier to implement and tune. Having smaller lookahead distance has some advantages [Andersen et al., 2016] It makes the pure pursuit algorithm be robust to disturbances and large lateral error. The input to the algorithm is also way points, rather than smooth curves, and is therefore less susceptible to discretization related issues. However, small lookahead distance also cause the corner cutting and steady state error problems, especially when the inappropriate lookahead distance is selected. Also, the smaller the distance is, the more aggressively the vehicle returns to the path, which is not smother. Therefore, Pure pursuit algorithm needs to regulate the lookahead distance appropriately for the robustness of the vehicle’s autonomous driving.

2.2.2 Geometry Lateral Stanley Control Algorithm

Geometry Lateral Stanley control algorithm, also known as Stanley controller is another popular path tracking algorithm for the autonomous driving. It is first used by Stanford University’s DARPA Grand Challenge Team at 2005 [Thrun, S. et al., 2006]. The Stanley method computes the steering command based on a non-linear control law that considers the cross-track error of the vehicle to the path, as measured from the front axle of the vehicle, as well as the heading error of the vehicle with respect to the path. The algorithm has been proven to exponentially converges to zero cross track error. [Andersen et al., 2016]

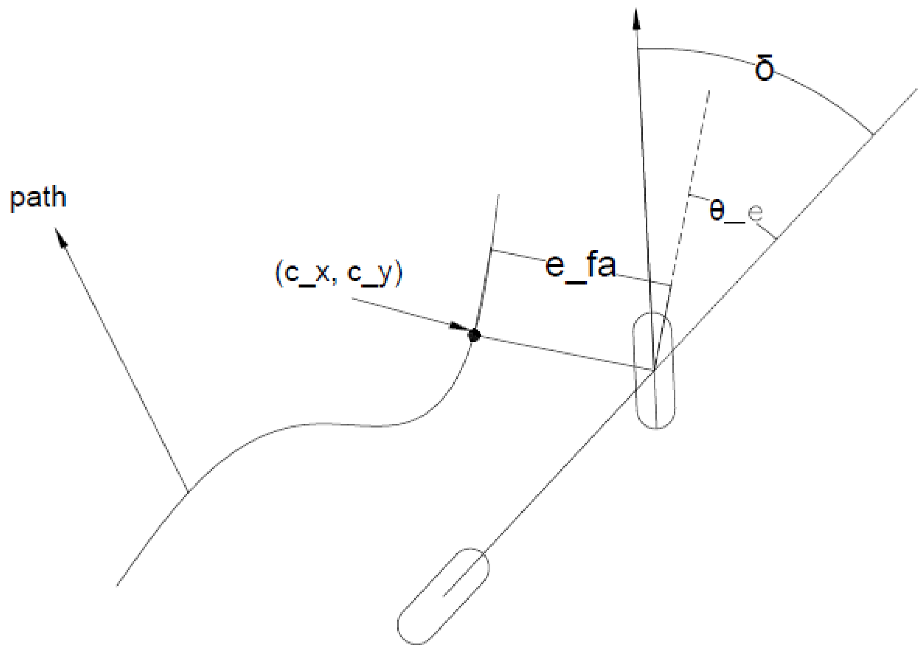

Figure 2-3. Lateral Stanley Algorithm Vehicle Model

Let’s define an intuitive steering law to accomplish three goals: to correct heading error, position error, and obey max steering angle bounds. Stanley method geometry can be seen at Figure 2-3.

At Figure 2-3, co-locating the point of control with the steered front wheels allows for an intuitive control law, where the first term simply keeps the wheels aligned with the given path by setting the steering angle δ equal to the heading error θ_e. It can be written as

$$\theta_e = \theta - \theta_p \tag{2-11}$$

where θ is the heading of the vehicle and θ_p is the heading at the nearest path point. To eliminate the cross-track error from the steer, there are several things to do.

- Essentially proportional to error
- Inversely proportional to speed
- Limit effect for large errors with inverse tangent
- Gain k determined experimentally

Then we can gain the formula about the steering angle δ as:

$$\delta(t) = \tan^{-1}(\frac{ke(t)}{v_f(t)}) \tag{2-12}$$

where k is a gain parameter, $v_f(t)$ is the longitude velocity and $e(t)$ is the cross-track error. Considering the heading error θ_e, when the cross-track error is non zero, the resulting steering control law is given as

$$\delta(t) = \theta_e(t) + \tan^{-1}(\frac{ke(t)}{v_f(t)}) \tag{2-13}$$

And the maximum and minimum steering angles are written as

$$\delta(t) \in [\delta_{min}, \delta_{max}]$$

It is clear that the desired effect is achieved under this steering angle limit.

As the cross-track error increases, the wheels are steered further towards the path. It means that the larger the heading error is, the larger the steering will be corrected. It can be written as

$$\tan^{-1}(\frac{ke(t)}{v_f(t)}) \approx \frac{\pi}{2}$$

And the steering angle becomes

$$\delta(t) = \theta_e(t) + \frac{\pi}{2} \tag{2-14}$$

We can see at Figure 2-3 that as heading changes due to steering angle, the heading correction counteracts the cross-track correction, and drives the steering angle back to zero. And the vehicle approaches the path, the cross-track error drops, and steering command starts to correct heading alignment.

When the steering error does not exceed its maximum steering angle range, the derivative of the cross-track error is

$$\dot{e}(t) = \frac{-ke(t)}{\sqrt{1 + (\frac{ke(t)}{v_f})^2}} \tag{2-15}$$

For small cross track errors, Equation 2-15 leads to exponential decay characteristics

$$\dot{e}(t) \approx -ke(t).$$

2.2.3 PID Feedback Control

Proportional-Integral-Derivative feedback control (PID feedback control) is one of the most common controllers used in industrial control applications. PID controller is feasible and easy to be implemented in many engineering research areas. In this research, PID controller was used to control the steering for path tracking algorithm.

The proportional (P), integral (I) and derivative (D) controllers work in tandem to give rise to a much more efficient PID controller. [Franklin et al., 1994] The PID controller takes advantage of all the three primary controllers to generate a sophisticated control action that proportionally corrects the error, then dampens the resulting overshoots and reduces any steady-state error over the time.

In continuous time, the PID controller is represented as follows.

$$u(t) = k_p * e(t) + k_I * \int_{t_0}^{t} e(t)dt + k_D * \frac{d}{dt} e(t) \tag{2-16}$$

In discrete time, the above equation takes the following form.

$$u_{t+1} = k_p * e(t) + k_I * \sum_{i=t_0}^{t} e_i + k_D * \left[\frac{e_t - e_{t-1}}{\Delta t}\right] \tag{2-17}$$

As stated, earlier, $t - t_0$ represents the temporal size of history buffer over which the error is integrated.

Generally, P controller, I controller and D controller have their own characteristics. The P controller can be thought as correcting the present error by generating a control action proportional to it. The I controller can be thought of as correcting any past error by generating a control action proportional to the error accumulated over time. Finally, the D controller can be thought of as correcting any future error by generating a control action proportional to the rate of change of error. By combining these features, PID control can show the right performance for many different systems in various areas.

On the base upon the system parameters, if they can be achieved or estimated precisely, PID gains can be designed. Moreover, the PID gain can be designed just based on the system tracking error and treats the system to be “black box” if the system parameters are unknown. [Prabir, 2017]

2.2.4 Model Predictive Control

Model Predictive Control (MPC) is the latest method of process control which is used to control a process while satisfying a set of constraints. The phenomenal success of MPC in the process industries is well described at Qin and Badgwell’s research [Qin & Badgwell, 2003]. MPC relies on dynamic models of the process, most often linear empirical models obtained by system identification. MPC use linear or non-linear motion models of the system to predict its future steps and determine the optimal control action by numerically solving a bounded optimization problem at each time step. MPC has been regarded as a good method for autonomous driving due to its conceptual simplicity and its ability to control complex systems easily and effectively with hard control constraints and many inputs and outputs. Figure 2-4 shows the diagram of MPC.

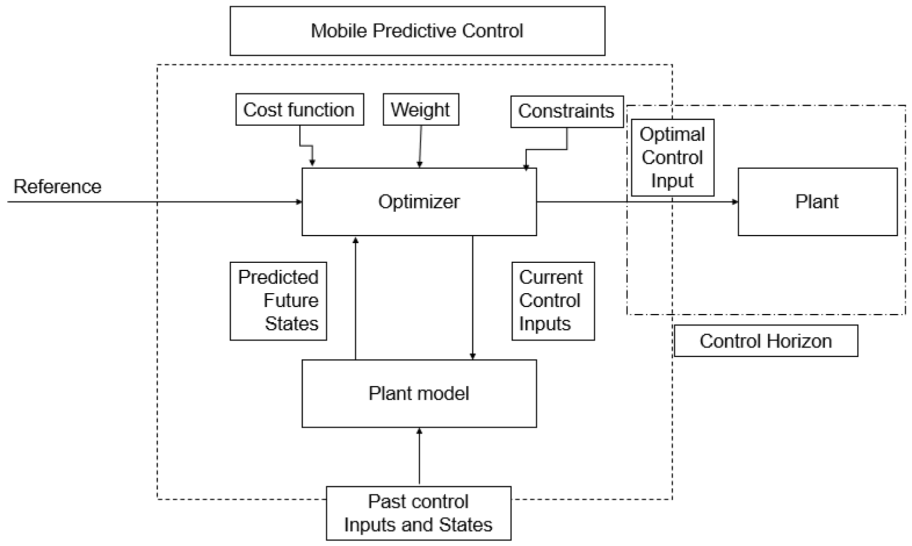

Figure 2-4. Model Predictive Control Diagram

Given the current state and the reference trajectory to follow, MPC involves simulating different control inputs without actually applying them to the system, predicting the resulting future states in form of a predicted trajectory using motion model up to a certain prediction horizon, selecting the optimal set of control inputs corresponding to minimal cost trajectory considering constraints at each step in time, and applying the very first set of optimal control inputs up to a certain control horizon to the ego vehicle, discarding the rest.

Parameters of MPC Design are sample time (T_s), prediction horizon (p), control horizon (m), constraints and weights. Each parameters are needed to be set at certain values to show the good result of MPC. [Samak et al., 2011]

- Sample time: $0.05 * t_r \leq T_s \leq 0.1 * t_r$
- Prediction Horizon: $\frac{t_s}{T_s} \leq p \ \leq 1.5 * \frac{t_s}{T_s}$
- Control horizon: $0.1 * p \ \leq m \ \leq 0.2 * p$
- Constraints: There are two kinds of constraints. Hard constraints, which cannot be violated under any circumstances, are recommended for control inputs thereby accounting for actuation limits. Soft constraints, which can be violated with minimum necessary amount, are recommended for outputs and time derivatives of control inputs that may be occasionally violated.

In the process of MPC, the cost function is a step of evaluating and minimizing the cost within the constraints of the task system. [Wang et al., 2018] The exact choice of cost function depends very much upon the specific problem statement requirements, and is up to the control engineer. The cost function can be written as:

$$J = \sum_{i=1}^{p} w_e {e_{k+i}}^2 + \sum_{i=0}^{p-1} w_{\Delta u} \Delta {u_{k+i}}^2 \tag{2-17}$$

- $p\text{: }prediction\ horizon$

- $w: waypoint$
- $e: predcited\ error$
- $\Delta u: steering\ wheel\ angle\ increment$

With all processes written above, the optimization process is proceeded. The optimization is to choose the optimal set of control inputs corresponding to the predicted trajectory with the lowest cost before optimized inputs go to the plant.

MPC is regarded as a good method for the path tracking algorithm for many years. However, MPC is not an absolute solution for path tracking and has some limits. [Lundström et al., 1995] Limits include:

- Limited model choices
- Sub-optimal feedback
- Lack of nominal stability
- Sub-optimal or inefficient solution of the dynamic optimization

Due to these limitations, MPC has been researched to compensate its limits, and various kinds of extended MPC, such as adaptive MPC, robust MPC, and nonlinear MPC are developed.

2.3 Kalman Filter

Kalman filter uses a series of measurements observed over time, including statistical noise and other inaccuracies. Also, it can produce estimates of unknown variables which tend to be more accurate than those based on a single measurement alone, by estimating a joint probability distribution over the variables for each timeframe. It is also known as linear quadratic estimation (LQE).

Kalman filter is good for continuously changing systems. They have the advantage that they are light on memory, and fast processing, making them well suited for real time problems and embedded system. [Thrun, S. et al, 2005] Due to this advantage, Kalman filter has been used for many areas including robotics, aerospace, manufacturing, autonomous engineering, etc. Kalman filter is also famous for it was used in the Apollo project.

In Kalman filter, the distribution is given by Gaussian, which is a continuous function over a space of locations. The estimation part is being done by two steps. [Paul et al., 2015] The algorithm first predicts the state and then perform measurement update. These two steps are called ‘Measurement Update’ and ‘Motion Update’.

Kalman filter must have the model of the world described as a set of differential equations. The Kalman filter model first assumes the matrix or true state-space form as

$$\dot{x} = Fx + Gu + w \tag{2-18}$$

where x is a column vector with the states of the system, F the system dynamics matrix, u is a known vector, which is sometimes called the control vector, and w is a white-noise process, which is also expressed as a vector. There is a process-noise matrix Q that is related to the process-noise vector according to

$$Q = E[ww^T] \tag{2-19}$$

The Kalman filter formulation requires that the measurement be linearly related to the states according to

$$z = Hx + v \tag{2-20}$$

where z is the measurement vector, H is the measurement matrix, and v is white measurement noise, which is also expressed as a vector.

Generally, Kalman filter can be applied to only linear Gaussian models for non-linearities and unmodeled dynamics can seriously decrease the filter performance, even when it was planned to perform with unknown stochastic signals as inputs. This is because the effect of unmodeled dynamics depends on the input, and, therefore, can bring the estimation algorithm to instability. On the other hand, independent white noise signals will not let the algorithm diverge. Extended Kalman filter, a nonlinear version of the Kalman filter, came out later to compensate for this problem. [Cuenca et al., 2019]

CHAPTER 3
EXPERIMENT PREPARATION

Before we implement pure pursuit algorithm and PID feedback control, we need to do pre-processes: design the vehicle model, create the obstacle avoidance algorithm and plan the referenced path for the autonomous vehicle. After the optimal reference is set, the autonomous vehicle is ready to drive with utilizing pure pursuit algorithm and PID feedback control for steering. Figure 3-1 shows the progress of these whole process of autonomous driving.

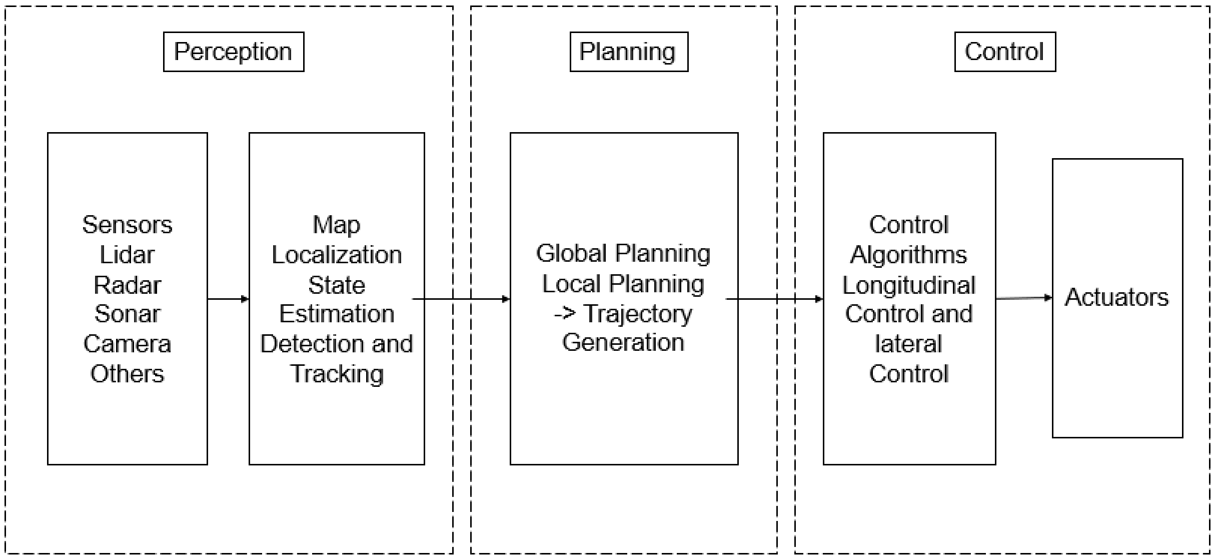
Perception
Planning
Control
Sensors Lidar Radar Sonar Camera Others
Map Localization State Estimation Detection and Tracking
Global Planning Local Planning -> Trajectory Generation
Control Algorithms Longitudinal Control and lateral Control
Actuators

Figure 3-1. Autonomous Driving System Architecture

In this chapter, we are going to look at how this research is performed based on Figure 3-1 and how experiment results will be compared.

3.1 Simulation World Setup

The research is based on the simulation world from MATLAB 2020b, a proprietary multi-paradigm programming language and numeric computing environment developed by MathWorks. Simulation world defines the simplified real-time world, kinematics, and dynamics of autonomous vehicle in this simulation world before applying path planning algorithm. The following steps are performed before the path planning algorithm is executed

3.2 Ackermann Steering Based Ego Vehicle Model

In this research, the vehicle model is based on Ackermann Steering algorithm. This algorithm is based on kinematic bicycle Ackermann steering model. This model can be seen at Figure 3-2.

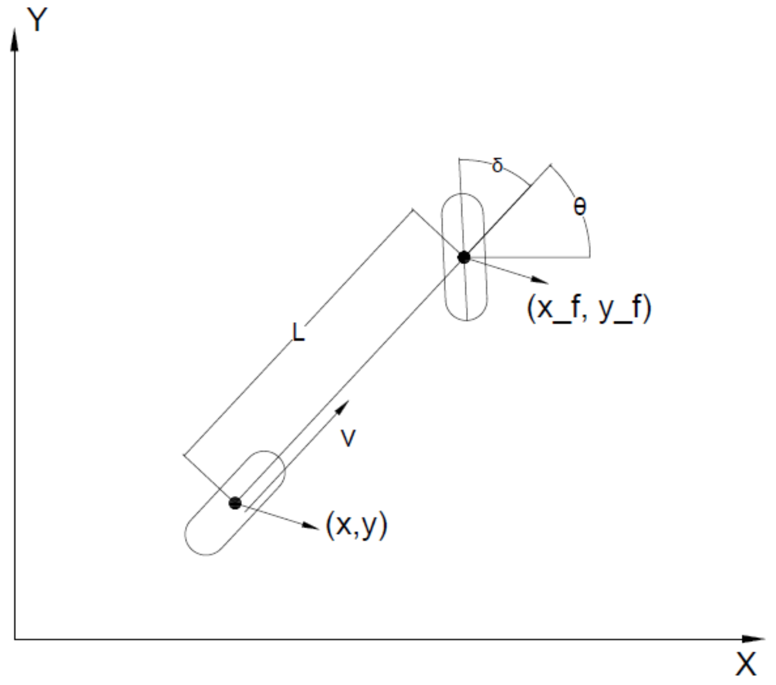

Figure 3-2. Kinematic Bicycle Ackermann Steering Model

The vehicle model for this research is a non-holonomic front wheel-based model. At here, the velocity and the rear-axle centers are constrained in the Ackermann Steering model. In order to get the vehicle back to the path at higher speeds, for lower speed, a low steering angle suffices whereas, it is essential to steer hard towards the path. This explains the importance of involving kinematics for vehicle modeling.

Chinmay Samak stated that the kinematic vehicle model can nearly-accurately represent a vehicle driving slowly and making smooth turns. [Samak et al., 2020] But if we consider a fast-driving car, such as a racing car, it is very likely to happen that the kinematic vehicle model would fail to capture the actual system dynamics. In this research, the vehicle model is considered as 'low-speed' driving model.

Let the ego vehicle state vector 'q', and it can be defined as follows.

$$q = [x, y, \theta, v]^T \tag{3-1}$$

Here, each variables of Equation 3-1 is defined as follows.

v_x be the longitudinal velocity

θ be the heading of the vehicle

(x, y) be the real axle position

Assume that the wheels have no lateral slip and only front wheel is steerable.

Intuitively from the Figure 3-2, we can write the expressions

$$R = \frac{v_x}{\dot{\theta}} \tag{3-2}$$

$$\frac{v_x \tan(\delta)}{L} = \frac{v_x}{R} \tag{3-3}$$

$$\tan(\delta) = \frac{L}{R} \tag{3-4}$$

Considering the bicycle model in the x-y plane with v_x as rear-wheel axle velocity or longitudinal velocity, and (x, y) its instantaneous position, we can write

$$\dot{x} = v_x \cos(\theta) \tag{3-5}$$

$$\dot{y} = v_x \sin(\theta) \tag{3-6}$$

$$\dot{\theta} = \frac{v \tan(\delta)}{L} \tag{3-7}$$

For the purpose of control, it is useful to write the kinematic model from Equation 3-5, 3-6, and 3-7 in the two-input driftless form

$$\dot{q} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \\ \dot{\delta} \end{bmatrix} = \begin{bmatrix} \cos(\theta) \\ \sin(\theta) \\ (\frac{\tan(\delta)}{L}) \\ 0 \end{bmatrix} v + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \dot{\delta} \tag{3-8}$$

where v and $\dot{\delta}$ are the longitudinal velocity and the angular velocity of the steered wheel respectively.

Ignoring the dynamic analysis for the vehicle model clearly cause negative effects on tracking performance as speed are increased and path curvatures changes. To run the simulation based on the continuous time, dynamic vehicle model is necessary. Due to the cost of computing, the simple linear dynamic bicycle model is applied. Detailed description about the dynamic bicycle model which is applied in this research is based on Jarrod M. Snider's research. [Snider, 2009]

3.3 Implementation of Path Tracking Controller

This research focuses on implementation of popular path tracking controllers used for autonomous driving. The path tracking controller will be utilized in two ways. First, Pure pursuit, MPC, and PID feedback controller will be coded and processed without any other additional method that can improve their properties. Second, extended Kalman filter will be added to each tracking algorithm to check how their performances are improved or otherwise, worse than the filter is applied.

3.3.1 Pure Pursuit Algorithm

The process of pure pursuit algorithm is following the steps below:

- Obtain the current position of the vehicle in the coordinate system
- Find the nearest point of the vehicle on the reference track

- Select the appropriate look-ahead distance and find the target point
- Convert the coordinates of the target point into the vehicle's coordinate system
- Calculate and obtain the steering angle
- Cycle after updating the vehicle position

In the vehicle's coordinate system, (x, y) is the pose of the goal point on the reference path, L_a , is the lookahead distance and R is the desired turning radius. The desired steering angle of the vehicle can be determined using the angle α, the angle between the heading angle of the vehicle and the lookahead vector. The pure pursuit geometry which is applied is shown in Figure 3-3

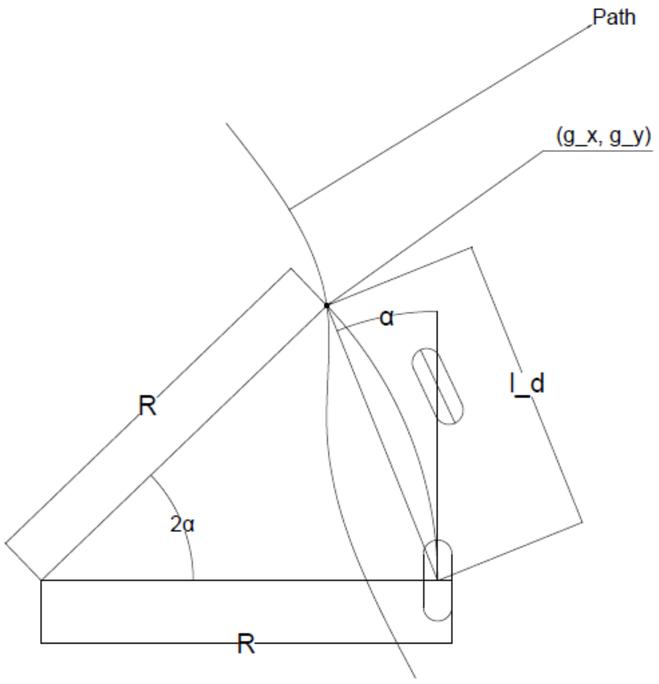

Figure 3-3. Pure Pursuit Geometry with The Reference Path

The equations for the angle α can be written as below:

$$\frac{L_d}{\sin(2\alpha)} = \frac{R}{\sin(\frac{\pi}{2} - \alpha)}$$
$$\frac{L_d}{2\sin(\alpha)\cos(\alpha)} = \frac{R}{\cos(\alpha)}$$
$$\frac{1}{R} = k = \frac{2\sin(\alpha)}{L_d}$$

where L_d is the lookahead distance, and k is the curvature of the circular arc. Applying an Ackermann model with two degrees of freedom, the steering angle is given as

$$\delta_{L_d}(t) = \tan^{-1}(\frac{2\mathrm{L}\sin(\alpha)}{L_d}) \tag{3-9}$$

where L is the vehicle’s wheelbase.

The pure pursuit is influenced by the lookahead distance, which is dynamically adjusted with the velocity. The characteristic of the pure pursuit is that a sufficient look-ahead distance will result in cutting corners while tracking a curved path. [Park et al., 2015]

3.3.2 PID Feedback Control

PID controller is a very commonly used controller in a wide variety of industrial process. PID controller operates based on an error value which is the difference between a measured process feedback and a desired setpoint. At PID feedback control, the single loop PID feed control was adopted. The process of this loop is shown in the Figure 3-4.

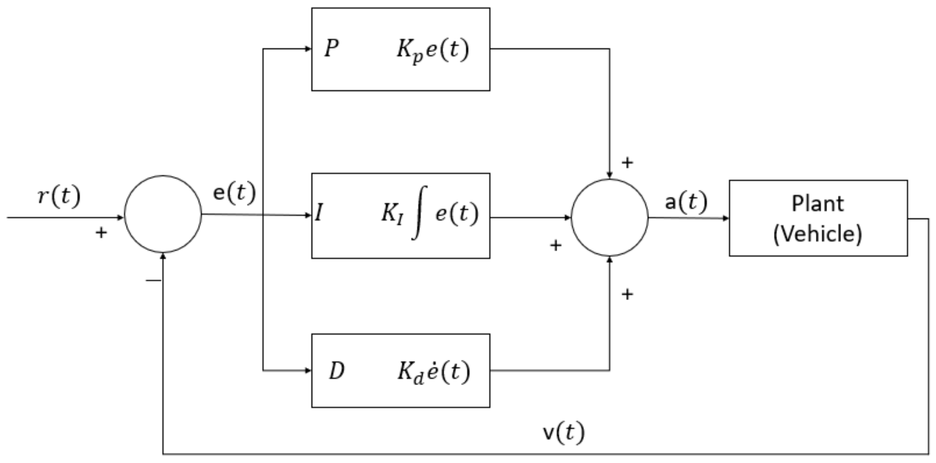

Figure 3-4. Illustration of PID Feedback

There are four kinds of controllers that combines more than two terms. The most basic PID feedback controller, P controller is the controller which uses only the proportional gain. In continuous time, the P controller is represented as follows:

$$u(t) = k_p * e(t) \tag{3-10}$$

PD controller can be thought of as a compound controller constituted of the proportional and derivative controllers. In continuous time, the PD controller is represented as follows:

$$u(t) = k_p * e(t) + k_D * \frac{d}{dt} e(t) \tag{3-11}$$

PI controller is a combination of proportional and integral controllers. In continuous time, the PI controller is represented as follows:

$$u(t) = k_p * e(t) + k_I * \int_{t_0}^{t} e(t)\, dt \tag{3-12}$$

PID controller consists of proportional, integral and derivative controllers. These three controllers work in tandem to give rise to a much more efficient PID controller. In continuous time, the PID controller is represented as follows:

$$u(t) = k_p * e(t) + k_I * \int_{t_0}^{t} e(t)\,dt + k_D * \frac{d}{dt} e(t) \qquad (3\text{-}13)$$

e: the error signal

k_p: the proportional gain

k_I: the integral gain

k_D: the derivative gain

When PID controller is adjusted, in Mohamad's research, he depicted how to calculate the steering angle. [Abed et al. 2020] The steering angle we need to find can be calculated by the equation below.

$$\delta(t) = -(k_p * e(t) + k_I * \int_{t_0}^{t} e(t)\,dt + k_D * \frac{d}{dt} e(t)) \qquad (3\text{-}14)$$

3.3.3 PD Feedback Control Application

In this research, PD feedback control was chosen for the implementation and autonomous driving control. This is because unlike other controller methods, PD controller can improve transient performance. Also, this controller can sooth the system response and reduce any overshoots, which are critical for the stability of driving. This is necessary to autonomous driving because the improvement of transient performance can enhance the equilibrium of the system, resulting in a safe driving.

Since the derivative controller, D, is extremely weak to noise, the tune of the derivative controller is essential. In this research, the value of the derivative gain is set to be quite bigger than the proportional gain and the feedforward curvature parameter was added in the steering angle calculation to compensate this problem. [Samak et al., 2011] The related equation is shown below.

$$\delta(t) = -\left(k_p * e(t) + +k_D * \frac{d}{dt}e(t)\right) + feedfoward\_curvature \tag{3-15}$$

For the formula of PD control, the error $e(t)$ is the vector error between the reference path data and the position data of the moving vehicle. The error consists of position data, (x, y, θ) with the lateral error data.

3.3.4 Model Predictive Control

In this research, Model Predictive Control (MPC) action is computed on-line rather than a pre-computed, off-line, control law.

The MPC uses, at each sampling instant, the plant's current input and output measurements, the plant's current state and the plant's model for two objects: calculate, over a finite horizon, a future consequence that optimizes a given performance index and satisfies constraints on the control action, and use the first control in the sequence as the plant's input.

In this research, the MPC is constructed based on the steps from Vavasis's textbook related to MPC. [Vavasis, 1998] The steps of MPC code are below:

- Insert parameters
- Convert (x, y) and yaw to the error dynamics
- For the finite horizon, update error dynamics
- Set up the time-varying error dynamics model
- Apply the delay compensation
- Convex the optimization
- Calculate the predicted trajectory

3.4 Extended Kalman Filter Application

This research focuses on how the performance of autonomous driving can be improved and how the trajectory tracking can be more robust and exact. The way how the extended Kalman filter is applied to the path tracking steps is shown in the Figure 3-5. The noise applied in the step of filtering is Gaussian noise.

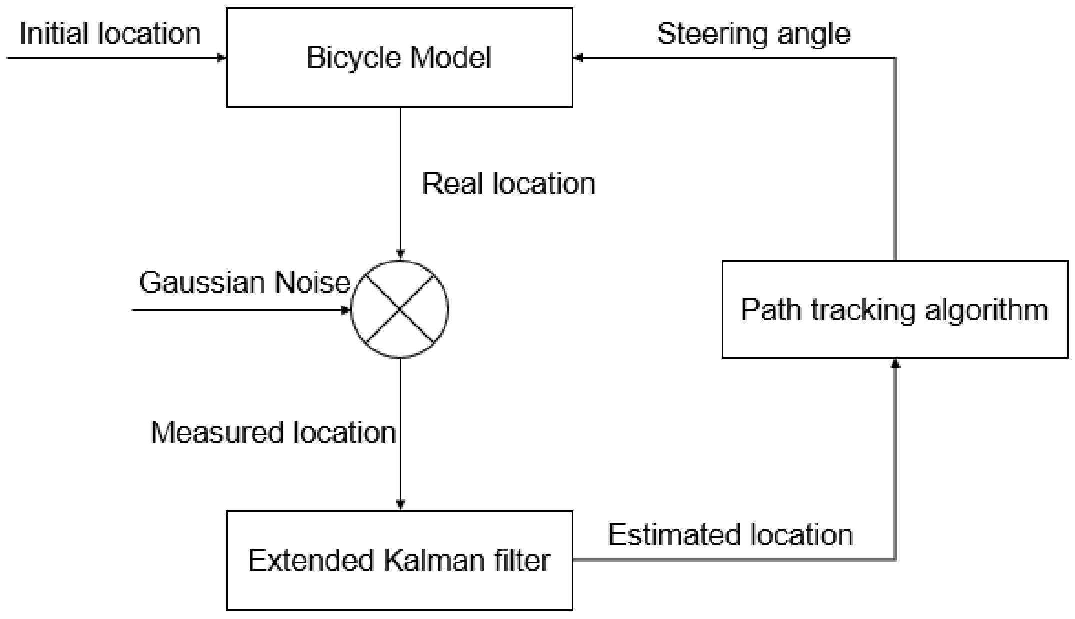

Figure 3-5. Extended Kalman Filter Application

Gaussian noise, also known as a white noise, is a type of noise which is in the form of Gaussian distribution. The noise is always happened in real driving environment in the process of finding the vehicle’s position via the normal GPS. In this research, to simulate the vehicle model as similar as the real vehicle drives in the road environment, gaussian noise is applied to see whether the extended Kalman filter works well or not.

In this research, the extended Kalman filter is applied. The state transition and observation models need not only be linear functions of the state, but also may instead be nonlinear functions. These functions are shown as Equation 3-16 and 3-17.

$$x_k = f(x_{k-1}, u_{k-1}) + w_{k-1} \tag{3-16}$$

$$z_k = h(x_k) + v_k \tag{3-17}$$

where f is the function of the previous state, x_{k-1}, and the control input, u_{k-1}, that provides the current state x_k. h is the measurement function that relates the current state, x_k, to the measurement z_k. w_{k-1} and v_k are Gaussian noises for the process model and the measurement model with covariance Q and R, respectively.

At here, the control input includes multiple input sets. These sets include the position data, (x, y) and the heading angle, yaw. The output includes the estimated position and the yaw angle.

The algorithm of the extended Kalman filter is mostly same with the basic Kalman filter. The one thing that we need to do is to obtain the Jacobian matrix, first-order partial derivative of a vector function with respect to a vector, of each model in each time step as:

$$F_{k-1} = \left.\frac{\partial f}{\partial x}\right|_{\widehat{x_{k-1}}^{+}, u_{k-1}} \tag{3-18}$$

$$H_k = \left.\frac{\partial h}{\partial x}\right|_{\widehat{x_k}^{-}} \tag{3-19}$$

By doing this, we can linearize the models about the current estimate. The filter algorithm is very similar to the basic Kalman filter. The specific description and the code applied for this research are included in Intech's research. [Y. Kim & Bang, 2019]

3.5 Experiment Standards and Conditions

The purpose of this research is to implement path tracking algorithms and find a way to improve their performances in trajectory tracking. The evaluation standards for this research are below:

- Steering angle evaluation
- The status of Latitude change
- The trajectory created by the path tracking algorithm
- The performance change after the extended Kalman Filter is applied

And the parameters of the vehicle model are shown below:

- Steering dynamics time constant: 0.27
- Wheelbase: 2.5m
- Width: 0.724m
- Steering limit: 30°
- Velocity range: -5 km/h ~ 10 km/h

For the pure pursuit, the lookahead distance is constant as 6.0 m.

CHAPTER 4
RESULT COMPARISON

4.1 Simulation Result without The Extended Kalman Filter

First experiment is based on the simulation without running the extended Kalman filter to estimate the position. The reference path for the vehicle model is shown in Figure 4-1. The red colored path is the reference path generated by the MATLAB code. At this code, I created the arbitrary path using the spline.

Then, the main simulation code for the experiment was exploit. In this program, we can choose what kinds of algorithms we can choose. At every algorithm, I add a noise error to conduct the simulation with fully conditions. The simulation is conducted under the condition without the extended Kalman filter. The velocity of the vehicle is constant to compare the performance between Pure pursuit algorithm, PID feedback control algorithm, and Model predictive algorithm.

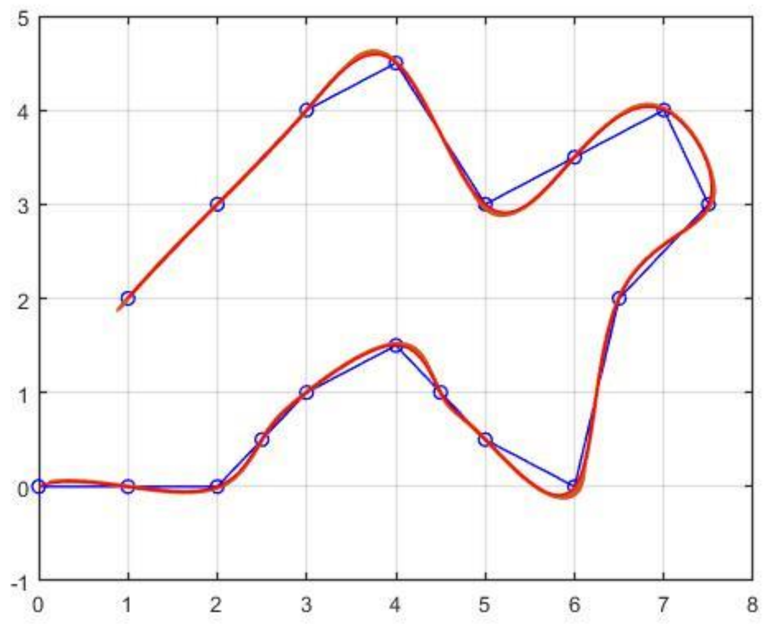

Figure 4-1. Reference Path Generated by PathDesign.m

The result of each algorithms is described below. The simulation was exploited with the video so that the vehicle model can be checked with the up-to-date data.

Each figure shows the result of path tracking algorithms. At Figure 4-2, the pure pursuit algorithm was conducted. The lookahead distance is 8 m, and this model drives the estimated trajectory during 35 seconds. We can check from the Figure 4-2 that the vehicle model driving under the pure pursuit algorithm drove the trajectory fully within 35 seconds. The result also shows that between 10~15 seconds and 25~30 seconds, the steering angle fluctuated more than any other parts, and the changes of the latitude error were also bigger than any other parts. Without those parts, the simulation result of the pure pursuit algorithm shows that the vehicle model tracked the trajectory well and was robust.

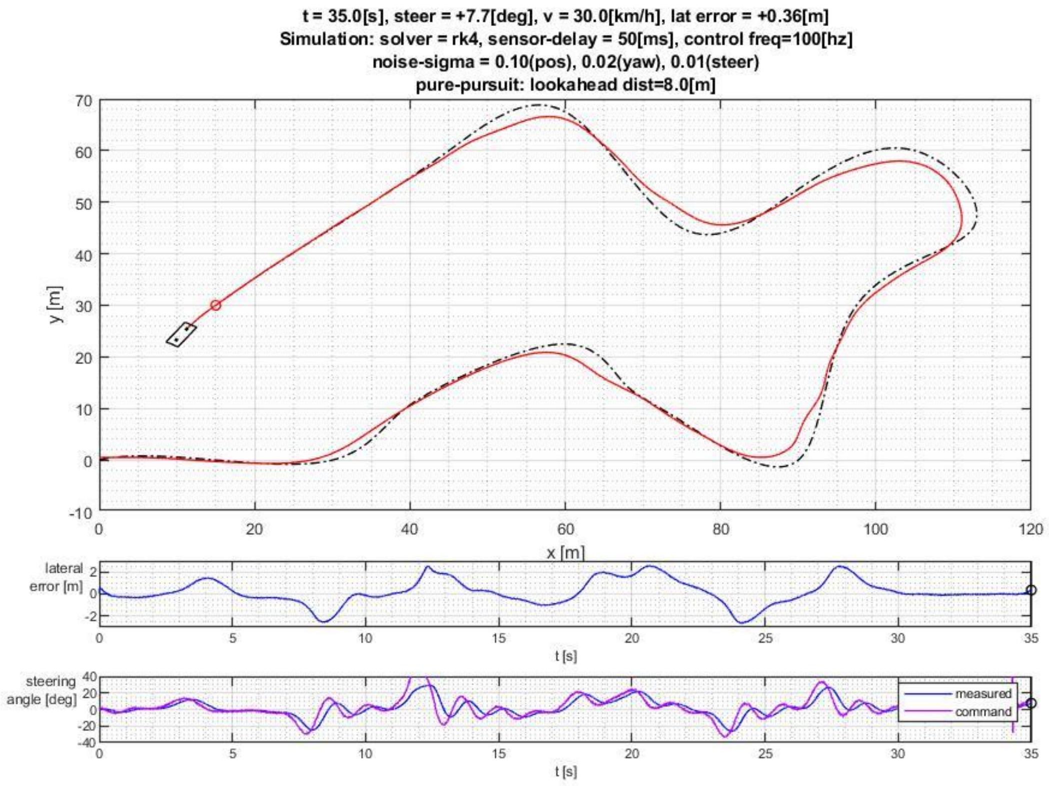

Figure 4-2. Simulation Result with Pure Pursuit Algorithm

Figure 4-3 is the result of PD feedback controller. At here, all parameters that are given and constant are the same with other path tracking algorithm. Compared to the result of the pure pursuit algorithm, the result of PD feedback control algorithm showed the unstable state during the driving. The vehicle model could not finish its path within the manual time. And unlike the result of the pure pursuit algorithm, the steering angle change rate during the simulation shows unstable states, and especially during the time from 10 to 15 seconds, it bounds a lot more than the steering angle limit. This result shows that PD control at this point, needs to be tuned manually to run the trajectory with minimum error.

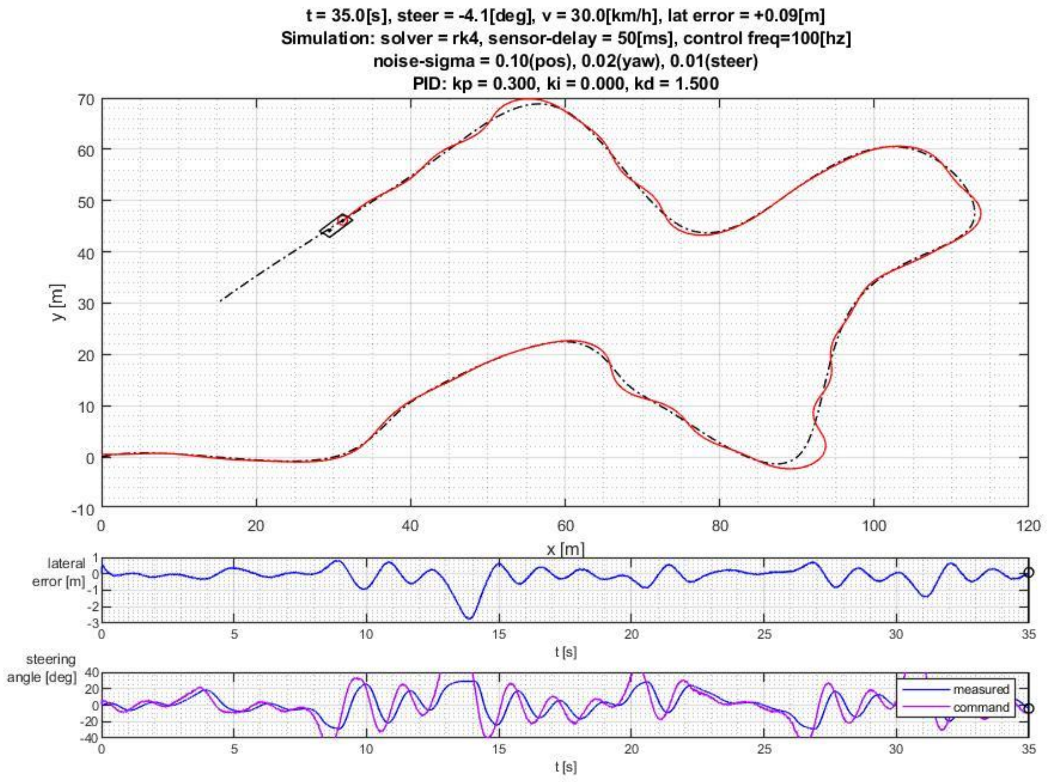

Figure 4-3. Simulation Result of PD Feedback Control

Figure 4-4 shows the simulation result of MPC. In Figure 4-4, the vehicle model tracked the trajectory very similar to the reference path. The result showed that the vehicle model with MPC drove the trajectory fully within 35 seconds. The steering angle change rate is really stable, and it bounds between the steering angle boundary. The latitude error shows the stable state and the value of it shows that MPC shows the robustness better than two control algorithms conducted above. Especially, in the time boundary between 10 and 15 seconds, unlike two other control algorithm showed unstable result, MPC shows much stable than other two control algorithm.

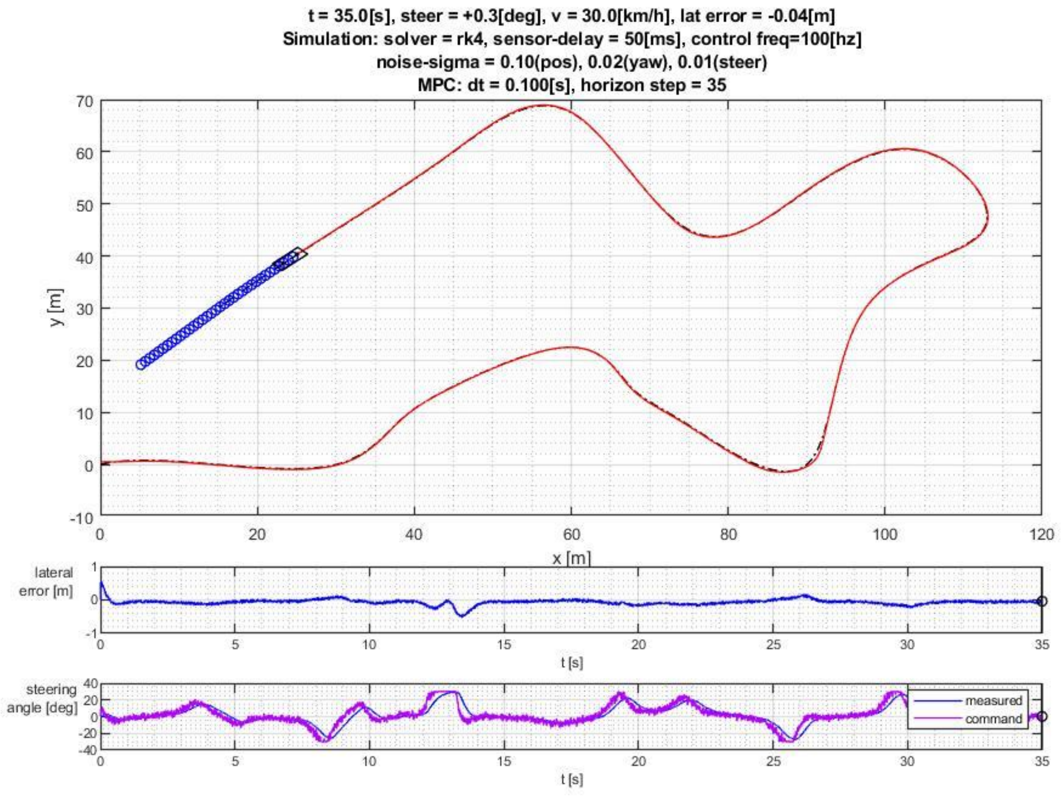

Figure 4-4. Simulation Result of MPC

4.2 Simulation Result with The Extended Kalman Filter

In this simulation, the extended Kalman filter was applied to get the estimated position data for the path tracking controller. In the extended Kalman filter, the standard noise for the estimation of the position is set manually to check the exact effect of the extended Kalman filter.

In Figure 4-5, the pure pursuit algorithm is simulated with the extended Kalman filter. Figure 4-5 shows that the trajectory tracking performance is a little increased, and unlike the simulation without the extended Kalman filter, in the range from 10 seconds to 15 seconds, the vehicle model shows better performance of trajectory tracking in the simulation. And the latitude error change rate is decreased. However, the graph shows the increase of the instability.

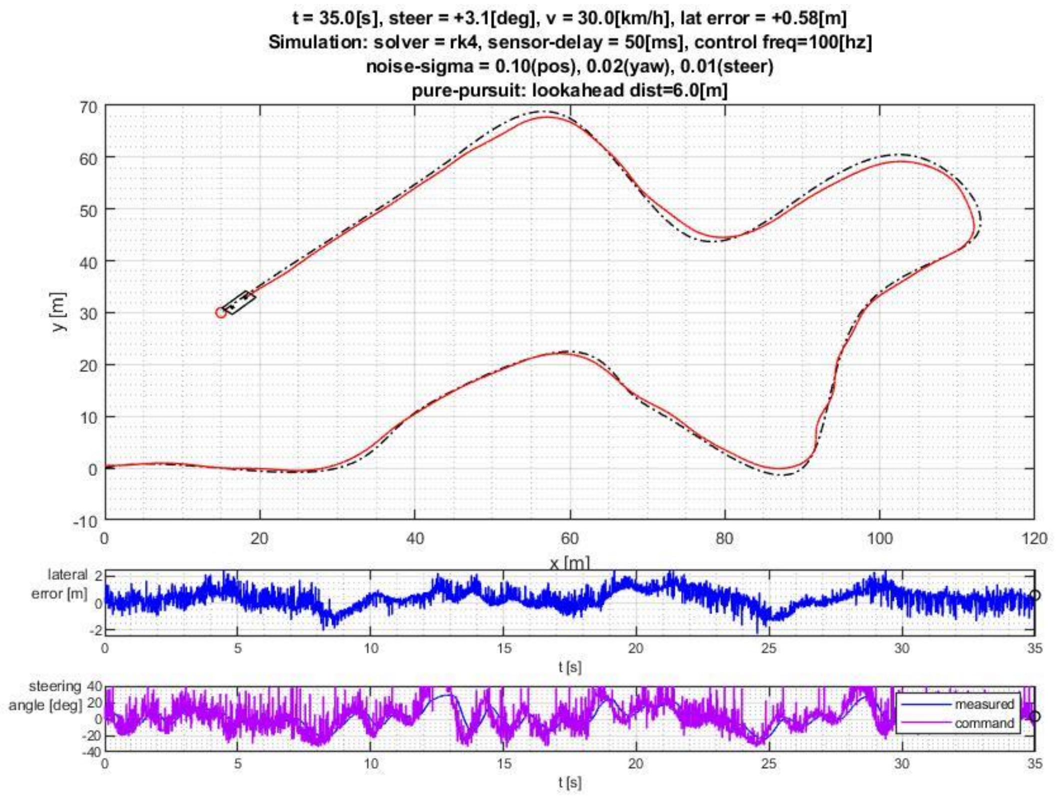

Figure 4-5. Simulation Result of Pure Pursuit Algorithm With EKF

In Figure 4-6, the simulation result shows that the performance of PD feedback control is increased since the trajectory tracking is better than when the extended Kalman filter was not applied. However, it shows that the fluctuation between 10 and 15 seconds happened when the simulation is running. And between 25 to 30 second and 10 to 15 second, the steering angle change rate increased drastically as same as the simulation without the extended Kalman filter, resulting in the unstable state. But as a trajectory tracking performance, the simulation with the extended Kalman filter shows better result than when the simulation was exploited without the filter.

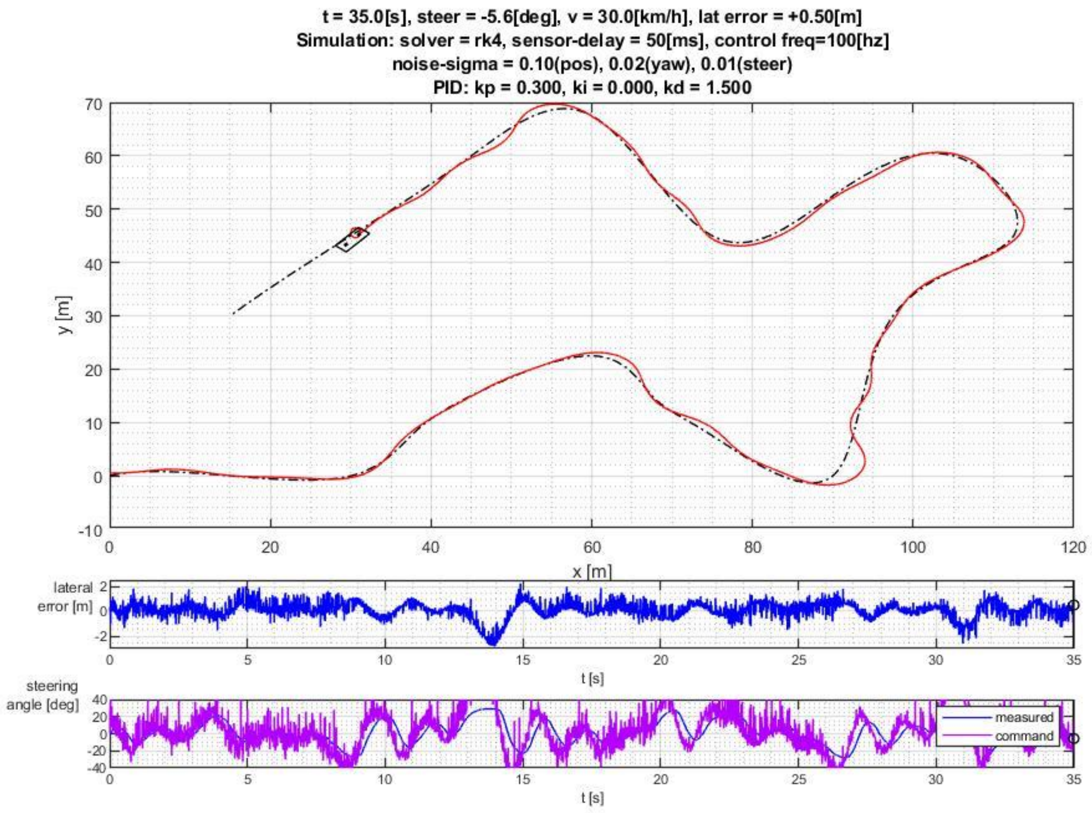

Figure 4-6. Simulation Result of PD Feedback Control with EKF

Figure 4-7 shows the simulation result of MPC with the extended Kalman filter. In Figure 4-7, it shows that the performance of MPC is mostly same as the MPC without the filter. However, the steering angle change rate and the latitude error change rate are shown as unstable, resulting in the decrease of the robustness.

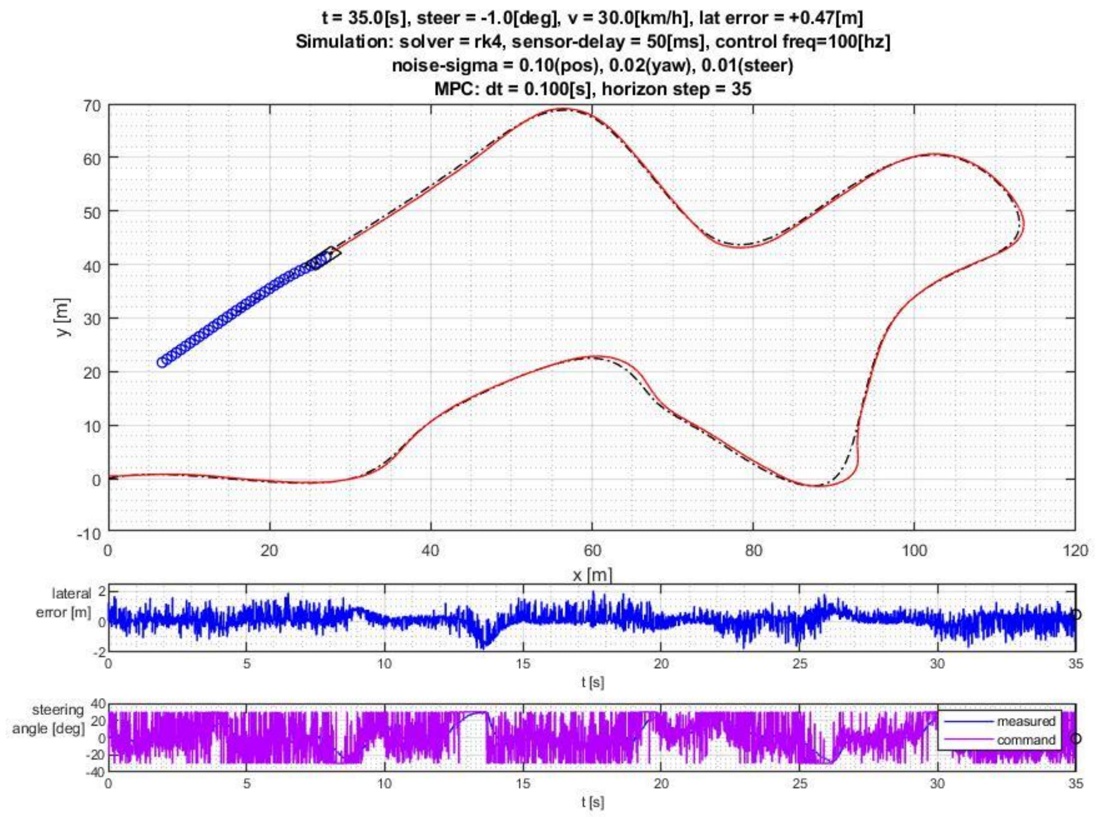

Figure 4-7. Simulation Result of MPC with EKF.

To sum up, the performance of path tracking algorithm becomes better than when the path tracking algorithm simulation is performed without filter. Trajectory tracking becomes better with the filter. However, the simulation result also shows that the stability and robustness of path tracking algorithm are decreased. The steering angle and the latitude error change rate is increased, resulting in the decrease of stability in autonomous driving.

CHAPTER 5
CONCLUSION AND FUTURE WORK

5.1 Conclusion

This research is performed by using MATLAB 2020b programming to simulate the vehicle model following the path tracking algorithm to track the trajectory. The purpose of this research is to enhance the performance of trajectory tracking and optimize the trajectory by applying the extended Kalman filter.

About the result of the experiment, the simulation was conducted with constant speed, initial low noise error, and constant parameters of the vehicle model to check two variables: steering angle and latitude error. The simulation was conducted with two conditions: the simulation without the extended Kalman filter, and the simulation with the extended Kalman filter. The extended Kalman filter, in this research, was utilized to estimated the position data of the vehicle model after previous position data was changed by Gaussian noise. The filter calculated the position, $x, y, and\ yaw$ data, and this data goes to the path tracking controller to find the steering angle.

The result shows that the extended Kalman filter has an effect on the performance of tracking the optimal trajectory for the autonomous driving. Especially in PD feedback controller simulation, it is distinguished that trajectory tracking became better than when simulation was conducted without the extended Kalman filter.

However, in the result of steering angle and the latitude error change rate, it became much unstable than when the extended Kalman filter was not utilized. This result tells that applying the extended Kalman filter can be a way to increase the performance of trajectory tracking, but it can also decrease the stability of path tracking algorithm. Therefore, the purpose to enhance the trajectory tracking performance and finding the optimal trajectory can be seen as a minor success.

5.2 Future Work

This research was conducted under the limited conditions for the simulation. In the real world, the velocity of the vehicle changes continuously, and the vehicle model mostly follows the nonlinear dynamics during the autonomous driving. Since the stability of autonomous vehicles is important, especially when the vehicle drives the complicated road environment that includes lots of curve, such as roads on the mountain, future research will focus on finding resolution to achieve the stability of path tracking algorithm under the road environment as similar as the real road environment does.

Next future research will also include the Stanley controller for the simulation. In this research, Stanley controller was not considered as a controller simulation. This is because Stanley controller was not considered as a controller simulation for this research since it was necessary to run the simulation based on different variables and constraints unlike other path tracking algorithms, and it needs more time to make the code and run the simulation separately.

APPENDIX A
STRUCTURE OF THE CODE

This research is performed by using MATLAB 2020b programming. This simulation can be divided into six packages – path generate, kinematic model of Ackermann steering algorithm, path tracking algorithms, main simulation, extended Kalman filter, and simulation running code with or without the adjustment of the extended Kalman filter are described in Table 1 below. Each package and their descriptions are below:

Table A-1. Path generate

Code function name	Description	Code result
PathDesign.m	Generate the reference path for the vehicle	Path for the simulation is created

Table A-2. Kinematic model of Ackermann steering algorithm

Code function name	Description	Code result
kinematics_model.m	Generate the derivative value of position and steering angle	The derivative value of position and steering angle

Table A-3. Path tracking algorithms

Code function name	Description	Code result
pure_pursuit.m	Receive the information about the initial position and generate the steering angle for the path tracking.	Steering angle, lookahead distance and velocity are calculated
pid_control.m	Receive the information about the initial position and generate the steering angle for the path tracking. PD control is conducted.	Steering angle and position data are calculated
model_predictive_controller.m	Receive the information about the initial position and generate the steering angle for the path tracking.	Steering angle and position data are calculated

Table A-4. Main simulation

Code function name	Description	Code result
main_simulation.m	Run the whole process and show the result of the simulation.	Trajectory tracking simulation video, the change of steering angle, velocity, yaw rate, and position data

Table A-5. Extended Kalman filter.

Code function name	Description	Code result
EKF.m	Calculation for the estimated position data from the bicycle model.	Position data: x, y, yaw and velocity.

Table A-6. Simulation for running the control process

Code function name	Description	Code result
Simulate_rk4.m	Calculation process by combining vehicle model algorithm, and path tracking. Extended Kalman filter is applied when it is needed.	Estimated Trajectory for the vehicle.

APPENDIX B
SOURCE CODE

1. PathDesign.m

```
%% Design path from points by spline

point = [0, 0;
    1, 0;
    2, 0;
    2.5, 0.5;
    3, 1;
    4, 1.5;
    4.5, 1;
    5, 0.5;
    6, 0;
    6.5, 2;
    7.5, 3;
    7, 4;
    6, 3.5;
    5, 3;
    4, 4.5;
    3, 4;
    2, 3;
    1, 2;
    ];

s = 1:1:length(point);


px_spline = spline(s, point(:,1), 1:0.01:length(point));
py_spline = spline(s, point(:,2), 1:0.01:length(point));

p_spline = [px_spline', py_spline'];


%% Insert yaw of the path direction

yaw = zeros(length(px_spline), 1);
for i = 2:length(px_spline)-1
    x_forward = px_spline(i+1);
    x_backward = px_spline(i-1);
    y_forward = py_spline(i+1);
    y_backward = py_spline(i-1);
    yaw(i) = atan2(y_forward-y_backward, x_forward-x_backward);
end
yaw(1) = yaw(2);
yaw(end) = yaw(end-1);


%% Plot with the quiver

arrow_scale = 0.01;
```

```
figure(101);
plot(point(:,1), point(:,2), 'bo-'); hold on;
quiver(px_spline', py_spline', cos(yaw)*arrow_scale, sin(yaw)*arrow_scale);
plot(px_spline, py_spline,'r-'); grid on; hold off;

%% save path for the simulation
path = [px_spline', py_spline', yaw];

save('path', 'path')
```

2. kinematics_model.m

```
function d_state = kinematics_model(state, input, param)

% Find the derivative states of x,y,yaw and angular velocity of the steered
% wheel
v_des = input(1);
delta_des = input(2);

% limits by Ackermann boundary condition
delta_des = max(min(delta_des, param.steer_lim), -param.steer_lim);
v_des = max(min(v_des, param.vel_max), param.vel_min);

% x = state(1);
% y = state(2);
yaw = state(3);
delta = state(4);

v = v_des;

% Calculate
d_x = v * cos(yaw);
d_y = v * sin(yaw);
d_yaw = v * tan(delta) / param.wheelbase;
d_delta = - (delta - delta_des) / param.tau;

% add steady state error caused by friction
if abs(delta - delta_des) < param.steering_steady_state_error_deg / 180 * pi
    d_delta = 0;
end

% Result
d_state = [d_x, d_y, d_yaw, d_delta];
```

3. pure_pursuit.m

```
function [u, debug_info] = pure_pursuit(state, t, ref, param)
%
%state = [x, y, yaw, delta]
%u = [v_des, delta_des]
%ref = [x_ref; y_ref; yaw_ref; v_ref];
```

```
%

% Pure Pursuit Parameters
IDX_X = 1;
IDX_Y = 2;
IDX_XY = 1:2;
IDX_YAW = 3;
IDX_VEL = 4;
%IDX_CURVATURE = 5;
%IDX_TIME = 6;
%IDX_DELTA = 4;

% Lookahead distance
lookahead_dist = param.pure_pursuit_lookahead;

% Calculate the distance between the position and the reference path
distance = vecnorm(ref(:,IDX_XY)' - state(IDX_XY)');
[~, min_index] = min(distance);
pr = ref(min_index, :);

% Velocity
v_des = pr(IDX_VEL);

% Pure pursuit geometry
for i = min_index:length(ref)
    dist = norm(ref(i, IDX_XY) - state(:,IDX_XY));
    if dist > lookahead_dist
        break;
    end
end
lookahead_pt = ref(i, :);

alpha = atan2(lookahead_pt(IDX_Y) - state(IDX_Y), lookahead_pt(IDX_X) -
state(IDX_X)) - state(IDX_YAW);

omega_des = 2 * v_des * sin(alpha) / param.wheelbase;
delta_des = atan2(omega_des * param.wheelbase, v_des); % the steering angle

% Input: velocity and steering angle
u = [v_des, delta_des];


% latitude error calculation for debugging
sp_yaw = pr(IDX_YAW);
T_xy2lonlat = [cos(sp_yaw), sin(sp_yaw);
             -sin(sp_yaw), cos(sp_yaw)];
error_xy = (state(IDX_XY) - pr(IDX_XY))';
error_lonlat = T_xy2lonlat * error_xy;
error_lat = error_lonlat(2);

% Input 2: lookahead point, yaw, and latitude
debug_info = [lookahead_pt(IDX_X), lookahead_pt(IDX_Y),
lookahead_pt(IDX_YAW), error_lat];
```

4. pid_control.m

```
function [u, debug_info] = pid_controller(state, t, ref, param)
%
% state = [x, y, yaw, delta]
% u = [v_des, delta_des]
% ref = [x_ref; y_ref; yaw_ref; v_ref];
%

% Parameters for PID control
IDX_X = 1;
IDX_Y = 2;
IDX_XY = 1:2;
IDX_YAW = 3;
IDX_VEL = 4;
IDX_CURVATURE = 5;
% IDX_TIME = 6;


% LON = 1;
% latitude;
LAT = 2;

kp = 0.3 * 1;
kd = 0.5 * 3;

yaw = state(IDX_YAW);


distance = vecnorm(ref(:,1:2)' - state(1:2)');
[~, min_index] = min(distance);

pr = ref(min_index, :);

v_des = pr(IDX_VEL);


% feedforward input calculation
ff_curvature = atan(param.wheelbase * pr(IDX_CURVATURE));

% coordinate transformation to body frame
T = [cos(yaw), sin(yaw);
    -sin(yaw), cos(yaw)];
error_xy = (state(IDX_XY) - pr(IDX_XY))';
error_latlon = T * error_xy;
error_yaw = yaw - pr(IDX_YAW);

% Error of yaw calculation
while (-2*pi <= error_yaw && error_yaw <= 2*pi) == 0
    if (error_yaw >= 2*pi)
        error_yaw = error_yaw - 2*pi;
    elseif (error_yaw <= -2*pi)
        error_yaw = error_yaw + 2*pi;
    end
```

```
end
if (error_yaw > pi)
    error_yaw = error_yaw - 2*pi;
elseif (error_yaw < -pi)
    error_yaw = error_yaw + 2*pi;
end

% PD calculation
delta_des = -kp * (error_latlon(LAT)) - kd * error_yaw + ff_curvature;
fb_lat = -kp * (error_latlon(LAT));
fb_yaw = - kd * error_yaw;

u = [v_des, delta_des];

debug_info = [pr(IDX_X), pr(IDX_Y), pr(IDX_YAW), fb_lat, fb_yaw,
ff_curvature, error_latlon(LAT)];
```

5. model_predictive_controller.m

```
function [u, debug_info] = model_predictive_controller(state, t, ref, param)
%
% state = [x, y, yaw, delta]
% u = [v_des, delta_des]
% ref = [x_ref; y_ref; yaw_ref; v_ref; k_ref; t_ref];
%
%
%

% =================== for delay compensation ==============================
persistent deltades_buffer
if isempty(deltades_buffer)
    deltades_buffer = zeros(param.mpc_delay_comp_step, 1);
end
% =========================================================================

deg2rad = pi / 180;
delay_step = param.mpc_delay_comp_step;

IDX_X = 1;
IDX_Y = 2;
IDX_XY = 1:2;
IDX_YAW = 3;
IDX_VEL = 4;
IDX_CURVATURE = 5;
IDX_TIME = 6;

IDX_DELTA = 4;

%% convert xy-yaw to error dynamics

% calculate nearest point (use as initial state)
distance = vecnorm(ref(:, IDX_XY)' - state(IDX_XY)');
[~, min_index] = min(distance);
```

```
ref_sp = ref(min_index, :);

% convert x,y to longitude and latitude model
sp_yaw = ref_sp(IDX_YAW);
T_xy2lonlat = [cos(sp_yaw), sin(sp_yaw);
             -sin(sp_yaw), cos(sp_yaw)];
error_xy = (state(IDX_XY) - ref_sp(IDX_XY))';
error_lonlat = T_xy2lonlat * error_xy;
error_lat = error_lonlat(2);



% calculate yaw error
error_yaw = state(IDX_YAW) - sp_yaw;
while (-2*pi <= error_yaw && error_yaw <= 2*pi) == 0
    if (error_yaw >= 2*pi)
        error_yaw = error_yaw - 2*pi;
    elseif (error_yaw <= -2*pi)
        error_yaw = error_yaw + 2*pi;
    end
end
if (error_yaw > pi)
    error_yaw = error_yaw - 2*pi;
elseif (error_yaw < -pi)
    error_yaw = error_yaw + 2*pi;
end

% initial state for error dynamics
x0 = [error_lat; error_yaw; state(IDX_DELTA)];

%% update error dynamics for the horizon

% -- set mpc parameters --
mpc_dt = param.mpc_dt;
mpc_n = param.mpc_n;
Q = param.mpc_Q;
R = param.mpc_R;
mpc_t = ref_sp(IDX_TIME);
DIM_X = 3;
DIM_Y = 2;
DIM_U = 1;
Aex = zeros(DIM_X*mpc_n, DIM_X);
Bex = zeros(DIM_X*mpc_n, DIM_U*mpc_n);
Wex = zeros(DIM_X*mpc_n, 1);
Cex = zeros(DIM_Y*mpc_n, DIM_X*mpc_n);
Qex = zeros(DIM_Y*mpc_n, DIM_Y*mpc_n);
Rex = zeros(DIM_U*mpc_n, DIM_U*mpc_n);
mpc_ref_v = zeros(mpc_n + delay_step, 1);
debug_ref_mat = zeros(mpc_n + delay_step,5);

% =================== for delay compensation ==============================
% -- apply delay compensation : update dynamics with increasing mpt_t --
x_curr = x0;
for i = 1:delay_step
    if mpc_t > ref(end, IDX_TIME)
        mpc_t = ref(end, IDX_TIME);
        disp('[MPC] path is too short to predict dynamics');
    end
    ref_now = interp1q(ref(:, IDX_TIME), ref(:,1:5), mpc_t);
    debug_ref_mat(i,:) = ref_now;
    v_ = ref_now(IDX_VEL);
    k_ = ref_now(IDX_CURVATURE);

    % get discrete state matrix
    % NOTE : use control_dt as delta time, not mpc_dt.
    [Ad, Bd, wd, ~] = get_error_dynamics_state_matrix(param.control_dt, v_,
param.wheelbase, param.tau, k_);
    u_now = deltades_buffer(end - i + 1);
    x_next = Ad * x_curr + Bd * u_now + wd;

    mpc_t = mpc_t + param.control_dt; % THIS IS NOT mpc_dt, BUT control_dt
    x_curr = x_next;

    mpc_ref_v(i) = v_;
end
x0 = x_curr;
% ==========================================================================

% -- mpc matrix for i = 1 --
ref_i_ = interp1q(ref(:, IDX_TIME), ref(:,1:5), mpc_t);
debug_ref_mat(1 + delay_step,:) = ref_i_; % MODIFIED FOR DELAY
v_ = ref_i_(IDX_VEL);
k_ = ref_i_(IDX_CURVATURE);
[Ad, Bd, wd, Cd] = get_error_dynamics_state_matrix(mpc_dt, v_,
param.wheelbase, param.tau, k_);
Aex(1:DIM_X, :) = Ad;
Bex(1:DIM_X, 1:DIM_U) = Bd;
Wex(1:DIM_X) = wd;
Cex(1:DIM_Y, 1:DIM_X) = Cd;
Qex(1:DIM_Y, 1:DIM_Y) = Q;
Rex(1:DIM_U, 1:DIM_U) = R;

mpc_ref_v(1 + delay_step) = v_;

% -- mpc matrix for i = 2:n --
for i = 2:mpc_n

    % update mpc time
    mpc_t = mpc_t + mpc_dt;
    if mpc_t > ref(end, IDX_TIME)
        mpc_t = ref(end, IDX_TIME);
        disp('[MPC] path is too short to predict dynamics');
    end

    % get reference information
    ref_i_ = interp1q(ref(:, IDX_TIME), ref(:,1:5), mpc_t);
    debug_ref_mat(i + delay_step,:) = ref_i_;
    v_ = ref_i_(IDX_VEL);
    k_ = ref_i_(IDX_CURVATURE);

    % get discrete state matrix
    [Ad, Bd, wd, Cd] = get_error_dynamics_state_matrix(mpc_dt, v_,
param.wheelbase, param.tau, k_);

    % update mpc matrix
    idx_x_i = (i-1)*DIM_X+1:i*DIM_X;
    idx_x_i_prev = (i-2)*DIM_X+1:(i-1)*DIM_X;
    idx_u_i = (i-1)*DIM_U+1:i*DIM_U;
    idx_y_i = (i-1)*DIM_Y+1:i*DIM_Y;
    Aex(idx_x_i, :) = Ad * Aex(idx_x_i_prev, :);
    for j = 1:i-1
        idx_u_j = (j-1)*DIM_U+1:j*DIM_U;
        Bex(idx_x_i, idx_u_j) = Ad * Bex(idx_x_i_prev, idx_u_j);
    end
    Bex(idx_x_i, idx_u_i) = Bd;
    Wex(idx_x_i) = Ad * Wex(idx_x_i_prev) + wd;
    Cex(idx_y_i, idx_x_i) = Cd;
    Qex(idx_y_i, idx_y_i) = Q;
    Rex(idx_u_i, idx_u_i) = R;

    mpc_ref_v(i + delay_step) = v_;

end


%% convex optimization

mat1 = Bex' * Cex' * Qex * Cex * Bex + Rex;
mat2 = (x0' * Aex' + Wex') * Cex' * Qex * Cex * Bex;

steering_rate_lim = param.mpc_constraint_steer_rate_deg * deg2rad;

if param.mpc_solve_without_constraint == true
    input_vec = -mat1 \ mat2';
else

    H_ = (mat1 + mat1') / 2;
    f_ = mat2;

    % Add steering rate constraint
    tmp = -eye(mpc_n-1, mpc_n);
    tmp(1:end,2:end) = tmp(1:end,2:end) + eye(mpc_n-1);
    T_ = kron(tmp, [0,0,1]) / mpc_dt;
    dsteer_vec_tmp_ = T_ * (Aex * x0 + Wex);
    A_ = [T_ * Bex; -T_ * Bex];
    b_ = [steering_rate_lim * ones(mpc_n-1,1) - dsteer_vec_tmp_;
steering_rate_lim * ones(mpc_n-1,1) + dsteer_vec_tmp_];

    % Constraints for upper and lower steering boundary
    lb_ = -param.mpc_constraint_steering_deg * deg2rad * ones(mpc_n *
DIM_U,1);
    ub_ = param.mpc_constraint_steering_deg * deg2rad * ones(mpc_n *
DIM_U,1);
    options_ = optimoptions('quadprog', 'Algorithm','interior-point-convex',
'Display', 'off');
```

```
    [x_, fval, exitflag, output, lambda] = quadprog(H_, f_, A_, b_, [], [],
lb_, ub_, [], options_);
    input_vec = x_;

end

delta_des = input_vec(1);
v_des = ref_sp(IDX_VEL);
u = [v_des, delta_des];

% =================== for delay compensation ==============================
deltades_buffer = [delta_des; deltades_buffer(1:end-1)];
% =========================================================================

%% (debug) Calculate predicted trajectory

x_ = state;
predictd_states = zeros(length(input_vec), length(state));
for i = 1:length(input_vec)
    x_next = calc_kinematics_model(x_, input_vec(i), mpc_dt, mpc_ref_v(i),
param.wheelbase, param.tau);
    predictd_states(i,:) = x_next';
    x_ = x_next;
end

predictd_states_vector = reshape(predictd_states, [], 1);

debug_ref_mat_no_delay_comp = debug_ref_mat(delay_step + 1:end, :);

predicted_error = Aex*x0 + Bex*input_vec + Wex;
predicted_error = transpose(reshape(predicted_error, 3, []));
predicted_state_ideal = debug_ref_mat_no_delay_comp(:,IDX_XY) + ...
    [-sin(debug_ref_mat_no_delay_comp(:,IDX_YAW)).*predicted_error(:,1),
cos(debug_ref_mat_no_delay_comp(:,IDX_YAW)).*predicted_error(:,1)];

predicted_state_ideal = (reshape(predicted_state_ideal, [], 1));

debug_info = [input_vec', predictd_states_vector', predicted_state_ideal',
error_lat];


end

%% time varying error dynamics model
% used for error dynamics update
function [Ad, Bd, wd, Cd] = get_error_dynamics_state_matrix(dt, v, L, tau,
curvature)

    % linearization around delta = 0
    % A = [0, v, 0;
    %      0, 0, v/L;
    %      0, 0, -1/tau];
    % B = [0; 0; 1/tau];
    % C = [1, 0, 0;
    %       0, 1, 0];
    % w = [0;
    %      -v*curvature;
    %       0];

    % linearization around delta = delta_ref (better accuracy than delta=0)
    delta_r = atan(L*curvature);
    if (abs(delta_r) >= 40 /180 * pi)
        delta_r = (40 /180 * pi)*sign(delta_r);
    end
    cos_delta_r_squared_inv = 1 / ((cos(delta_r))^2);

    % difinition for continuous model
    A = [0, v, 0;
         0, 0, v/L*cos_delta_r_squared_inv;
         0, 0, -1/tau];
    B = [0; 0; 1/tau];
    C = [1, 0, 0;
         0, 1, 0];
    w = [0;
        -v*curvature + v/L*(tan(delta_r)-delta_r*cos_delta_r_squared_inv);
         0];

    % discretization
    % Ad = eye(3) + A * dt;
    I = eye(3);
    Ad = (I - dt * 0.5 * A) \ (I + dt * 0.5 * A);
    Bd = B * dt;
    Cd = C;
    wd = w * dt;
end

%% xy-yaw dynamics model
% used for debug (predicted trajectory calculation)
function x_next = calc_kinematics_model(x, u, dt, v, L, tau)

    % x = [x, y, yaw, delta]
    x_next = zeros(4,1);
    yaw = x(3);
    delta = x(4);

    % x
    x_next(1) = x(1) + v*cos(yaw)*dt;

    % y
    x_next(2) = x(2) + v*sin(yaw)*dt;

    % yaw
    x_next(3) = x(3) + v*tan(delta)/L*dt;

    % delta
    x_next(4) = x(4) - (x(4) - u)/tau*dt;

end
```

6. main_simulation.m

```
%
% state = [x, y, yaw, delta]
% input = [v_des, delta_des]
% ref = [x_ref, y_ref, yaw_ref, v_ref]
%
%
%

clear variables;
close all;


set(0, 'defaultAxesFontSize', 10);
set(0, 'defaultTextFontSize', 12);
set(0, 'DefaultAxesLineWidth', 1.0, 'DefaultLineLineWidth', 1.0);


addpath ../Path

control_mode_option = ["pure_pursuit", "pid", "mpc"];
control_mode = control_mode_option(1);

%% preliminaries
rad2deg = 180 / pi;
deg2rad = pi / 180;
kmh2ms = 1000 / 3600;

simulation_time = 35;
simulation_rk4_time_step = 0.002; % simulation time step

vel_ref = 30 * kmh2ms;

% for dynamics model
param.tau = 0.27; % steering dynamics: 1d-approximated time constant
param.wheelbase = 2.5;
param.steer_lim = 30 * deg2rad;
param.vel_max = 10;
param.vel_min = -5;

param.input_delay = 0.05; % [s]
param.control_dt = 0.01; % [s]
param.measurement_noise_stddev = [0.1, 0.1, 1.0*deg2rad, 0.5*deg2rad]; %
measurement noise
%param.measurement_noise_stddev = [0,0,0,0]; % measurement noise
param.steering_steady_state_error_deg = 1;

% for pure pursuit only
param.pure_pursuit_lookahead = 6.0; % [m]

% for mpc only
```

```
param.mpc_dt = 0.1;
param.mpc_n = 35;
param.mpc_constraint_steering_deg = 30;
param.mpc_constraint_steer_rate_deg = 280;
param.mpc_model_dim = 3;
param.mpc_Q = diag([1,2]);
param.mpc_R = 0.5;
param.mpc_delay_comp_step = round(param.input_delay / param.control_dt);
%param.mpc_delay_comp_step = 0.0;

% use the input ahead of the delay time
param.mpc_sensor_delay = param.input_delay;


%% simulation parameters

% initial position (x, y, yaw, delta)
x0 = [0, 0.5, 0, 0];

ts = 0;
dt = simulation_rk4_time_step;
tf = simulation_time;
t = ts:dt:tf;

%% reference trajectory design

PathDesign; % using spline
load path; % x, y, yaw

ref = zeros(length(path), 6);
IDX_X = 1;
IDX_Y = 2;
IDX_XY = 1:2;
IDX_XYYAW = 1:3;
IDX_YAW = 3;
IDX_VEL = 4;
IDX_CURVATURE = 5;
IDX_TIME = 6;

IDX_STEER = 4;


path_size_scale = 15;
path(:,IDX_XY) = path(:,IDX_XY) * path_size_scale;
ref(:,IDX_XYYAW) = path(:,IDX_XYYAW);

ref(:,IDX_VEL) = ones(length(path),1)*vel_ref;

% insert curvature into path
for i = 2:length(ref)-1
    p1_ = ref(i-1,IDX_XY);
    p2_ = ref(i, IDX_XY);
    p3_ = ref(i+1, IDX_XY);
    A_ = ((p2_(1)-p1_(1))*(p3_(2)-p1_(2)) - (p2_(2)-p1_(2))*(p3_(1)-p1_(1)))
/ 2;
    ref(i, IDX_CURVATURE) = 4 * A_ / (norm(p1_-p2_) * norm(p2_-p3_) *
norm(p3_-p1_));
end

% insert relative time into path
for i = 2:length(ref)
    v_ = ref(i,IDX_VEL);
    d_ = norm(ref(i,IDX_XY)-ref(i-1,IDX_XY));
    dt_ = d_ / v_;
    ref(i, IDX_TIME) = ref(i-1, IDX_TIME) + dt_;
end


%% simulation

if control_mode == "pure_pursuit"
    [X, U, debug] = simulate_rk4(@kinematics_model, @pure_pursuit, x0, ref,
ts, dt, tf, param);
    lat_error_vec = debug(:,end);
elseif control_mode == "pid"
   [X, U, debug] = simulate_rk4(@kinematics_model, @pid_controller, x0, ref,
ts, dt, tf, param);
   lat_error_vec = debug(:,end);
elseif control_mode == "mpc"
    param.mpc_solve_without_constraint = false;
    [X, U, debug] = simulate_rk4(@kinematics_model,
@model_predictive_controller, x0, ref, ts, dt, tf, param);
    lat_error_vec = debug(:,end);
end
fprintf("lattitude error: mean square = %f, max = %f",
norm(lat_error_vec)/simulation_time, max(lat_error_vec));


%% movie plot

sp_num = 18;
subpl1 = 'subplot(sp_num,sp_num, sp_num+1:sp_num*12);';
subpl2 = 'subplot(sp_num,sp_num, sp_num*13+1:sp_num*15);';
subpl3 = 'subplot(sp_num,sp_num, sp_num*16+1:sp_num*18);';

% tire2steer = 12.5;

fig_trajectory_result = figure(1);
set(fig_trajectory_result, 'Position', [700 700 1000 1000]);

eval(subpl1);
plot(ref(:,1), ref(:,2),'k-.'); grid on; grid minor; hold on;
% plot(X(:,1), X(:,2));
% legend('ref','tracked');
xlabel('x [m]'); ylabel('y [m]');
% img_orig = imread('handle.jpg');
% img = imrotate(img_orig, 10);
% handle_plt = image([150, 170],[110, 90], img);
% handle_plt_point = [160, 100, 0];

eval(subpl2);
plot(t, lat_error_vec, 'b'); grid on; grid minor; hold on;
xlabel('t [s]'); ylabel({'latitude','error [m]'});
ylh = get(gca,'ylabel');
gyl = get(ylh);                                                      %
Object Information
ylp = get(ylh, 'Position');
set(ylh, 'Rotation',0, 'Position',ylp, 'VerticalAlignment','bottom',
'HorizontalAlignment','right')
ulim = ceil(2*max(lat_error_vec))/2;
dlim = floor(2*min(lat_error_vec))/2;
ylim([dlim, ulim]);

eval(subpl3);
p1 = plot(t, X(:,IDX_STEER)*rad2deg, 'b'); grid on; grid minor; hold on;
p2 = plot(t, U(:,2)*rad2deg, 'Color', [0.7 0. 1]); hold on;
legend([p1,p2], {'measured','command'})
xlabel('t [s]'); ylabel({'steering','angle [deg]'},'Rotation',0);
y2h = get(gca,'ylabel');
gyl = get(y2h);                                                      %
Object Information
y2p = get(y2h, 'Position');
set(y2h, 'Rotation',0, 'Position',y2p, 'VerticalAlignment','bottom',
'HorizontalAlignment','right')
ulim = round(2*max(X(:,IDX_STEER)*rad2deg))/2;
dlim = round(2*min(X(:,IDX_STEER)*rad2deg))/2;
ylim([dlim, ulim]);

z_axis = [0 0 1];
setpoint = []; rear_tire = []; front_tire = []; body = []; tracked = [];
setpoint_ideal = []; error_point = []; steer_point = []; time_bar_laterror =
[]; time_bar_steer = [];
L = param.wheelbase;
rear_length = 1;
front_length = 1;
side_width = 0.9;
fig_draw_i = 1:round(1/dt/20):length(t);

% for movie
clear frame_vec;
frame_vec(length(fig_draw_i)) = struct('cdata', [], 'colormap',[]);

j = 1;
fig_trajectory_result; hold on;
for i = fig_draw_i
    eval(subpl1);
    disp(t(i))
    rear_x = X(i,1);
    rear_y = X(i,2);
    yaw = X(i,3);
    delta = X(i,IDX_STEER);
    front_x = rear_x + L;
    front_y = rear_y;
    delete([setpoint, rear_tire, front_tire, body, tracked, setpoint_ideal,
error_point, steer_point, time_bar_laterror, time_bar_steer]);
    tracked = plot(X(1:i,1), X(1:i,2),'r');
    title_draw = "t = "+num2str(t(i),'%5.1f') + "[s], steer = " +
num2str(delta*rad2deg,'%+3.1f') + "[deg], v = " + ...
        num2str(vel_ref*3600/1000,'%3.1f') + "[km/h], lat error =
"+num2str(lat_error_vec(i),'%+2.2f') + "[m]";
    title_draw = [title_draw; "Simulation: solver = rk4, sensor-delay = "  +
num2str(param.input_delay*1000, '%d') + "[ms], control freq=" + ...
        num2str(1/param.control_dt, '%d') + "[hz]"];
    title_draw = [title_draw; "noise-sigma = " +
num2str(param.measurement_noise_stddev(1),'%2.2f')+"(pos), "+ ...
        num2str(param.measurement_noise_stddev(3),'%2.2f')+"(yaw),
"+num2str(param.measurement_noise_stddev(4),'%2.2f')+"(steer)"];
    if control_mode == "mpc"
        title_draw = [title_draw; "MPC: dt = " + num2str(param.mpc_dt,
'%3.3f') + "[s], horizon step = " + num2str(param.mpc_n, '%d')];
        pred_states = debug(i, param.mpc_n+1:param.mpc_n*(4+1));
        pred_states = reshape(pred_states, param.mpc_n,
length(pred_states)/param.mpc_n);
        setpoint = plot(pred_states(:,1), pred_states(:,2), 'bo'); % include
linealize error
        %pred_error = debug(i, param.mpc_n*(4+1)+1:param.mpc_n*(2+4+1));
        %pred_error = reshape(pred_error, param.mpc_n,
length(pred_error)/param.mpc_n);
        %setpoint_ideal = plot(pred_error(:,1), pred_error(:,2), 'mx'); %
without linealize error
    elseif control_mode == "pure_pursuit"
        title_draw = [title_draw; "pure-pursuit: lookahead
dist="+num2str(param.pure_pursuit_lookahead, '%1.1f')+"[m]"];
        sp = debug(i,:);
        setpoint = plot(sp(1), sp(2), 'ro');
    elseif control_mode == "pid"
        title_draw = [title_draw; "PID: kp = " + num2str(0.3, '%3.3f') + ",
ki = " + num2str(0, '%3.3f') + ", kd = " + num2str(1.5, '%3.3f')];
        sp = debug(i,:);
        setpoint = plot(sp(1), sp(2), 'ro');
    end
    rear_tire = plot([rear_x-0.3, rear_x+0.3],[rear_y, rear_y], 'k',
'LineWidth', 2.0);
    front_tire = plot([front_x-0.3, front_x+0.3],[front_y, front_y], 'k',
'LineWidth', 2.0);
    body = plot([rear_x-rear_length, front_x+front_length,
front_x+front_length, rear_x-rear_length, rear_x-rear_length], ...
        [rear_y-side_width, front_y-side_width, front_y+side_width,
rear_y+side_width, rear_y-side_width],'k');
    rear_origin = [rear_x, rear_y, 0];
    front_origin = [rear_x + L*cos(yaw), rear_y + L*sin(yaw), 0];
    rotate(body, z_axis, yaw * rad2deg, rear_origin);
    rotate(rear_tire, z_axis, yaw * rad2deg, rear_origin);
    rotate(front_tire, z_axis, yaw * rad2deg, rear_origin);
    rotate(front_tire, z_axis, delta * rad2deg, front_origin);
%   rotate(handle_plt, [0,0,1], delta*tire2steer*rad2deg, handle_plt_point)
    title(title_draw);
    xlim([0 120]);

    % lat error
    eval(subpl2);
    error_point = plot(t(i), lat_error_vec(i), 'ko');
    time_bar_laterror = plot([t(i), t(i)], [100, -100], 'k');

    % steering
    eval(subpl3);
    steer_point = plot(t(i), X(i, IDX_STEER)*rad2deg, 'ko');
    time_bar_steer = plot([t(i), t(i)], [100, -100], 'k');
    legend([p1,p2], {'measured','command'})
    ylim([-40 40]);


    drawnow;
    frame_vec(j) = getframe(fig_trajectory_result);

    j = j + 1;
end
```

7. EKF.m

```
function [x_pos_filtered] = EKF(x_position,x)
%% settings
dt = 1; % time between time steps
sig_mea_true = [0.02; 0.02; 0.4]; % true value of standard deviation of
measurement noise
sig_pro = [0.02; 0.02; 0.4]; % user input of standard deviation of process
noise
sig_mea = [0.02; 0.02; 0.4]; % user input of standard deviation of
measurement noise
sig_init = [0; 0.5; 0;]; % standard deviation of initial guess
Q = diag(sig_pro.^2); % process noise covariance matrix
R = diag(sig_mea.^2); % measurement noise covariance matrix
F = eye(3)*dt; % state transition matrix
B = eye(3); % control-input matrix
u = zeros(3,1); % control vector
H = zeros(3,3); % measurement matrix - to be determined
%% Position data
% Real location
x_real = x(1:3);
% Measuring location
x_target = x_position;
%% extended Kalman filter simulation
% filtering
    % initial guess
    x_est = x_real + normrnd(0, sig_init)';
    P = [eye(3)*sig_init(1)^2];
    %% Prediction

    % predicted state estimate
    x_est = F*x_est' + B*u;

    % predicted error covariance
    P = F*P*F' + Q;

    %%% Update
```

```
     % obtain measurement
       p = x_real - x_target; % true relative position
       z_true = [atan2(p(1), p(2));
                 atan2(p(3), sqrt(p(1)^2 + p(2)^2));
                 norm(p)]; % true measurement

       z = z_true + normrnd(0, sig_mea_true); % erroneous measurement

     % predicted meausrement
       pp = x_est' - x_target; % predicted relative position
       z_p = [atan2(pp(1), pp(2));
              atan2(pp(3), sqrt(pp(1)^2 + pp(2)^2));
              norm(pp)]; % predicted measurement

     % measurement residual
       y = z - z_p;

     % measurement matrix
       H = [pp(2)/(pp(1)^2+pp(2)^2), -pp(1)/(pp(1)^2+pp(2)^2), 0;
           -pp(1)*pp(3)/(pp*pp')/norm(pp(1:2)), -
pp(2)*pp(3)/(pp*pp')/norm(pp(1:2)), 1/norm(pp(1:2));
           pp(1)/norm(pp), pp(2)/norm(pp), pp(3)/norm(pp)];

     % Kalman gain
       K = P*H'/(R+H*P*H');

     % updated state estimate
       x_est = x_est + K*y;
       x_estimated_pos = x_est';
     % updated error covariance
       P = (eye(3) - K*H)*P;

%% get result
x_pos_filtered = mean(x_estimated_pos,3);
```

8. simulate_rk4.m

```
function [state_log, input_log, debug_info] = simulate_rk4(model, controller,
x0, ref, ts, dt, tf, param)

x = x0;
i = 1;

t_vec = ts:dt:tf;

state_log = zeros(length(t_vec), length(x0));
[tmp_u, tmp_u_debug] = controller(x0, ts, ref, param);
input_log = zeros(length(t_vec), length(tmp_u));
debug_info = zeros(length(t_vec), length(tmp_u_debug));

input_delay = param.input_delay;
delay_count = round(input_delay / dt);
input_buf = zeros(delay_count, length(tmp_u));
u = zeros(size(tmp_u)); % initial input
```

```
u_debug = zeros(size(tmp_u_debug));

control_dt = param.control_dt;
control_count = round(control_dt / dt);

for t = ts:dt:tf
    % -- control once per control_count time --
    if mod(i, control_count) == 0
        % add noise
        x_noised = x + rand(1, length(x0)) .* param.measurement_noise_stddev;
        %% When the filter is applied
        %x_position = x_noised(1:3);
        %[x_pos_filtered] = EKF(x_position, x);
        %x_estimated = [x_pos_filtered, x_noised(:,4)];
        %[u, u_debug] = controller(x_estimated, t, ref, param);
        %% When the filter is not applied
         [u, u_debug] = controller(x_noised, t, ref, param);
    end

    % -- add input delay --
    input_buf = [u; input_buf(1:end-1, :)];
    u_delayed = input_buf(end,:);

    % -- runge-kutta --
    k1 = model(x, u_delayed, param);
    k2 = model(x + k1*dt/2, u_delayed, param);
    k3 = model(x + k2*dt/2, u_delayed, param);
    k4 = model(x + dt*k3, u_delayed, param);
    x = x + (k1 + 2*k2 + 2*k3 + k4) * dt / 6;

    % -- save data --
    state_log(i,:) = x;
    input_log(i,:) = u;
    debug_info(i,:) = u_debug;

    i = i + 1;
    disp(t);
end
```

LIST OF REFERENCES

Abed, M. E., Aly, M., Ammar, H. H., & Shalaby, R. (2020). Steering Control for Autonomous Vehicles Using PID Control with Gradient Descent Tuning and Behavioral Cloning. *2nd Novel Intelligent and Leading Emerging Sciences Conference, NILES 2020*, 583–587. https://doi.org/10.1109/NILES50944.2020.9257946

Andersen, H., Chong, Z. J., Eng, Y. H., Pendleton, S., & Ang, M. H. (2016). Geometric path tracking algorithm for autonomous driving in pedestrian environment. *IEEE/ASME International Conference on Advanced Intelligent Mechatronics, AIM*, *2016-Septe*, 1669–1674. https://doi.org/10.1109/AIM.2016.7577010

Chen, Y., Shan, Y., Chen, L., Huang, K., & Cao, D. (2018). Optimization of Pure Pursuit Controller based on PID Controller and Low-pass Filter. *IEEE Conference on Intelligent Transportation Systems, Proceedings, ITSC*, *2018-Novem*, 3294–3299. https://doi.org/10.1109/ITSC.2018.8569416

Cuenca, Á., Zhan, W., Salt, J., Alcaina, J., Tang, C., & Tomizuka, M. (2019). A remote control strategy for an autonomous vehicle with slow sensor using kalman filtering and dual-rate control. *Sensors (Switzerland)*, *19*(13). https://doi.org/10.3390/s19132983

Franklin, G., Powell, J. D., & Emami-Naeini, A. (1994). Feedback control of dynamic systems, 3e. In *American Society of Mechanical Engineers, Dynamic Systems and Control Division (Publication) DSC* (Vols. 55–2). https://doi.org/10.1016/0005-1098(92)90050-p

Giesbrecht, J. (2004). Ground Vehicles. *Journal of Terramechanics*, *December*, 1–58.

Giesbrecht, J., Mackay, D., Collier, J., Verret, S., & Suffield, D. (2005). *Path Tracking for Unmanned Ground Vehicle Navigation Implementation and Adaptation of the Pure Pursuit Algorithm Defence Research and Recherche et développement Development Canada pour la défense Canada*. *December*.

Jazar, R. N. (2019). Advanced vehicle dynamics. In *Advanced Vehicle Dynamics*. https://doi.org/10.1007/9783030130626

Kim, D. J., Lee, S. H., & Chung, C. C. (2019). Kalman Filter based Path Generation and Tracking Control for Autonomous In-door Valet Parking System. *International Conference on Control, Automation and Systems*, *2019-Octob*(Iccas), 700–705. https://doi.org/10.23919/ICCAS47443.2019.8971557

Kim, Y., & Bang, H. (2019). Introduction to Kalman Filter and Its Applications. *Introduction and Implementations of the Kalman Filter*, *September*. https://doi.org/10.5772/intechopen.80600

Lundström, P., Lee, J. H., Morari, M., & Skogestad, S. (1995). Limitations of dynamic matrix control. *Computers and Chemical Engineering*, *19*(4), 409–421. https://doi.org/10.1016/0098-1354(94)00063-T

Park, M., Lee, S., & Han, W. (2015). Development of steering control system for autonomous vehicle using geometry-based path tracking algorithm. *ETRI Journal*, *37*(3), 617–625. https://doi.org/10.4218/etrij.15.0114.0123

Qin, S. J., & Badgwell, T. A. (2003). Process Control Dynamic. *Control Engineering Practice*, *11*, 733–764.

Samak, C., Samak, T., & Kandhasamy, S. (2011). Control strategies for autonomous vehicles. *ArXiv*.

Snider, J. M. (2009). Automatic Steering Methods for Autonomous Automobile Path Tracking. *Work*, *February*, 1–78. http://www.ri.cmu.edu/pub_files/2009/2/Automatic_Steering_Methods_for_Autonomous_Automobile_Path_Tracking.pdf

Vavasis, S. (1998). *Convex Optimization*. https://doi.org/10.1201/9781420049503-c34

Vu, T. M. (2012). Vehicle steering dynamic calculation and simulation. *23rd DAAAM International Symposium on Intelligent Manufacturing and Automation 2012*, *1*(1), 237–242.

Wallace, R., Stentz, A., Thorpe, C., Moravec, H., Whittaker, W., & Kanade, T. (1985). First Results in Robot Road Following. *Computers in Engineering, Proceedings of the International Computers in Engineering Conference And*, *1*, 381–387.

Wang, W., Chen, Y., Diekel, Z. M., & Jia, Y. (2018). Cost Functions based Dynamic Optimization for Robot Action Planning. *ACM/IEEE International Conference on Human-Robot Interaction*, 277–278. https://doi.org/10.1145/3173386.3177021

BIOGRAPHICAL SKETCH

Jin-hyuk Yu was born in 1993 in Seoul, South Korea. He completed his Bachelor of Science in Mechanical Engineering at the Kookmin University. In the middle of his undergraduate course, he served the military service for about two years as a Fire Direction Controller (FDC). He had participated the Exchange student course at Northeastern Illinois University, Chicago, Illinois for six months. For future career, he pursued to do the Master of Science in Mechanical Engineering at the University of Florida. Under the lead of his advisor, Dr. Carl D. Crane, he participated in his project ‘Indy Autonomous Challenge’, and did various works with his laboratory members. His current research goes to Autonomous Vehicle and Control. After his graduation, he will continue to do research further in Autonomous Vehicle and Control area and seek for a job related to this. His research interests are ‘Autonomous Vehicle, Robotics, Optimal control and UAV control.