ARTICLE

Implementation of Lateral Control Algorithms for Autonomous Vehicles

Aakash Dammala and Pranav Koli

Abstract

This paper evaluates Model Predictive Control (MPC), Pure Pursuit (PP), and Linear Quadratic Regulator (LQR) algorithms and uses dynamic programming to calculate lateral error, heading error, and algorithm execution time at different speeds of 3 m/s, 7 m/s, and 10 m/s in order to examine the performance parameters and suitable surroundings of intelligent vehicle lateral control algorithms. At low speeds, vehicles need precise control and as the speed increases, other factors such as lateral stability are required. However, most of the existing work looks at only one speed, so it does not provide quantitative comparisons between different operating conditions. Although model predictive control (MPC) achieves high tracking accuracy, its computational cost is highly variable, whereas linear quadratic regulation (LQR) provides strong real-time performance at the expense of larger heading errors, and pure pursuit (PP) exhibits poor low-speed accuracy but a moderate increase in computation time at high speed; therefore, quantitative analysis is required to delineate the operating scenarios for which each algorithm is most suitable.

These results indicate that MPC and LQR are preferable when high tracking accuracy and stability are required, whereas PP is more suitable for applications that prioritize real-time performance and computational efficiency across varying speeds.

Keywords: model predictive control algorithm; pure pursuit algorithm; linear quadratic regulator algorithm; lateral error; heading error; algorithm execution time

1. Introduction

With the fast growth of autonomous driving, vehicle lateral control algorithms are now essential for ensuring these systems operate safely and reliably. Their main task is to keep the vehicle accurately following the planned path, whether driving in cities, on highways, or through complex environments. The autonomous vehicle market is booming, expected to grow from \$76 billion in 2021 to \$400 billion by 2030, with advanced technologies deployed widely in transportation and logistics. By 2035, 33 million autonomous vehicles are anticipated on the road. However, traditional control methods struggle to meet the demands of such dynamic conditions. Research shows that lateral deviations cause 34% of accidents, with risks increasing sharply when lateral error or heading deviations exceed certain limits. Therefore, developing precise lateral control algorithms like MPC and LQR for functions such as lane keeping and emergency obstacle avoidance is critical to improving autonomous driving safety. These findings underline that high-accuracy lateral control, tightly coupled with vehicle dynamics, is a critical direction for advancing safe and stable intelligent driving systems and continues to be a major focus in recent research.

The main lateral control methods for vehicles include Model Predictive Control (MPC), Pure Pursuit (PP), Linear Quadratic Regulator (LQR), and Stanley algorithms. Zhu Xingjian et al. [1] developed a weighted adaptive MPC system, demonstrating its reliability and effectiveness in real-world unmanned vehicle applications. Zhiheng et al. [2]enhanced traditional MPC and tracking algorithms by incorporating dynamic constraints like centroid side slip angle, acceleration, and tire lateral deflection, establishing a layered trajectory planning and tracking control framework that improves tracking performance under challenging conditions and external disturbances. Sun Yinjian [3] utilized a 6-degree-of-freedom vehicle dynamic model combined with a magic formula tire model and MPC with soft lateral deflection angle constraints to design a trajectory tracking controller that actively controls the front wheels, balancing trajectory accuracy with vehicle stability. V Changoski [4] analyzed future autonomous vehicles' steer-by-wire systems using MPC controllers to optimize vehicle behavior. Zhu Zhongwen [5] addressed trajectory tracking accuracy and stability issues for autonomous vehicles on complex paths by developing a nonlinear MPC strategy that enhances robustness across varying speeds. These studies collectively highlight the advancements in lateral control strategies aiming to improve tracking precision, robustness, and vehicle stability in autonomous driving systems. Xie X.Y. et al. [6] addressed the long computation times and real-time challenges of MPC by proposing a model predictive trajectory tracking method with variable control time step sizes, which reduces calculation time by adjusting the step size dynamically. Kayacan E et al. [7] improved control accuracy and stability during disturbances by integrating a linear MPC with feedforward and robust control actions. Liu Yifan et al. [8]developed a pure tracking controller that dynamically adjusts the look-ahead distance using speed and lateral deviation as fuzzy control inputs, overcoming the limitations of fixed look-ahead distances in traditional pure tracking models. Liu Weidong et al. [9] introduced an enhanced pure tracking path controller combining planned paths and speed data, validating its effectiveness experimentally on real vehicles with improved tracking precision and steering smoothness. Meng Wang et al. [10] proposed a path-tracking strategy based on improved pure tracking that adaptively modifies the forward distance using vehicle speed and road curvature, reducing large tracking errors in curves. Chen Ridong [11] enhanced the Pure Pursuit algorithm with steering angle compensation through two-point preview, and developed high-speed and low-speed controllers incorporating preview functions, steering constraints, and speed control based on track curvature. Ni et al. [12] optimized the LQR controller with a rolling optimization strategy based on lateral tracking error, enhancing its tracking accuracy by modeling robot motion via the Lagrangian method. These contributions collectively advance lateral control algorithms by improving computational efficiency, robustness, adaptability, and accuracy in autonomous vehicle trajectory tracking. Trapanese et al. [13] enhanced traditional LQR controllers by integrating feedforward control, markedly improving lateral tracking accuracy. Gao Linlin et al. [14] proposed an improved LQR method incorporating feedforward control, which enhanced both adaptability and control precision. To improve lateral stability and tracking accuracy under high-speed and high-curvature conditions, Chen Liang et al. [15] developed an intelligent vehicle LQR controller based on optimizing front wheel lateral force, effectively reducing path tracking errors while enhancing stability. Lu Yang et al. [16] tackled low accuracy and stability issues in LQR trajectory tracking by introducing a lateral control method optimized through a genetic algorithm for weight matrix selection. Zhang Yajuan et al. [17] presented an Adaptive LQR (ALQR) controller that adapts in real time to varying road adhesion conditions by updating tire cornering stiffness, significantly boosting tracking performance compared to conventional LQR, especially on low-adhesion surfaces. Collectively, these advancements improve LQR-based lateral control through feedforward integration, parameter optimization, and adaptive strategies to ensure higher accuracy and vehicle stability in autonomous driving.

Most studies on autonomous vehicle lateral control algorithms like MPC, PP, and LQR focus on single fixed speeds, lacking comprehensive comparisons across a range of speeds (3 to 10 m/s). They often emphasize lateral errors while overlooking the balance between heading errors and computation time, which is crucial for precise, real-time control. Additionally, evaluations tend to be qualitative, rather than quantitative metrics such as standard deviation, peak error, or computational growth rate, limiting the ability to make clear, data-driven decisions on algorithm selection.

2. Vehicle Kinematics Model

Controlling a vehicle's motion starts with creating a model based on data, and one common choice is the two-degree-of-freedom prediction model, which gives a simple yet effective way to describe lateral motion. This kinematic model as shown in the Figure 1 is used here to develop the vehicle's lateral control. In addition to this, there are other models, such as dynamic models, that consider additional vehicle behaviors such as velocity, yaw, and tire forces. These dynamic models are great for high-speed and complex driving situations, but require a lot of computation and careful tuning. Data-driven models, which use machine learning methods like neural networks, can learn the vehicle's behavior from data and adapt to tricky conditions such as icy roads or worn tires. However, they need sufficient quality data and might struggle with real-time use due to their size and complexity. Geometric path-tracking models, like the Pure Pursuit algorithm, use simple geometric rules to calculate steering commands. They are fast and work well at low speeds with gentle curves, such as in warehouses or shuttle buses. Lastly, sliding mode control offers strong robustness against uncertainties and disturbances by forcing the system to follow a designed trajectory, but it can cause high-frequency oscillations that require filtering, adding complexity. This variety of models provides options for balancing accuracy, robustness, computational complexity, and suitability for different driving conditions.

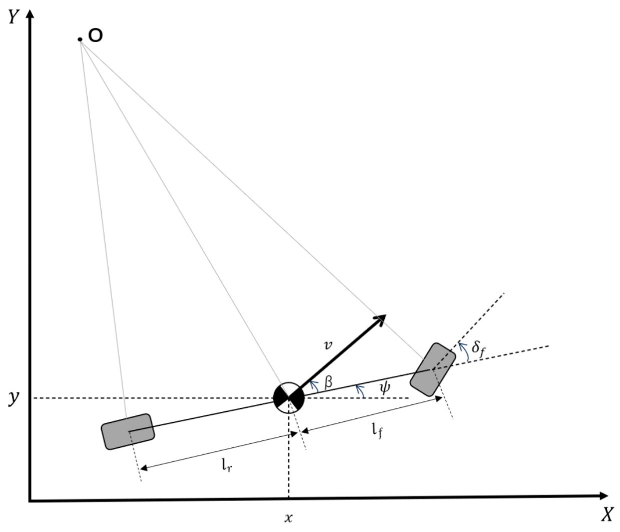

Figure 1. Two-degree-of-freedom prediction model

The degree-of-freedom prediction model is widely used in vehicle lateral control because it offers several advantages. First, it is highly efficient computationally by simplifying vehicle dynamics to key motions like lateral displacement and yaw angle, which reduces model complexity. Second, it strikes a good balance between accuracy and practicality, effectively representing vehicle behavior in low to medium speed scenarios such as city driving, even though it omits some dynamic details. Third, its foundation on rigid body kinematics equations means it has a straightforward mathematical form that is easy to linearize, while its parameters have clear physical meaning, making calibration and validation simpler and less costly. Lastly, this model fits well with mainstream control methods like MPC and LQR because its low dimensionality allows direct incorporation into control frameworks without causing optimization difficulties or computational delays seen with complex models.

Where v: The speed of the center of mass of the vehicle, which represents the actual speed of the vehicle during driving.

ψ: yaw angle (the angle between the longitudinal axis of the vehicle and the X-axis), representing the direction of the vehicle.

β: lateral deflection angle of the centroid (the angle between the longitudinal axis and the speed of the vehicle).

δ_f: Front wheel angle.

When establishing a vehicle kinematic model, the assumptions made are [18]:

1. By ignoring vertical movement (the Z-axis), the vehicle's motion is simplified to a two-dimensional plane.

2. If the steering angle is very small and both front or both rear wheels turn at the same rate, the wheels can be treated as a single unit; this lets us reduce the four-wheel system to a two-wheel model.

3. At low speeds, lateral tire forces are negligible, so the direction of the front wheels can be assumed to match the vehicle's velocity, which further simplifies the control model. The passage in the image provides assumptions and descriptions for a kinematic vehicle model:

4. If the vehicle's speed changes gradually—with acceleration kept low—the load transfer between front and rear axles can be ignored.

5. If the entire vehicle and suspension are considered rigid, and tires do not slip, the yaw angle can represent the actual movement direction.

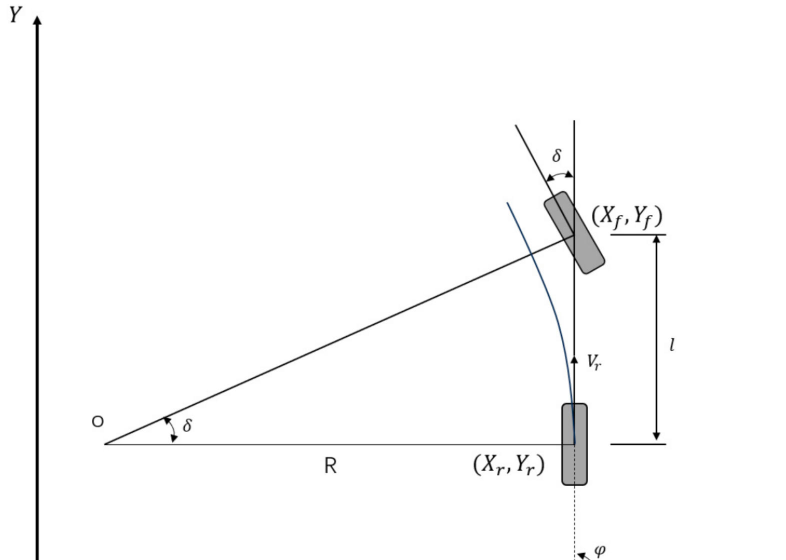

Figure 2. Vehicle motion model

Figure 2's kinematic model is constructed based on certain assumptions outlined in Figure

1. It defines the positions of the vehicle's front wheels (X_f, Y_f) and rear wheels (X_r, Y_r), along with core variables like the steering angle δ, wheelbase l, and the radius of the driving path R . Two distinct coordinate systems are commonly used: the inertial (global) coordinate system, labeled XOY, which serves as a reference for navigation systems, and the vehicle body coordinate system, which is aligned with the vehicle's forward direction and is primarily used to describe its relative motion.

In Figure 2, the vehicle body coordinate system takes the forward direction as its reference, while the inertial system provides a general spatial reference. The relationship between these two is governed by the angle φ, which measures their orientation difference and is defined as negative in the clockwise direction. This approach helps to establish a clear mathematical framework for modeling and analyzing vehicle motion, ensuring both precision and consistency when interpreting the movement and position of the vehicle in different spatial contexts.

Based on Figure 2, the kinematic model for vehicle motion is described as follows. The derivatives of the rear wheel position and the vehicle's orientation angle are given by:

$$\begin{bmatrix} \dot{X}_r \\ \dot{Y}_r \\ \dot{\varphi} \end{bmatrix} = \begin{bmatrix} \cos\varphi \\ \sin\varphi \\ \tan\delta/l \end{bmatrix} V_r \tag{1}$$

$\dot{X}_r = V_r \cos\varphi$ describes forward movement along the inertial X-axis.

$\dot{Y}_r = V_r \sin\varphi$ describes movement along the inertial Y-axis.

$\dot{\varphi} = V_r \tan\delta/l$ gives the rate of change of heading, which increases as steering angle and speed increase, or as wheelbase decreases.

In this form, the two control inputs are vehicle speed V_r and angular velocity r:

V_r term: Accounts for motion along the vehicle's heading φ.

$\omega_r$term: Accounts for motion around the vehicle's center (rotation)

$$\begin{bmatrix} \dot{X}_r \\ \dot{Y}_r \\ \dot{\varphi} \end{bmatrix} = \begin{bmatrix} \cos\varphi \\ \sin\varphi \\ 0 \end{bmatrix} V_r + \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \omega_r \tag{2}$$

The state vector l is $[X_r, Y_r, \varphi]^T$, the control input K is $[V_r, \delta]$, and the angular velocity is defined as $\omega_r = \frac{V_r}{l} \tan\delta$.

The yaw rate increases with both speed and steering angle, and decreases as the wheelbase increases.

Small steering angles (δ) can be linearized as $\tan\delta \approx \delta$.

3. Horizontal Control Algorithm

3.1 Model Predictive Control (MPC)

Model Predictive Control (MPC) achieves multi-objective and constrained control through rolling time-domain optimization. It can manage multivariable systems and explicitly handle physical limits, such as steering constraints, making it effective for complex path tracking. With an adaptable prediction model, MPC responds to changing vehicle dynamics and disturbances like crosswinds. By predicting multi-step trajectories, it ensures smoother steering and improved ride comfort. MPC performs well in highway lane-keeping [19] and emergency obstacle avoidance, especially when combined with the Magic Formula tire model to handle nonlinear tire forces. Its core process includes three stages: model prediction, rolling optimization, and feedback correction. At each time step k, as shown in Figure 3, the reference trajectory is represented by curve 1. The controller predicts the system's output (curve 2) over a future horizon $[k, k+p]$ using the current state, control inputs, and a prediction model. By solving an optimization problem, a sequence of control actions (curve 4) is generated. The first value in this sequence is applied at the current step, and the process repeats at each time step for continuous and effective control. Given the nonlinear nature of the vehicle's kinematics, direct computation is difficult. To enable efficient computation, the system is linearized. The state vector is defined as $\tilde{\imath} = [\tilde{X}_r\ \tilde{Y}_r\ \tilde{\varphi}]^T$ and the control vector as $\tilde{\kappa} = [\tilde{V}_r\ \tilde{\delta}]^T$. The system's nonlinear dynamics are first linearized around an operating point to simplify control design using a Taylor series expansion.

$$\dot{\tilde{l}} = A\tilde{x} + B\tilde{\kappa} \tag{3}$$

This is the standard continuous-time state-space representation, where $\tilde{x}$ is the state vector (e.g., position, heading), $\tilde{u}$ is the control input vector (e.g., steering angle, acceleration), and $\tilde{A}$ and $\tilde{B}$ are the state and input matrices containing linearized system parameters.

$$A = \begin{bmatrix} 0 & 0 & -V_r \sin\theta_r \\ 0 & 0 & V_r \cos\theta_r \\ 0 & 0 & 0 \end{bmatrix}, \quad B = \begin{bmatrix} \cos\theta_r & 0 \\ \sin\theta_r & 0 \\ \tan\delta/l & V_r/l\cos^2\delta_r \end{bmatrix}.$$

For digital control, the continuous model is converted to a discrete-time representation using a sampling period T.

$$\tilde{l}(k+1) = A_{k,t}\tilde{l}(k) + B_{k,t}\tilde{\kappa}(k) \tag{4}$$

It allows us to calculate the state $\tilde{x}$ at the next time step $k+1$ based on the current state $\tilde{x}(k)$ and the current control input $\tilde{u}(k)$.

$$A_{k,t} = \begin{bmatrix} 1 & 0 & -V_r T \sin\varphi_r \\ 0 & 1 & V_r T \cos\varphi_r \\ 0 & 0 & 1 \end{bmatrix}, \quad B_{k,t} = \begin{bmatrix} T\cos\varphi_r & 0 \\ T\sin\varphi_r & 0 \\ T\tan\delta_r/l & TV_r/l\cos^2\delta_r \end{bmatrix},$$

$\tilde{l}(k)$ is the state quantity at the current sampling moment, $\tilde{\kappa}(k)$ is the current control quantity, $\tilde{l}(k+1)$ is the state quantity at the next sampling moment, and T is the sampling period.

MPC often optimizes the control increment $\Delta u(k)$ instead of the control amount $u(k)$. This requires introducing an augmented state vector.

The new augmented state $\tilde{\xi}(k)$ combines the current state $\tilde{x}(k)$ and the control input from the previous step $u(k-1)$:

$$\tilde{\xi}(k) = \begin{bmatrix} \tilde{x}(k) \\ u(k-1) \end{bmatrix}$$

By including $u(k-1)$, the system dynamics can be reformulated to treat $\Delta u(k) = u(k) - u(k-1)$ as the input variable.

The evolution of the augmented state is described by the control increment $\Delta u(k)$.

$$\tilde{\xi}(k+1) = \tilde{A}_{\xi}\tilde{\xi}(k) + \tilde{B}_{\xi}\Delta u(k) \tag{5}$$

This is the fundamental discrete state equation used for prediction. The control increment $\Delta u(k)$ becomes the input to the augmented system.

The measured output $\eta(k)$ is a linear combination of the augmented state.

$$\eta(k) = \tilde{C}_{\xi}\tilde{\xi}(k) \tag{6}$$

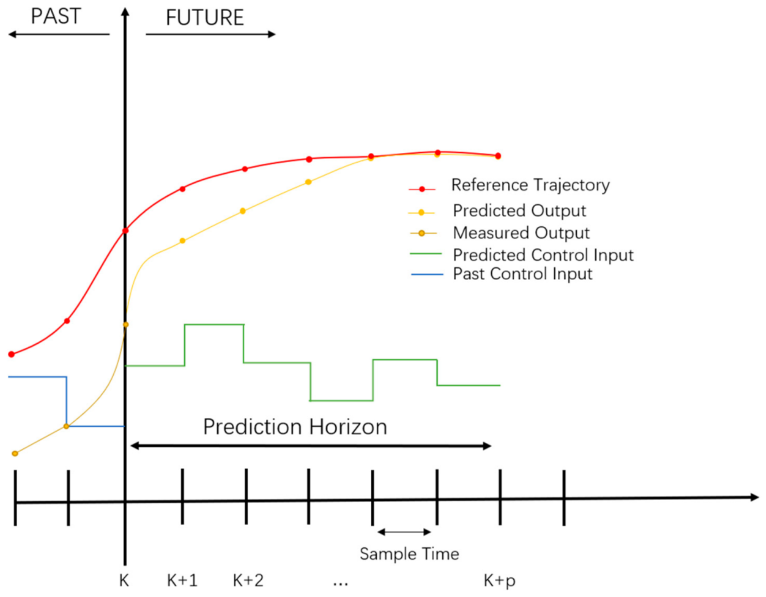

Figure 3. MPC Schematic Diagram

This relates the internal state (which includes $\tilde{x}$) to the externally measurable quantities (e.g., position error, velocity).

where

$$\tilde{A}_k = \begin{bmatrix} A_k & B_k \\ 0 & I_m \end{bmatrix}, \quad \tilde{B}_k = \begin{bmatrix} B_k \\ I_m \end{bmatrix}, \quad \tilde{C}_k = \begin{bmatrix} C_k & 0 \end{bmatrix}$$

and I_m represents the one-dimensional and two-dimensional identity matrices.

MPC predicts the system's output over a Prediction Horizon N_p. This is done by recursively applying the augmented state equation. According to (5) and (6) state prediction, we can obtain these equations to express the future outputs as a linear function of the known current state and the unknown future control increments.

$$\zeta(k+1) = \tilde{A}_k \tilde{\xi}(k) + \tilde{B}_k \Delta u(k)$$

$$\zeta(k+2) = \tilde{A}_k \zeta(k+1) + \tilde{B}_k \Delta u(k+1) = \tilde{A}_k^2 \tilde{\xi}(k) + \tilde{A}_k \tilde{B}_k \Delta u(k) + \tilde{B}_k \Delta u(k+1)$$

$$\zeta(k+N_p) = \tilde{A}_k^{N_p} \tilde{\xi}(k) + \tilde{A}_k^{N_p-1} \tilde{B}_k \Delta u(k) + \cdots + \tilde{B}_k \Delta u(k+N_p-1) \tag{7}$$

According to (7), the system output of the new state space equation can be calculated as follows:

$$\eta(k+1) = \tilde{C}_k \tilde{\xi}(k+1) = \tilde{C}_k \tilde{A}_k \tilde{\xi}(k) + \tilde{C}_k \tilde{B}_k \Delta u(k)$$

$$\eta(k+2) = \tilde{C}_k \tilde{A}_k^2 \tilde{\xi}(k) + \tilde{C}_k \tilde{A}_k \tilde{B}_k \Delta u(k) + \tilde{C}_k \tilde{B}_k \Delta u(k+1)$$

$$\eta(k+N_p) = \tilde{C}_k \tilde{A}_k^{N_p} \tilde{\xi}(k) + \tilde{C}_k \tilde{A}_k^{N_p-1} \tilde{B}_k \Delta u(k) + \cdots + \tilde{C}_k \tilde{B}_k \Delta u(k+N_p-1) \tag{8}$$

All predicted outputs Y over N_p are written in a single matrix equation:

$$Y = \psi \tilde{\xi}(k) + \Theta \Delta U \tag{9}$$

- Y: Vector of predicted outputs $\eta(k+1)$ to $\eta(k+N_p)$.

- $\tilde{\xi}(k)$: The current augmented state (known).
- ΔU: The sequence of control increments to be optimized (unknown).
- Ψ and Θ: Time-invariant matrices derived from $\tilde{A}_{\xi}$, $\tilde{B}_{\xi}$, and $\tilde{C}_{\xi}$.

where

$$Y = \begin{bmatrix} \eta(k+1) \\ \eta(k+2) \\ \vdots \\ \eta(k+N_c) \\ \vdots \\ \eta(K+N_p) \end{bmatrix}, \quad \psi = \begin{bmatrix} \tilde{C}_k\tilde{A}_k \\ \tilde{C}_k\tilde{A}_k^2 \\ \vdots \\ \tilde{C}_k\tilde{A}_k^{N_c} \\ \vdots \\ \tilde{C}_k\tilde{A}_k^{N_p} \end{bmatrix},$$

$$\Theta = \begin{bmatrix} \tilde{C}_k\tilde{B}_k & 0 & 0 & 0 \\ \tilde{C}_k\tilde{A}_k\tilde{B}_k & \tilde{C}_k\tilde{B}_k & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots \\ \tilde{C}_k\tilde{A}_k^{N_c-1}\tilde{B}_k & \tilde{C}_k\tilde{A}_k^{N_c-2}\tilde{B}_k & \cdots & \tilde{C}_k\tilde{B}_k \\ \vdots & & & \vdots \\ \tilde{C}_k\tilde{A}_k^{N_p-1}\tilde{B}_k & \tilde{C}_k\tilde{A}_k^{N_p-2}\tilde{B}_k & \cdots & \tilde{C}_k\tilde{A}_k^{N_p-N_c}\tilde{B}_k \end{bmatrix}, \quad \Delta U = \begin{bmatrix} \Delta u(k) \\ \Delta u(k+1) \\ \vdots \\ \Delta u(k+N_c) \end{bmatrix}$$

Define the reference output vector $Y_{ref}(k) = [\eta_{ref}(k+1), \ldots, \eta_{ref}(k+N_p)]^T$, let $E = \psi\tilde{\xi}(k)$. The goal is to minimize a cost function J that penalizes tracking error and control effort. J penalizes the difference between predicted output Y and reference Y_{ref} (tracking error) and the control increment ΔU (control effort). Substitute (9) into available

$$J = \Delta U^T(\Theta^T Q_Q\Theta+R_R)\Delta U+2(E^T Q_Q\Theta-Y_{ref}^T Q_Q\Theta)\Delta U+E^T Q_Q E+Y_{ref}^T Q_Q Y_{ref}-2Y_{ref}^T Q_Q E \tag{10}$$

- Q: Weighting matrix for the output error (penalizes deviations from Y_{ref}).
- R_k: Weighting matrix for the control increments (penalizes aggressive control action).
- E: Terms that do not depend on the optimization variable ΔU.

where $Q_Q = I_{N_p} \otimes Q, R_R = I_{N_p} \otimes R, \otimes$ represents the Kronecker product. The part after the second term in the formula is a constant and can be ignored during optimization and solution, so the performance evaluation function can be written as

$$J = \Delta U^T(\Theta^T Q_Q\Theta + R_R)\Delta U + 2(E^T Q_Q\Theta - Y_{ref}^T Q_Q\Theta)\Delta U \tag{11}$$

Let $H = \Theta^T Q_Q\Theta + R_R$, $g = \Theta^T Q_Q(E - Y_{ref})$. Then, Formula (11) can be rewritten as (12)

$$J = 2\left(\frac{1}{2}\Delta U^T H\Delta U + g^T\Delta U\right) \tag{12}$$

In optimization solution, Equation (12) is equivalent to

$$J = \frac{1}{2}\Delta U^T H\Delta U + g^T\Delta U \tag{13}$$

This is the objective function for the optimization solver.

- H (Hessian Matrix): This matrix must be positive definite for a unique minimum.
- g^T (Linear Term): This term captures the influence of the initial state and reference on the cost.

The control amount $u(k)$ is simply the previous amount plus the calculated increment. considering that there is the following relationship between the control amount and the control increment:

$$u(k+i) = u(k+i-1) + \Delta u(k) \tag{14}$$

This relationship is used to express the constraints on the absolute control amount u in terms of the variable ΔU.

The optimization must respect physical limits on both the control amount and the rate of change (increment). Before optimization, the variables in the constraints need to be unified, and the increments of the control quantity and the control quantity can be converted into the following form:

$$U_{\min} \le A_k \Delta U + U_t \le U_{\max} \tag{15}$$

$$U_{\min} \le A_k \Delta U + U_t \le U_{\max} \tag{16}$$

$$\Delta U_{\min} \le \Delta U_t \le \Delta U_{\max} \tag{17}$$

Let $U_t = 1_N \otimes u(k–1)$, where $1_N \in \mathbb{R}^{N_c}$ is a column vector of ones, and $u(k–1)$ is the control input at the previous time step. The control input is subject to bounds $U_{\min} \le U_t \le U_{\max}$, and the control increments are constrained by $\Delta U_{\min} \le \Delta U_t \le \Delta U_{\max}$.

$$A_k = \begin{bmatrix} 1 & 0 & \cdots & \cdots & 0 \\ 1 & 1 & 0 & \cdots & 0 \\ 1 & 1 & 1 & \ddots & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 1 & 1 & \cdots & \cdots & 1 \end{bmatrix}_{N_c \times N_c}$$

At this point, the optimization problem-solving method of model prediction control can be transformed into a standard quadratic planning problem:

$$\begin{gathered} \min_{\Delta U} \frac{1}{2} \Delta U^T H \Delta U + g^T \Delta U \\ \Delta U_{\min} \le \Delta U \le \Delta U_{\max} \\ U_{\min} \le A_k \Delta U + U_t \le U_{\max} \end{gathered} \tag{18}$$

The QP solver (such as MATLAB's quadprog) is used to obtain the optimal control input increments. The solution to the QP yields the optimal sequence of control increments over the Control Horizon N_c:

$$\Delta U^* = [\Delta u^*(k), \Delta u^*(k+1), \cdots, \Delta u^*(N_c - 1)]^T \tag{19}$$

According to the receding horizon principle, only the first element of $\Delta u^*(k)$ is applied to the system.

$$u(k) = u(k-1) + \Delta u^*(k) \tag{20}$$

The calculated input $u(k)$ is executed. At the next time step, the new state $\tilde{\zeta}(k+1)$ is measured, and the entire prediction and optimization cycle is repeated.

3.2 *Pure Pursuit Algorithm (PP)*

The Pure Pursuit (PP) algorithm directs the vehicle to follow a target point ahead using geometric relationships to calculate steering. Its simplicity and efficiency make it ideal for low-speed, low-complexity tasks (like warehouse AGVs and park vehicles) and as a backup control for advanced methods like MPC or LQR. PP is easy to implement since it doesn't require a dynamic model, but its performance decreases at high speeds [20] . In Figure 4, l_d is the distance from the rear axle to the target, and α is the angular deviation.

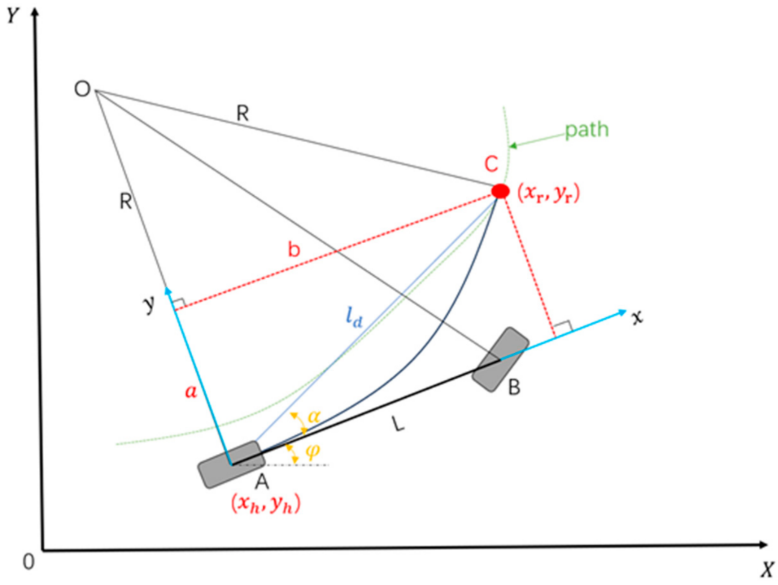

Figure 4. PP geometric model diagram

The vehicle is assumed to move along a circular arc with radius R, originating from the Instantaneous Center of Curvature (ICC), labeled O in the figure. The triangle $\triangle OAC$ is formed by the ICC and the two points on the path (A and C).

By using the Law of Sines on the triangle $\triangle OAC$ (as stated in the figure 4) to relate the lookahead distance l_d to the radius R:

$$\frac{l_d}{\sin(2\alpha)} = \frac{R}{\sin\left(\frac{\pi}{2} - \alpha\right)}$$

Since $\sin\left(\frac{\pi}{2} - \alpha\right) = \cos(\alpha)$ and $\sin(2\alpha) = 2\sin(\alpha)\cos(\alpha)$, the equation simplifies to:

$$\frac{l_d}{2\sin(\alpha)\cos(\alpha)} = \frac{R}{\cos(\alpha)}$$

By canceling $\cos(\alpha)$ and rearranging to solve for the curvature $k = 1/R$:

$$k = \frac{1}{R} = \frac{2\sin\alpha}{l_d} \tag{21}$$

To convert the required curvature (k) into a physical steering angle (δ), using the geometric relationship derived from the Ackermann steering model (the kinematic Bicycle Model):

$$\tan(\delta) = kL$$

Substituting the derived expression for k into the steering angle equation:

$$\tan(\delta) = \left(\frac{2\sin\alpha}{l_d}\right)L$$

Solving for the steering angle $\delta(t)$ results in the final Pure Pursuit control law

$$\delta(t) = \arctan\left(\frac{2L\sin(\alpha)}{l_d}\right) \tag{22}$$

3.3 Linear Quadratic Regular (LQR)

The Linear Quadratic Regulator (LQR) is an optimal control algorithm that minimizes a quadratic cost function for efficient real-time control. Solving the Riccati equation offline gives fast and stable computation, making LQR well-suited for embedded vehicle systems. LQR is robust to model errors and widely used for medium- and low-speed scenarios but is limited by its reliance on linear models [20].

To introduce LQR, consider the Linear Quadratic (LQ) Optimal Control problem for a linear time-varying system:

$$\begin{aligned} \dot{x} &= Ax + Bu \\ y &= Cx + Du \end{aligned} \tag{23}$$

Its goal is to design a control law that allows a dynamic system to operate as efficiently as possible while minimizing a specific measure of cost. This cost is often associated with how much effort the controller exerts and how far the system's states deviate from desired values over time. In practice, this cost is expressed mathematically using a quadratic functional, which means it penalizes both large deviations in system states and large control efforts in a smooth and mathematically tractable way. When both the system dynamics and the cost function exhibit linear and quadratic characteristics respectively, the problem is classified as a linear quadratic (LQ) problem. In this setting, the system being controlled follows linear dynamics—meaning its behavior can be expressed as a set of linear differential or difference equations—while the cost function is quadratic in both the system states and the control inputs. This combination is special because it leads to an analytically solvable optimization problem with elegant mathematical properties. The optimal solution to the LQ problem yields what is known as the Linear Quadratic Regulator, or LQR. The LQR controller determines the best possible control input as a linear combination of the system's current states. It does this through a mechanism called full-state feedback, where every measurable state variable of the system contributes to the control decision. This means that the controller has complete information about the system's internal conditions at any given time and uses that information to calculate the control effort needed to minimize the overall cost.

The design procedure for an LQR controller involves defining a cost function that imposes penalties on undesirable system behavior, such as large deviations from a desired trajectory or excessive actuator activity. In the Linear Quadratic Regulator (LQR) design, the matrices Q and R are the designer's primary tools for shaping the controller's performance. They are positive-semidefinite symmetric diagonal matrices ($Q \geq 0, R > 0$)) that directly define the cost function:

$$J = \int_0^\infty \left(x^T Q x + u^T R u\right) dt \tag{24}$$

Control Effort Cost (R Matrix)

$$\text{Input Cost} = u^T Ru = \sum_j R_{jj} u_j^2$$

State Deviation Cost (Q Matrix)

$$\text{State Cost} = x^T Qx = \sum_i Q_{ii} x_i^2$$

where x represents the state vector, u is the control input, Q is a weighting matrix that determines how much importance is given to the system states, and R is another weighting matrix that sets the cost of applying control inputs. By solving this optimization problem, one can compute a matrix of optimal feedback gains, denoted K, such that the control law $u = -Kx$. The core action is substituting the control law $u = -Kx$ into the input cost term $u^T Ru$ of the integral:

$$u^T Ru \to (-Kx)^T R(-Kx)$$

Recall the property of transposes: $(AB)^T = B^T A^T$. Applying this to the first term,

$$(-Kx)^T = x^T(-K)^T = x^T(-K^T)$$

Since K is a real matrix, the negative sign commutes with the transpose. Now substitute this back into the expression:

$$(-Kx)^T R(-Kx) = (x^T(-K^T))R(-Kx)$$

$$(x^T(-K^T))R(-Kx) = x^T K^T RKx$$

Now, substitute this simplified input cost term back into the total cost integral J:

$$J = \int_0^\infty \left(x^T Qx + x^T K^T RKx\right) dt$$

$$J = \int_0^\infty x^T \left(Q + K^T RK\right) x \, dt \tag{25}$$

Let's assume that the original integral form of the cost function, J can be equated to the evaluation of a quadratic form $x^T Px$ at the time boundaries (infinity and zero):

$$-x^T Px\Big|_{t=\infty}^{t=0} = \int_0^\infty x^T(Q + K^T RK)x dt \tag{26}$$

The matrix P is thus the matrix that captures the total future cost, given the state $x(t)$.

The fundamental theorem of calculus states that the integral of a derivative is the function itself evaluated at the boundaries:

$$\int_0^\infty \frac{d}{dt} f(t)\, dt = f(\infty) - f(0)$$

In LQR, we define $f(t) = -x^T Px$. Therefore, we can write:

$$\int_0^\infty \frac{d}{dt}(-x^T Px) dt = (-x^T Px)\Big|_{t=\infty}^{t=0}$$

$$\int_0^\infty \frac{d}{dt}(-x^T Px)dt = \int_0^\infty x^T(Q + K^T RK)xdt$$

This equality implies that the integrand of the derivative on the LHS must be equal to the integrand on the RHS for all time t:

$$\frac{d}{dt}(-x^T Px) = x^T(Q + K^T RK)x$$

This is the key relationship established by introducing P: the rate of change of the cost function (LHS) must equal the instantaneous cost (RHS).

Using the product rule for matrix derivatives (and noting P is constant in time for the infinite-time LQR):

$$\frac{d}{dt}\left(-x^T Px\right) = -\left(\dot{x}^T Px + x^T P\dot{x}\right) \tag{27}$$

Since $u = -Kx$, we know that

$$\dot{x} = (Ax + Bu) = Ax + B(-Kx) = (A - BK)x \tag{28}$$

Substituting $\dot{x} = (A - BK)x$ and its transpose $\dot{x}^T = x^T(A - BK)^T$:

$$\frac{d}{dt}(-x^T Px) = -\left[x^T(A - BK)^T Px + x^T P(A - BK)x\right]$$

Since x^T and x are on the outside of all terms, we can factor them out:

$$\frac{d}{dt}(-x^T Px) = x^T[-(A - BK)^T P - P(A - BK)]x$$

Now, we equate this result (LHS) to the simplified cost integrand (RHS), $x^T(Q+K^T RK)x$:

$$x^T[-(A - BK)^T P - P(A - BK)]x = x^T(Q + K^T RK)x \tag{29}$$

Since this equality must hold for any state $x \neq 0$, the quadratic form matrices themselves must be equal:

$$-(A - BK)^T P - P(A - BK) = Q + K^T RK$$

Rearranging the terms to set the equation to zero:

$$(A - BK)^T P + P(A - BK) + Q + K^T RK = 0$$

This matrix equation is known as the Closed-Loop Lyapunov Equation.

$$x^T[-(A - BK)^T P - P(A - BK) - (Q + K^T RK)]x = 0$$

$$x^T[(A - BK)^T P + P(A - BK) + Q + K^T RK]x = 0$$

Substitute the definition of K into the combined term $K^T RK$:

$$K^T RK = (R^{-1}B^T P)^T R(R^{-1}B^T P)$$

Applying the transpose rule $(R^{-1}B^T P)^T = P^T(B^T)^T(R^{-1})^T$. Since P, R, R^{-1} are symmetric, $P^T = P$ and $(R^{-1})^T = R^{-1}$. Also, $(B^T)^T = B$:

$$K^T RK = PBR^{-1} RR^{-1} B^T P$$

Since $RR^{-1} = I$ (Identity matrix), the term simplifies to the cross-term:

$$K^T RK = PBR^{-1} B^T P$$

Now substitute the term $PBR^{-1}B^TP$ for K^TRK, and substitute PBK for $PBR^{-1}B^TP$ (since $K = R^{-1}B^TP$) back into the zero-sum Lyapunov equation:

$$(A - BK)^T P + P(A - BK) + Q + K^T RK = 0$$

Expanding and substituting the derived cross-term:

$$A^T P - K^T B^T P + PA - PBK + Q + PBR^{-1} B^T P = 0$$

Finally, substitute the optimal gain $K = R^{-1}B^TP$ into the two middle terms:

- $K^T B^T P = (R^{-1} B^T P)^T B^T P = PBR^{-1} B^T P$
- $PBK = PB(R^{-1} B^T P) = PBR^{-1} B^T P$

The equation becomes:

$$A^T P - PBR^{-1} B^T P + PA - PBR^{-1} B^T P + Q + PBR^{-1} B^T P = 0$$

One positive cross-term cancels one negative cross-term, leaving the Algebraic Riccati Equation (ARE):

$$A^T P + PA - PBR^{-1} B^T P + Q = 0 \tag{30}$$

The LQR implementation boils down to solving this non-linear matrix equation for the unique, positive-definite symmetric matrix P, which then determines the optimal gain $K = R^{-1}B^TP$.

4. Experimental Setup

4.1 Hardware and Software Environment

For this project, I ran all the simulations on my laptop to test how the controllers perform. The laptop is quite powerful, equipped with an AMD Ryzen 7 8845HS processor and an NVIDIA RTX 4060 Ti graphics card. It also has 32GB of RAM, which helps in running the simulator smoothly.

On the software side, I used the CARLA simulator, specifically version 0.9.16. I installed CARLA locally on the machine for better access, but it is worth noting that this setup can also work using Docker if needed. The entire code for the controllers was written in Python 3.10. To make sure the code runs, there are a few Python libraries that need to be installed first. You can install these dependencies easily using the conda environment file 'environment.yml' or by manually installing packages like numpy, carla.

4.2 Vehicle and Environment

The simulation takes place in 'Town 01' map, which is one of the standard maps provided by CARLA. The vehicle I used for all the tests is the 'lincoln mkz 2017'. This car comes equipped with several sensors, including a LiDAR, camera, radar, and an IMU. While I didn't use all of these sensors for the current control logic, they are attached to the car for future studies.

4.3 Route Generation and Logic

Instead of driving manually, the path for the car is generated automatically. First, the code randomly selects 10 points on the map (using the seed provided for random number generator, for repeatability). Then, using CARLA's Global Route Planner (GRP), a path is created to connect these points. To make sure the car has smooth targets to follow, I interpolated this route so that there is a target pose (which includes both position and orientation) every 20 centimeters. The vehicle is then spawned exactly at the first point of this generated route. To help with debugging and to see where the car is trying to go, I added visual arrows in the simulation that show the path on the screen.

4.4 Route Generation and Logic

Each experiment is set to run for exactly 5 minutes. A very important part of this setup is how time is handled. I set the simulation timestep to be 0.05 seconds. I also forced CARLA to run in "Synchronous Mode." This is really important because it ensures that the simulation waits for my control algorithm to finish calculating before moving to the next frame. This way, the performance of the car depends only on the controller's logic, not on whether the computer was lagging at that moment.

4.5 Controllers and Configuration

The car needs to control both its speed and its steering. For the speed (longitudinal control), I used a PID controller. This algorithm looks at the target speed and automatically adjusts the brake and throttle values between 0 and 1 to maintain the vehicle's speed around the target speed.

For steering (lateral control), I can switch between three different algorithms: Pure Pursuit, LQR, or MPC. For the Pure Pursuit algorithm specifically, the lookahead distance isn't fixed; it changes dynamically in the configuration file based on how fast the car is going.

I didn't want to hardcode values inside the script, so I put all the settings into a file called config.yaml. This file holds every parameter needed to run the sim, like target speeds or PID gains. When running the program, I pass both this configuration file and the name of the output file as command-line arguments.

4.6 Data Collection

As the simulation runs, the code automatically records the results into a CSV file. This file contains the Timestamp, the name of the Algorithm used, the Vehicle Speed (m/s), the Lateral Error (meters), the Heading Error (radians), and the Execution Time (ms). This allows me to plot graphs and compare how well each algorithm drove the car.

5. Code Implementation

While the complete source code and the instructions to run the simulation are provided in my GitHub repository (https://github.com/AakashDammala/Compare-PP-LQR-MPC), I want to briefly explain the main logic behind the controllers here.

5.1 Pure Pursuit Algorithm

The core idea of this code is to find a point on the path ahead of the car and calculate the steering angle needed to reach it.

First, the function needs to know where the car currently is and where it should be going. I get the vehicle's position and rotation (yaw) from the simulator. Then, I pick a "target" point from our planned route. The distance to this point is determined by the lookahead variable, which can be adjusted in the configuration file.

```
def pure_pursuit_controller(self, target_path):
        transform = self.vehicle.get_transform()
        vehicle_loc = transform.location
        vehicle_yaw = math.radians(transform.rotation.yaw)
        lookahead = self.planning_config.get("PP_lookahead", 5)
        target = target_path[min(
            self.closest_idx + lookahead, len(target_path)-1)]
        target_loc = target.location
```

Once the target point is selected, I have to solve a coordinate problem. The map uses global coordinates (like a fixed grid), but the car steers relative to itself (left or right). The code below calculates the distance to the target (dx and dy) and then uses trigonometry to rotate these coordinates. This effectively converts the target point from the map's view to the driver's view.

```
        # Vector from car to target waypoint
        dx = target_loc.x - vehicle_loc.x
        dy = target_loc.y - vehicle_loc.y

        # Transform to vehicle coordinates
        x = dx * math.cos(vehicle_yaw) + dy * math.sin(vehicle_yaw)
        y = -dx * math.sin(vehicle_yaw) + dy * math.cos(vehicle_yaw)
```

Finally, the code calculates the actual steering angle. It uses the Pure Pursuit formula, which assumes the car moves like a bicycle. It calculates the curvature needed to drive an arc that connects the car's rear axle to the target point. L represents the distance between the front and rear wheels (wheelbase).

The last step is a safety check. CARLA expects steering values between -1.0 (full left) and 1.0 (full right), so I use the max and min functions to make sure the calculated angle stays within these limits before sending it to the car.

```
        # Pure pursuit steering formula
        L = self.planning_config["vehicle_wheelbase"]
        steer = math.atan2(2 * L * y, x * x + y * y)

        # Set limits for the steering cmd generated
        steer = max(-1.0, min(1.0, steer))

        return steer
```

5.2 Linear Quadratic Regulator Algorithm

The Linear Quadratic Regulator (LQR) is a control algorithm that focuses on optimization. Instead of just reacting to errors like a simple controller, LQR tries to find the perfect mathematical balance between two competing goals: minimizing the error (staying on the path) and minimizing the control effort (not steering too crazily).

First, the code gathers the vehicle's physical parameters, like its current speed (v) and the distance between the wheels (wheelbase L). We then calculate the "State Errors" which are simply how far the car is from the path (Lateral Error) and how much its angle is wrong (Heading Error).

A key feature we added here is latency compensation. Since the simulation takes a small amount of time (0.05 seconds) to process each frame, the car moves slightly before the steering command actually happens. To prevent the car from oscillating (wobbling), we predict what the error will be in the next step and use that for our calculations.

```
# Predict state 1 step into the future to compensate for delay
# Future Lat Error = Current Lat + (v * sin(Head Error) * dt)
pred_lat_error = lat_error + (v * head_error * dt)
pred_head_error = head_error  # Heading changes slower, usually ok
     to keep

state_vector = np.array([[pred_lat_error], [pred_head_error]])
```

Next, we define the State Space matrices, A and B. These matrices are the mathematical model of the car. Matrix A describes how the errors grow naturally over time if the steering wheel is held straight, while Matrix B describes how turning the steering wheel affects those errors.

```
# 3. Define Discrete State Space Matrices A and B
# Continuous A = [[0, v], [0, 0]], B = [[0], [v/L]]
# Discrete Approximation: A_d = I + A*dt, B_d = B*dt
A = np.array([[1.0, dt * v],
              [0.0, 1.0]])
B = np.array([[0.5 * dt * dt * v * v / L],  # Effect on Lateral
    Error (2nd order)
              [dt * v / L]])                # Effect on Heading
                 Error
```

Then we define the cost matrices, Q and R. These are basically the ”tuning knobs” of the algorithm. Q sets the penalty for errors, a high value here forces the car to stick to the path tightly. R sets the penalty for steering effort, a high value here forces the car to steer gently.

```
# 4. Define Q and R Costs
# Q penalizes lateral error heavily, R penalizes steering effort
Q = np.array([[self.planning_config["Q_cost_lateral"], 0.0],
              [0.0, self.planning_config["Q_cost_heading"]]])
R = np.array([[self.planning_config["R_cost"]]])
```

We don't solve the complicated math manually. We use a linear algebra solver (la.solve_discrete_are) to solve the Riccati Equation. This gives us the optimal Gain Matrix, K. This matrix tells us exactly how much we should steer for every meter of error we detect.

```
# 5. Solve Discrete Algebraic Riccati Equation (DARE)
P = la.solve_discrete_are(A, B, Q, R)

# 6. Compute Gain K = (R + B.T*P*B)^-1 * (B.T*P*A)
K = la.inv(R + B.T @ P @ B) @ (B.T @ P @ A)
```

Finally, we calculate the steering command by multiplying our errors by the Gain K ($u = -Kx$). Since the calculated steering angle might be larger than what the car can physically do, we use np.clip to limit the final output between -1.0 (full left) and 1.0 (full right) before sending it to CARLA.

```
# 7. Compute Optimal Control u = -Kx
state_vector = np.array([[pred_lat_error], [pred_head_error]])
```

```
        steer_rad = -(K @ state_vector)[0, 0]

        # Normalize to CARLA steering [-1, 1] (Assuming max steer ~30-40
            deg, e.g. 0.6 rad)
        steer_cmd = np.clip(steer_rad / self.max_steer_rad, -1.0, 1.0)

        return steer_cmd
```

5.3 Model Predictive Control Algorithm

Instead of just looking at where the car is right now (like PIDs), MPC looks into the future. It predicts where the car will go over the next few seconds based on its current speed and steering, and then calculates the best sequence of steering moves to keep it on the path.

First, we set up the basics. We get the simulation time step (dt), the car's wheelbase (L), and its speed (v). We also define the "Prediction Horizon" (N_p), which dictates how many steps into the future the algorithm looks.

```
        # Horizons
        Np = self.planning_config["prediction_horizon"]  # Prediction
            Horizon (How far we look ahead)
        Nc = self.planning_config["control_horizon"]  # Control Horizon

        # Current State (Global Coordinates)
        trans = self.vehicle.get_transform()
        current_state = np.array([x_cur, y_cur, phi_cur])
```

Next, we set up the optimization problem using python library cvxpy. This library solves the heavy math for us. We define our variables: x (the predicted future positions of the car) and Delta_U (the changes in steering angle).

The MPC acts like a Cost Function, defined by matrices Q and R. These act like penalties. Q Matrix: Penalizes being away from the target path. A high number here means "Accuracy is most important. "R Matrix: Penalizes changing the steering wheel too fast. A high number here means "Smoothness is most important."

The most critical part of this specific implementation is the loop where we predict the future. Inside this loop, we don't just assume the road is straight. We look at the specific waypoint on the path corresponding to that future time step. We update our mathematical model (A and B matrices) based on the road's heading at that point. This allows the car to "anticipate" curves before it actually reaches them.

```
        for t in range(Np):
            # Get Reference Waypoint for step t (Look at the road ahead)
            ref_idx = min(self.closest_idx + t, len(target_path) - 1)
            ref_wp = target_path[ref_idx]

            # Linearize A_k around the REFERENCE heading
            # This helps the model "know" a turn is coming
            A_k = np.eye(3)
            A_k[0, 2] = -v * np.sin(phi_ref) * dt
            A_k[1, 2] = v * np.cos(phi_ref) * dt
```

Finally, the solver calculates the perfect plan for the next 30 steps. However, we don't use all 30. We only take the very first step (delta_u_opt), apply it to the car, and then throw the rest away. In the next split-second frame, we recalculate the whole thing again. This is why it's called Receding Horizon Control, we are constantly re-planning as we drive.

```
        if prob.status in ["optimal", "optimal_inaccurate"]:
            # Apply first control increment
            delta_u_opt = Delta_U.value[0, 0]
            steer_rad_new = (self.last_steer * 0.6) + delta_u_opt
            return steer_cmd
```

6. Experimental Results

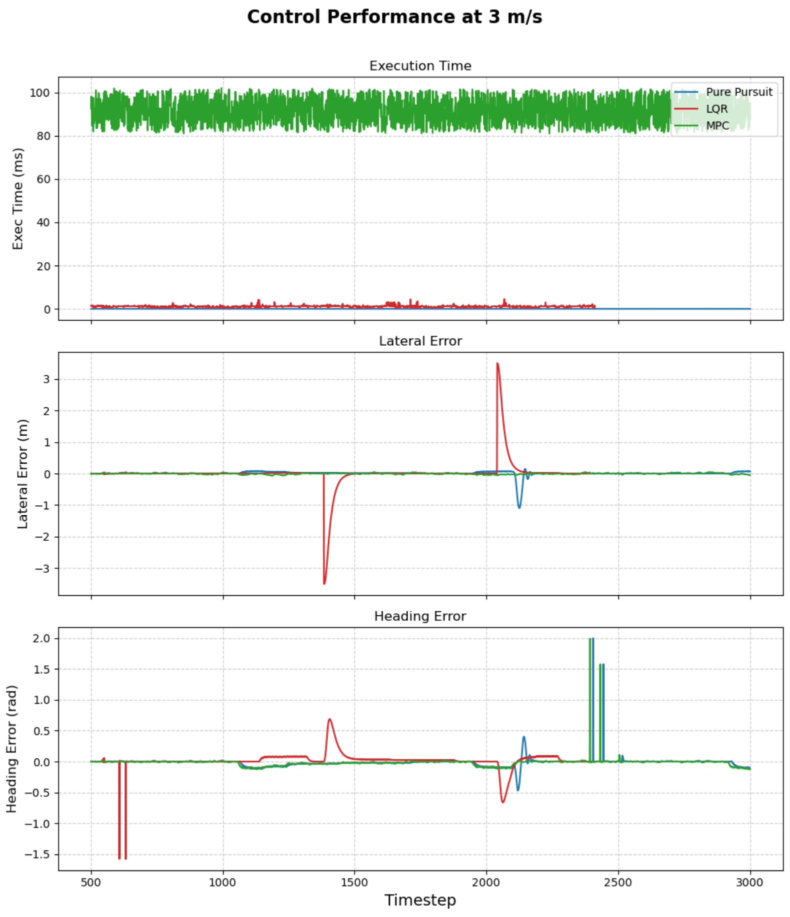

Figure 5. Comparison of Algorithm Execution Time, Lateral Error and Heading Error for 3 m/s

In this experiment, we compared three different control algorithms—Pure Pursuit, Linear Quadratic Regulator (LQR), and Model Predictive Control (MPC)—at speeds of 3 m/s, 7 m/s, and 10 m/s. We analyzed them based on how long they took to run (execution time) and how well they stayed on the path (lateral and heading error).

Computational Efficiency: When looking at execution time, the Pure Pursuit algorithm was the clear winner. As shown in the top graphs for 3 m/s and 7 m/s, Pure Pursuit is

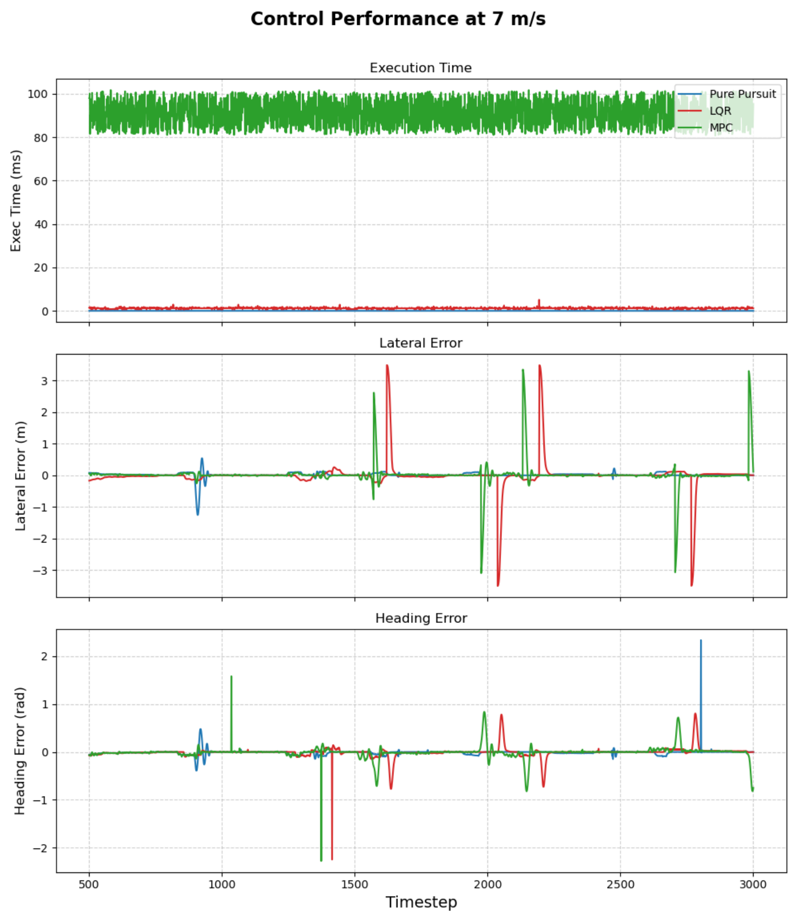

Figure 6. Comparison of Algorithm Execution Time, Lateral Error and Heading Error for 7 m/s

extremely lightweight and requires almost no time to calculate the steering angle. In contrast, LQR is bit heavier and MPC is the heaviest of all of them. It consistently took more time to run because it have to solve complex optimization matrices at every time step, whereas Pure Pursuit uses simple geometry.

Tracking Performance and Lane Changes: The lateral and heading error graphs show a distinct spike in error during the simulation. This sudden jump happens because the vehicle performs a lane change. Since the target path shifts instantly (resembling a mathematical ”step function”) and the car cannot move sideways instantly, the error naturally rises before the controller can compensate.

There is a trade-off between speed and precision. If the onboard computer has limited power, Pure Pursuit is the best choice because it is very fast. However, for high-speed driving where safety and staying exactly in the lane are critical, MPC and LQR are better choices despite being more computationally demanding.

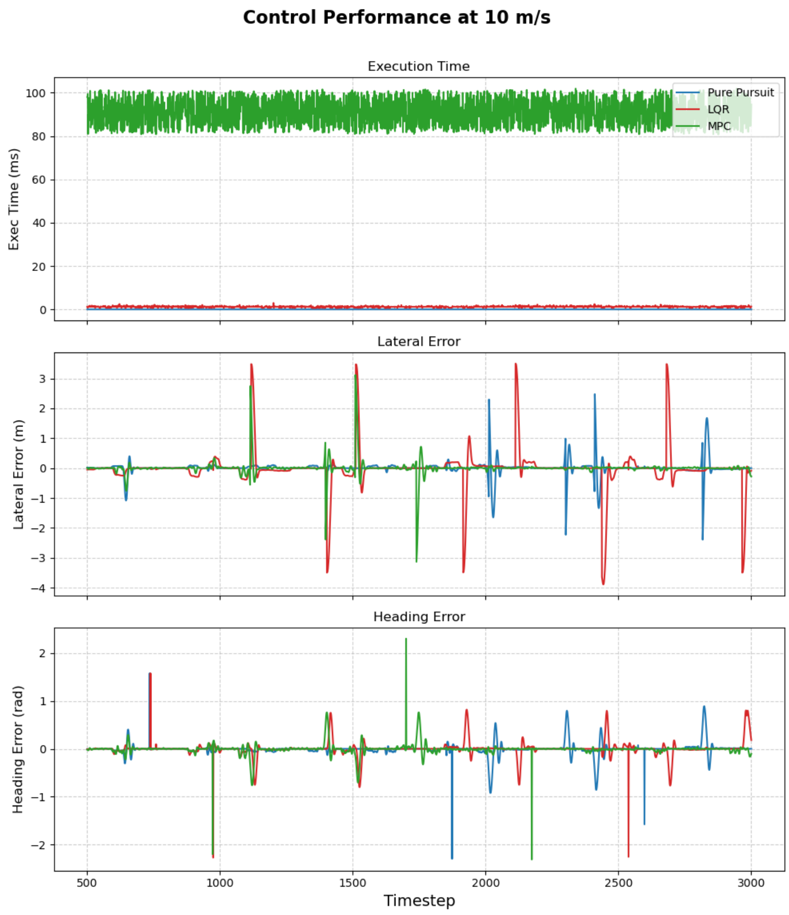

Figure 7. Comparison of Algorithm Execution Time, Lateral Error and Heading Error for 10 m/s

References

[1] X. Zhu. "Research on Trajectory Tracking Control for Autonomous Vehicles Based on Fuzzy Adaptive MPC". MA thesis. Chengdu, China: Xihua University, 2022.

[2] Z. You. "Research on Trajectory Tracking Control for Autonomous Vehicles Based on MPC Algorithm". MA thesis. Changchun, China: Jilin University, 2018.

[3] Y. Sun. "Research on Trajectory Tracking Control Algorithm for Autonomous Vehicles Based on Model Predictive Control". MA thesis. Beijing, China: Beijing Institute of Technology, 2015.

[4] V. Changoski et al. "Implementing Model Predictive Control (MPC) in Steer-by-Wire Systems for Future Automated Vehicles". In: IOP Conf. Ser. Mater. Sci. Eng. 1311 (2024), p. 012024. DOI: 10.1088/1757-899X/1311/1/012024.

[5] Z.W. Zhu et al. "Multi-shooting nonlinear MPC trajectory tracking control for autonomous vehicles". In: Control. Theory Appl. (2025). (accessed on 27 March 2025), pp. 1–9. URL: http://kns.cnki.net/kcms/detail/44.1240.TP.20250311.1350.016.html.

[6] X.Y. Xie et al. "Trajectory Tracking Control of Intelligent Vehicles Based on Changing the Time Step of the Control Horizon". In: J. Jilin Univ. (Eng. Technol. Ed.) 54 (2024), pp. 620–630. DOI: 10.13229/j.cnki.jdxbgxb.2024.04.004.

[7] E. Kayacan, H. Ramon, and W. Saeys. “Robust trajectory tracking error model-based predictive control for unmanned ground vehicles”. In: IEEE/ASME Trans. Mechatron. 21 (2015), pp. 806–814. DOI: 10.1109/TMECH.2015.2472719.

[8] Y. Liu et al. “A Fuzzy Path Tracking Control Method Based on the Pure Pursuit Model”. In: Mach. Des. Res. 38 (2022), pp. 136–140+157.

[9] W. Liu et al. “Path Tracking Control for Low-Speed Intelligent Vehicles Based on Improved Pure Pursuit”. In: Sci. Technol. Eng. 24 (2024), pp. 10983–10992.

[10] M. Wang et al. “Improved Pure Pursuit Algorithm Based Path Tracking Method for Autonomous Vehicle: Regular Papers”. In: J. Adv. Comput. Intell. Intell. Inform. 28 (2024), pp. 1034–1042. DOI: 10.20961/jaciii.v28i6.7455.

[11] R.D. Chen. “Research on Trajectory Tracking Control of Unmanned Racing Cars Based on the PP Algorithm”. MA thesis. Guangzhou, China: Guangdong University of Technology, 2022. DOI: 10.27218/d.cnki.ggdgy.2022.000329.

[12] J. Ni et al. “Path Tracking Motion Control Method of Tracked Robot Based on Improved LQR Control”. In: Proceedings of the 2022 41st Chinese Control Conference (CCC). Hefei, China, July 2022.

[13] S. Trapanese, A. Naddeo, and N. Cappetti. “A preventive evaluation of perceived postural comfort in car-cockpit design: Differences between the postural approach and the accurate muscular simulation under different load conditions in the case of steering-wheel usage”. In: Proceedings of the SAE 2016 World Congress and Exhibition. Detroit, MI, USA, Apr. 2016.

[14] L. Gao et al. “Research on Improved LQR Method for Lateral Motion Control of Autonomous Driving”. In: Mech. Sci. Technol. Aerosp. Eng. 40 (2021), pp. 435–441.

[15] L. Chen et al. “LQR Lateral Control of Intelligent Vehicles Based on Optimal Front Wheel Side Slip Force”. In: J. Tsinghua Univ. (Sci. Technol.) 61 (2021), pp. 906–912.

[16] Y. Lu, Y.F. Shen, and L.Z. Gao. “Optimal Design of Lateral LQR Controller for Autonomous Driving Based on Improved Genetic Algorithm”. In: Mach. Des. Res. 40 (2024), pp. 202–208. DOI: 10.13952/j.cnki.jscd.2024.03.030.

[17] Y.J. Zhang, L.X. Feng, and G.B. Li. “An Adaptive LQR Path Tracking Control Method for Intelligent Vehicles Considering Time-Varying Parameters”. In: Comput. Sci. (2025). (accessed on 27 March 2025), pp. 1–11. URL: http://kns.cnki.net/kcms/dtail/50.1075.tp.20241126.1821.011.html.

[18] J. Gong, K. Liu, and J. Qi. Model Predictive Control for Driverless Vehicles: A Major Tool for National Development. 2nd. Beijing, China: Posts & Telecom Press, 2020.

[19] H. Gan and X. Liao. “Lateral Control of Intelligent Electric Vehicles Based on Model Predictive Control Algorithm”. In: West. China Commun. Sci. Technol. 11 (2023), pp. 194–199. DOI: 10.13322/j.cnki.xbdxy.2023.11.026.

[20] C. Zheng, Y. Du, and Z. Liu. “A Review of Lateral Control Methods for Autonomous Vehicles”. In: Automob. Eng. 5 (2024), pp. 1–10. DOI: 10.3390/autoeng5010001.