Group 2: Ali Ghasemi, aghasemi@kth.se
Sofia Simões, ssimoes@kth.se

Problem: Q1.

In this question, the goal was to derive and implement the continuous-time linear model of the vehicle dynamics. Starting from the equations of motion given in the assignment, the system state vector was defined as

$$x(t) = \begin{bmatrix} e_d(t) & e_\psi(t) & v_y(t) & r(t) & \delta(t) \end{bmatrix}^T,$$

with control input

$$u(t) = \begin{bmatrix} u_\delta(t) & u_r(t) \end{bmatrix}^T,$$

and disturbance term given by the raceline curvature $\kappa(t)$.
The continuous-time dynamics can be written as

$$\dot{x}(t) = A_c x(t) + B_c u(t) + B_w \kappa(t),$$

$$y(t) = C x(t),$$

where $y(t) = [\, e_d(t), e_\psi(t) \,]^T$ is the measured output.
Using the vehicle parameters $(m, I_z, l_f, l_r, C_f, C_r)$ and the distance L_p, we introduced the constants

$$c_1 = -(C_f + C_r), \quad c_2 = -(C_f l_f - C_r l_r), \quad c_3 = -(C_f l_f^2 + C_r l_r^2).$$

From these, the continuous-time matrices were constructed as

$$A_c = \begin{bmatrix} 0 & v_{\text{ref}} & -1 & -L_p & 0 \\ 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & \frac{c_1}{m v_{\text{ref}}} & \frac{c_2}{m v_{\text{ref}}^2} - 1 & \frac{C_f}{m} \\ 0 & 0 & \frac{c_2}{v_{\text{ref}} I_z} & \frac{c_3}{v_{\text{ref}} I_z} & \frac{C_f l_f}{I_z} \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}, \quad B_c = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & \frac{1}{I_z} \\ 1 & 0 \end{bmatrix},$$

$$B_w = \begin{bmatrix} 0 \\ v_{\text{ref}} \\ 0 \\ 0 \\ 0 \end{bmatrix}, \quad C = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix}.$$

Finally, since the dynamics are formulated with respect to the curvilinear coordinate s instead of time, the system matrices are scaled by $1/v_{\text{ref}}$ as

$$A = \tfrac{1}{v_{\text{ref}}} A_c, \quad B = \tfrac{1}{v_{\text{ref}}} B_c, \quad B_w = \tfrac{1}{v_{\text{ref}}} B_w.$$

These matrices were implemented in the function `_compute_system_matrices` inside `linear_car_model.py`, which then allows conversion to a discrete-time representation via the provided `c2d` method.
The final `_compute_system_matrices` function is presented below.

```
def _compute_system_matrices(self,v_ref : float = 20.0):
        """
        Compute the system matrices A, B, C, D for the linearized
           vehicle model.

        :param v_ref: Reference velocity for the vehicle (m/s)
        :type v_ref: float
        """


        c1 = -(self.Cf + self.Cr)
        c2 = -(self.Cf * self.lf - self.Cr * self.lr)
        c3 = -(self.Cf * self.lf**2 + self.Cr * self.lr**2)


        # Remove upon completion


        v  = v_ref
        m  = self.mass
        Iz = self.Iz
        Cf = self.Cf
        lf = self.lf
        Lp = self.lp


        A = np.array([
            [0.0,   v,       -1.0,        -Lp,    0.0],
            [0.0,   0.0,     0.0,         -1.0,   0.0],
            [0.0,   0.0,     c1/(m*v),    (c2/(m*v**2) - 1.0),    Cf/m],
            [0.0,   0.0,     c2/(Iz*v),   c3/(Iz*v),  Cf*lf/Iz],
            [0.0,   0.0,     0.0,         0.0,    0.0]
        ])

        Bw = np.array([
            [0.0],
            [v],
            [0.0],
            [0.0],
            [0.0]
        ])

        B = np.array([
            [0.0,      0.0],
            [0.0,      0.0],
            [0.0,      0.0],
            [0.0,   1.0/Iz],
            [1.0,      0.0]
        ])

        C = np.array([
            [1.0, 0.0, 0.0, 0.0, 0.0],
```

```
            [0.0, 1.0, 0.0, 0.0, 0.0]
        ])

        D  = np.zeros((2,2))
        Dw = np.zeros((2,1))


        self.A_cont  = A   * 1/v_ref
        self.B_cont  = B   * 1/v_ref
        self.Bw_cont = Bw  * 1/v_ref
        self.C_cont  = C
        self.D_cont  = D
        self.Dw_cont = Dw
```

Problem: Q2.

In the second question, the task was to compute a feedforward state and input trajectory that best tracks the reference raceline. To do so, we formulated the finite-horizon optimal control problem given by

$$\begin{aligned}
\min \sum_{k=0}^{N-1} \left(y_k^T Q y_k + u_k^T R u_k \right) + y_N^T Q y_N \\
x_{k+1} = Ax_k + Bu_k + B_w \kappa_k \\
|\delta_k| \leq 25 \frac{\pi}{180} \\
y_k = Cx_k \\
x_0 = 0_5
\end{aligned}$$

Here, N corresponds to the number of raceline points, while κ_k denotes the curvature at each point of the reference trajectory. The cost matrices were chosen as

$$Q = \begin{bmatrix} 1000 & 0 \\ 0 & 1000 \end{bmatrix}, \qquad R = \begin{bmatrix} 0.1 & 0 \\ 0 & 0.001 \end{bmatrix},$$

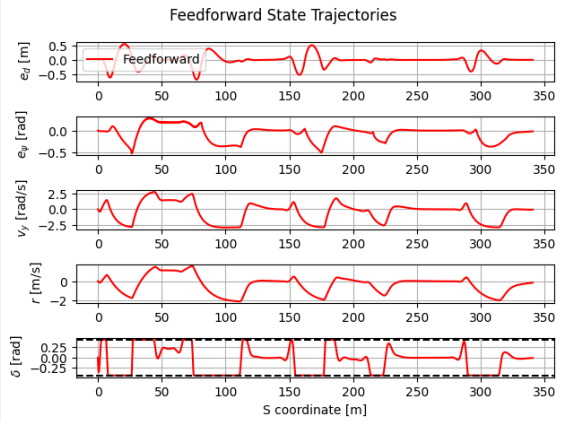
where Q penalizes lateral error and heading error, and R penalizes the control effort in steering angle rate and yaw rate input.
The problem was implemented in the function `compute_feedforward_action` within `sim.py`. Using `cvxpy`, we defined optimization variables for the entire trajectory:

$$x_{\text{ff}} = \begin{bmatrix} x_0^T & x_1^T & \dots & x_N^T \end{bmatrix}^T, \quad u_{\text{ff}} = \begin{bmatrix} u_0^T & u_1^T & \dots & u_{N-1}^T \end{bmatrix}^T.$$

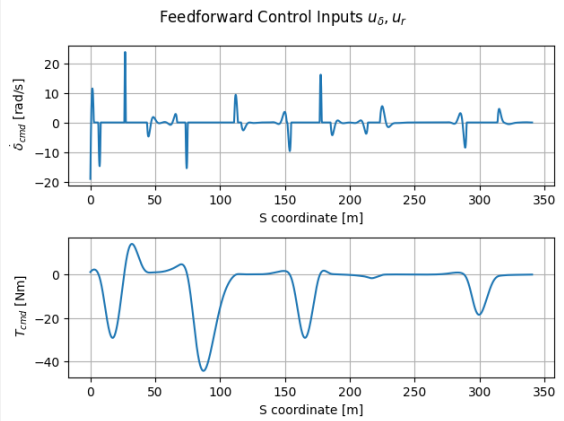
The dynamic constraints were enforced iteratively for each time step, with the raceline curvature acting as a disturbance input through B_w. The cost was accumulated across the horizon by penalizing both deviation from the raceline and input effort, with an additional terminal cost on the final state. Finally, the steering angle limit constraint

$$-\tfrac{25\pi}{180} \leq \delta_k \leq \tfrac{25\pi}{180}$$

was included to ensure validity of the linearized vehicle model.
Solving the optimization yields the optimal feedforward trajectories x_{ff} and u_{ff}, which serve as reference signals for the feedback controller design.
The obtained results, using the code below, are presented in Figure 1.



(a) State Trajectories.



(b) Control signals.

Figure 1: Feedforward Controller Design.

```python
def compute_feedforward_action(system : LinearCarModel, raceline : np.
    ndarray, ds : float ):
    """
    Compute the feedforward action for a given raceline and system.
    :param system: The linear car model system
    :type system: LinearCarModel
    :param raceline: The reference raceline to follow
    :type raceline: np.ndarray
    :param ds: Discretization step of the system
    :type ds: float
    :returns: The feedforward state and input trajectories.
    :rtype: Tuple[np.ndarray, np.ndarray]
    """

    # compute raceline curvature
    num_points              = raceline.shape[0]
    heading                 = np.unwrap(np.arctan2(np.gradient(
        raceline[:,1]), np.gradient(raceline[:,0])))
    curvature               = np.gradient(heading) / ds
    raceline_curvature      = curvature



    A  = system.A
    B  = system.B
    Bw = system.Bw
    C  = system.C

    # Define optimization variables
```

```
    x_ff = cp.Variable((num_points, system.n))              # TODO:
       create variable num_points x system.n
    u_ff = cp.Variable((num_points - 1, system.m))          # TODO:
       create variable num_points-1 x system.m
    y_ff = x_ff @ C.T  # TODO: create output as a function of system
       state

    # define cost function

    # define dynamic constraints
    constraints = []
    for i in range(num_points - 1):

        k_i           = raceline_curvature[i]
        constraints += [
                x_ff[i + 1] == A @ x_ff[i] + B @ u_ff[i] + Bw[:, 0] *
                   k_i
       ]


    # initial state constraint
    constraints += [x_ff[0] == np.zeros(system.n)]

    # Define cost function (minimize control effort and deviation from
        raceline)
    Q_ff = np.diag([1000.0, 1000.0])
    R_ff = np.diag([0.1, 0.001])
    cost = 0

    for i in range(num_points-1):
        cost += cp.quad_form(y_ff[i], Q_ff) + cp.quad_form(u_ff[i],
           R_ff)

    # Terminal cost
    cost += cp.quad_form(y_ff[num_points-1], Q_ff)

    # add constraint on steering angle
    constraints += [cp.abs(x_ff[:, 4]) <= np.deg2rad(25)]

    # Add constraints and objective to the problem
    problem = cp.Problem(cp.Minimize(cost), constraints)
    problem.solve(solver = cp.MOSEK)

    return x_ff.value, u_ff.value
```

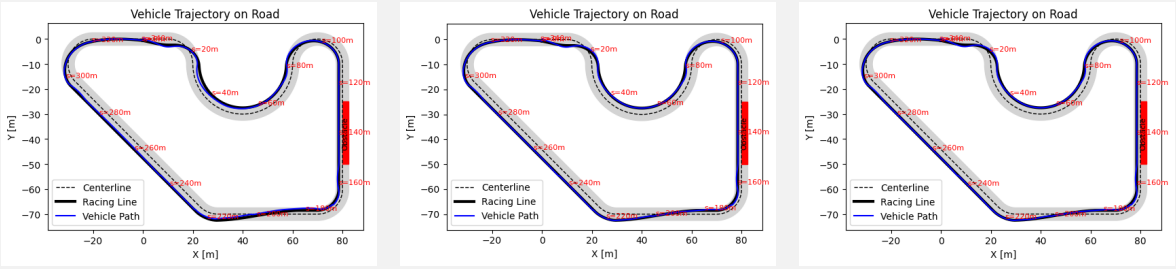
Problem: Q3.

With regard to bandwidth and control effort, it can be seen that Design 1 (small Q, and large R) has low bandwidth and is rather conservative when it comes to feedback correction. As a result, the controller response is relatively slow and the magnitude of control inputs is

modest. Design 2 (Large Q, Average R) increases the bandwidth. So it uses more bandwidth but still remains within our limits. But design 3 by the same Q and very small R acts very fast, and uses the most control effort, and this can result in saturation near the obstacles. By looking at state plots, as we expect, design 1 shows the longest transient response and error decays slowly. Designs 2 and 3 act very similarly; the error decays rapidly, and we almost have no overshoot.

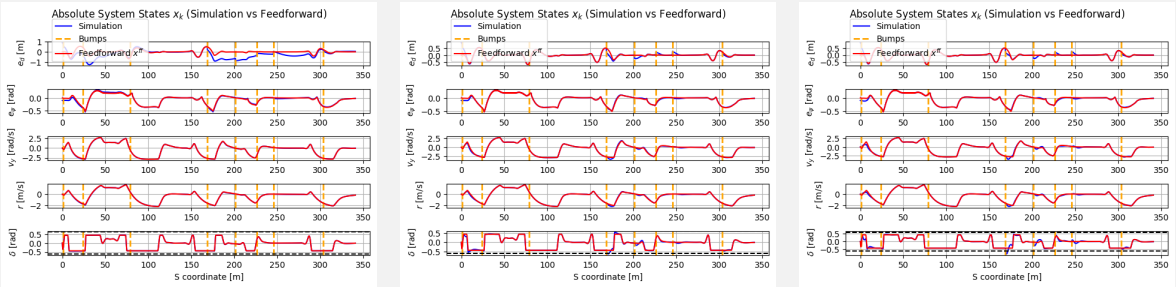
These are all aligned with the fact that a higher state weight Q (relative to R) increase the control bandwidth and penalises the error more aggressively. While reducing the R too much will result in pushing the actuators to work near their limits.

At the end, we can argue that Design 2 is the best trade-off, which has good tracking performance without using excessive control effort.



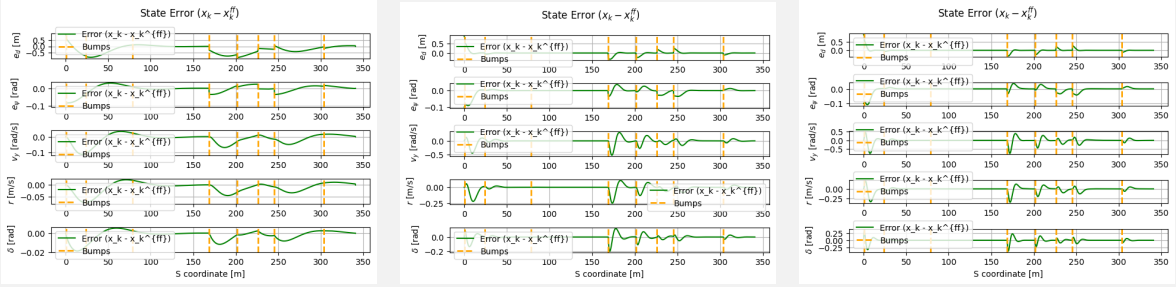
(a) Design I **(b)** Design II **(c)** Design III

Figure 2: Comparison of Trajectory for three nominated designs



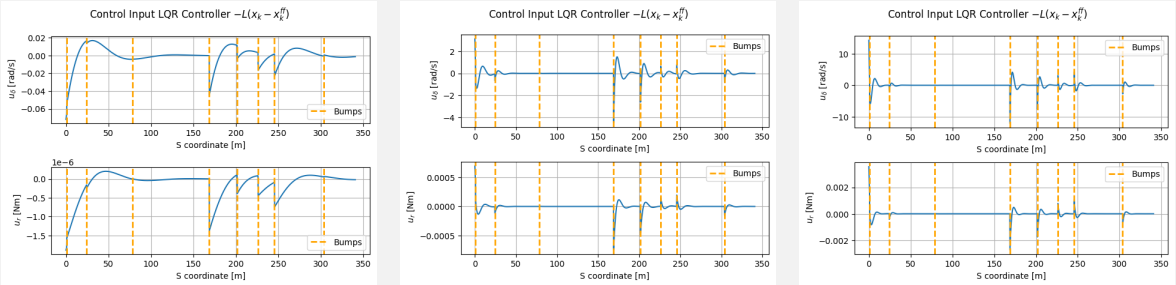
(a) Design I **(b)** Design II **(c)** Design III

Figure 3: Comparison of system states for three nominated designs



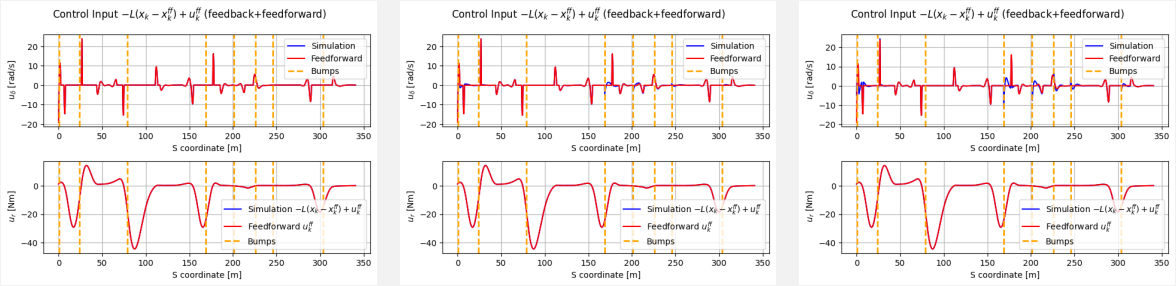
(a) Design I **(b)** Design II **(c)** Design III

Figure 4: Comparison of state error for three nominated designs



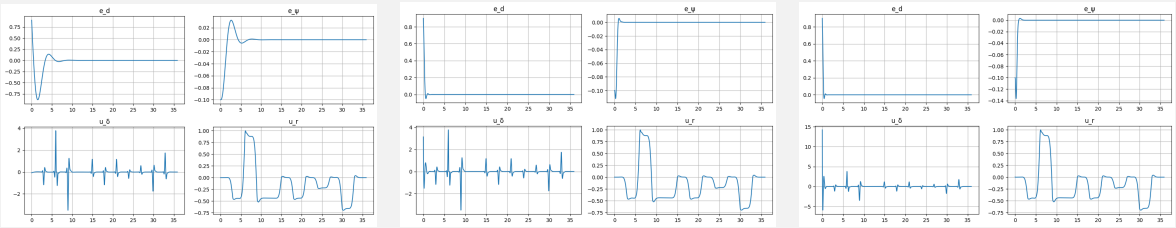
(a) Design I **(b)** Design II **(c)** Design III

Figure 5: Comparison of LQR control input for three nominated designs



(a) Design I **(b)** Design II **(c)** Design III

Figure 6: Comparison of complete control input for three nominated designs



(a) Design I **(b)** Design II **(c)** Design III

Figure 7: Comparison of Error and Control signals of three nominated designs.

Design	$u_\delta^{\min}$	$u_\delta^{\max}$	$u_r^{\min}$	$u_r^{\max}$	OS e_d [%]	T_s e_d [s]	OS e_ψ [%]	T_s e_ψ [s]
Design I	-3.5	3.78	-0.697	1	0	7.3	0	3.43
Design II	-3.5	3.78	-0.697	1	0	0.78	11.9	0.74
Design III	-5.93	14.2	-0.697	1	0	0.56	36.5	0.65

Table 1: Comparison of designs with input constraints and performance metrics.

Problem: Q4.

Finally, in the last question, we applied the Bryson's rule to tune the LQR controller. The idea is to normalize each state and input according to its maximum expected magnitude along the feedforward trajectory. Specifically, the coefficients were defined as

$$q_i = \max_{k=0,\dots,N} |x_{k,i}^{\text{ff}}|, \quad i = 1, \dots, 5,$$

$$r_i = \max_{k=0,\dots,N-1} |u_{k,i}^{\text{ff}}|, \quad i = 1, 2.$$

Using these coefficients, the state and input weighting matrices are given by

$$Q = \text{diag}\left(\frac{p_1}{q_1^2}, \frac{p_2}{q_2^2}, \frac{p_3}{q_3^2}, \frac{p_4}{q_4^2}, \frac{p_5}{q_5^2}\right), \qquad \text{and} \qquad R = \text{diag}\left(\frac{l_1}{r_1^2}, \frac{l_2}{r_2^2}\right),$$

where the coefficients p_i and l_i allow tuning the relative importance between states and inputs.

(a) Case $l_i \gg p_i$

For $p = [1, 1, 1, 1, 1]$ and $l = [100, 100]$, the LQR controller applies very small control inputs, resulting in slow correction of state errors. The vehicle tracks the reference trajectory conservatively, with larger lateral and heading deviations, but exhibits smooth and stable behavior. This illustrates the trade-off between minimizing actuator effort and maintaining accurate tracking.

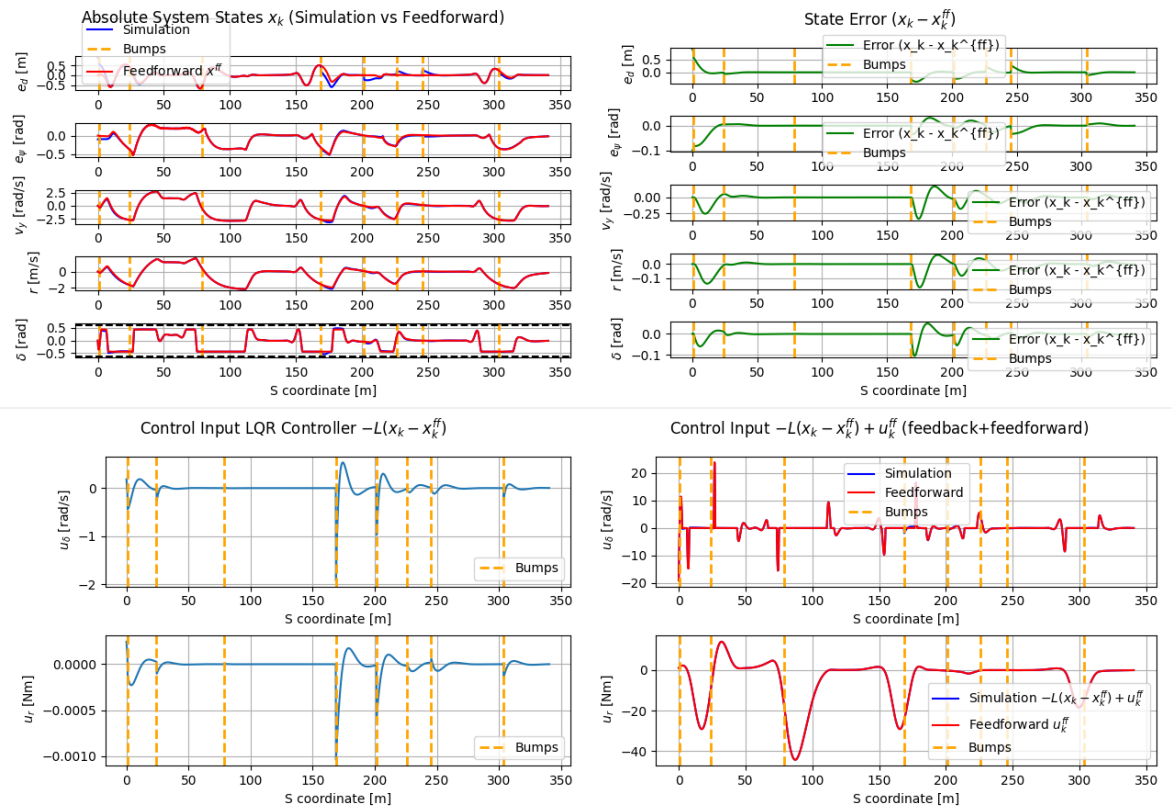


Figure 8: Results for the case $l_i \gg p_i$.

(b) Case $p_i \gg l_i$

For the case with $p = [100, 100, 100, 100, 100]$ and $l = [1, 1]$, the LQR controller aggressively corrects deviations using large control inputs, leading to fast convergence and small tracking errors. While tracking accuracy improves, the actuator effort is higher and the response less smooth, reflecting the trade-off of prioritizing state regulation over control economy.

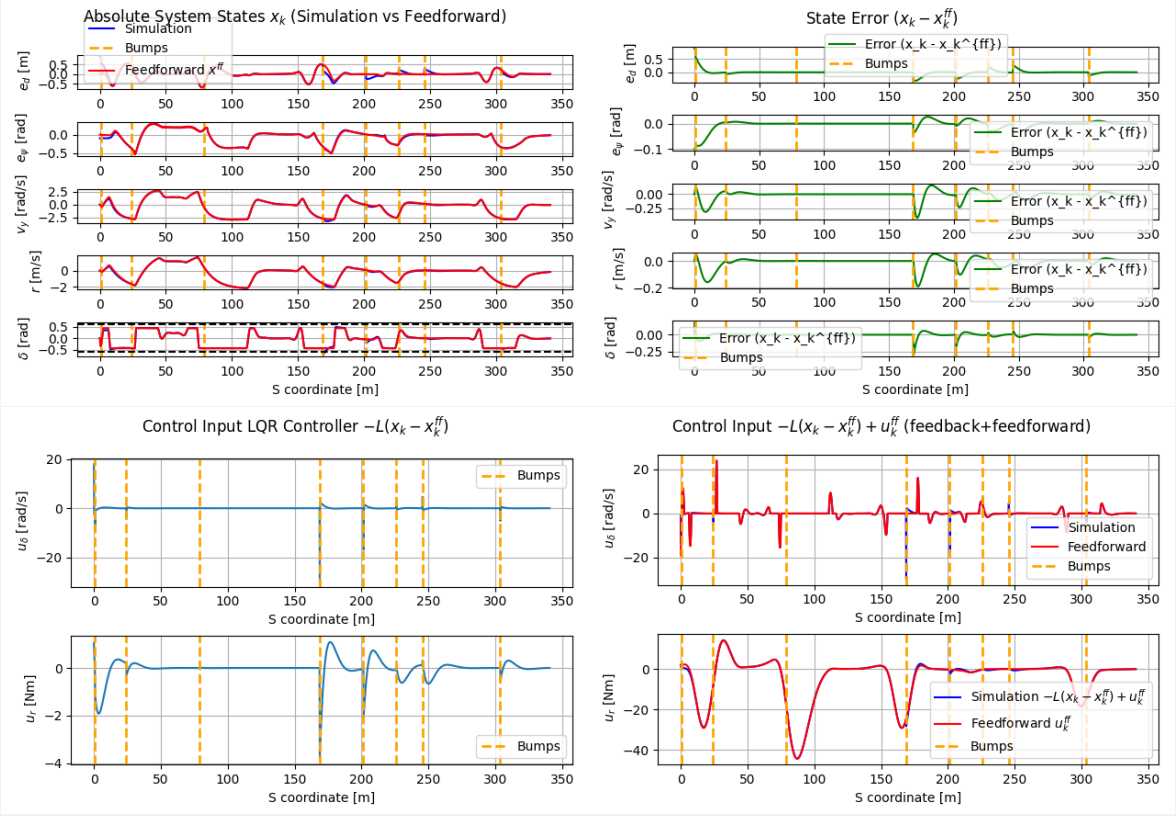


Figure 9: Results for the case $p_i \gg l_i$.

(c) Can we achieve a steering rate limit of 35 degrees/sec?

For this part, we followed the hint and re-tuned the LQR using Bryson scaling so the controller does not penalizes steering angle (small p_5) but heavily penalizes the steering rate (large l_1).

We then grid-searched those weights to see if any tuning could keep the total steering-rate command under $35°/s$ while still tracking well.

But based on our results, none of the LQR tunings met the $35°/s$ limit. The closest option still needed about $\pm 217°/s$ (with $\delta_{\max} \approx 3.6°$), and while tracking was quick and clean ($T_s(e_d) \approx 1.15\,s$, $T_s(e_\psi) \approx 0.86\,s$, $\sim 0\%$ overshoot), it still violates the rate limit.

We believe that this shows the big spikes are coming from the feedforward plan from Q2; and feedback tuning alone cannot pull down the signal below $35°/s$ without sacrificing tracking.

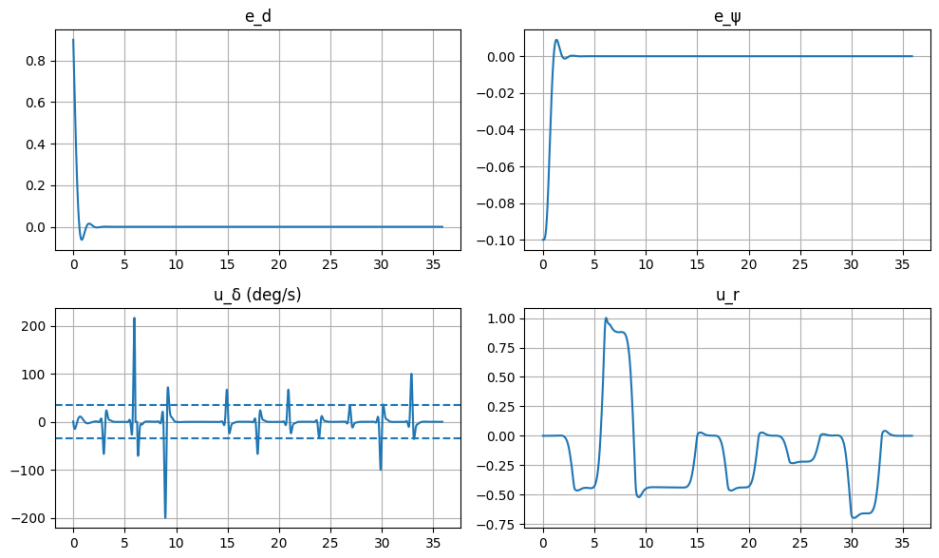


Figure 10: Best achieved performance after grid search of parameters.

To make the trade-off clear, we then kept a reasonable LQR and simulated a hard actuator rate limiter at 35°/s. Now we can see the rate stays within the bounds, but closed-loop performance collapsed.e_d and e_ψ became slow and damped (about 6.5% and 18.3% overshoot, with $\sim 36\,\mathrm{s}$ settling time) and the error plots increase drastically.
This means forcing the steering to move that slowly means the car cannot follow the planned trajectory. As a result, the error grows while the controller cannot work properly.

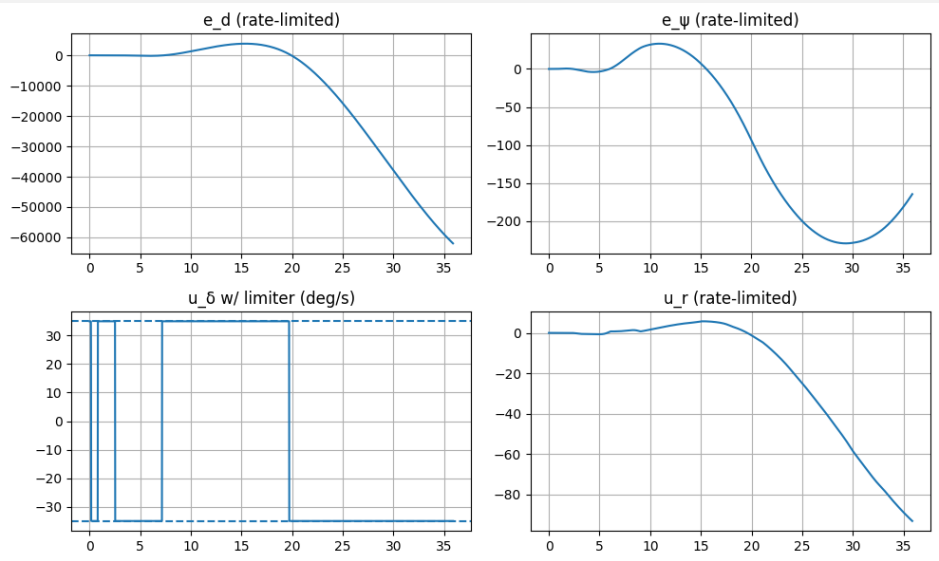


Figure 11: Performance after forcing the slow streeing rate.

(d) Can we achieve a steering angle limit of 35 degrees?

We used Bryson-scaled LQR and followed the hint "don't punish steering angle." So using

grid search (scaling down $p5$ and scaling up $l1$ and $l2$), we set a single weight for the tracking states $p_{\text{common}} = 3$ (this is our base weight for starting the grid search) and scaled them. Eventually we kept the angle weight small $p_5 = 0.5$ so the controller can freely use δ, and put only a mild penalty on steering-rate $l_1 = 5$ with a moderate yaw-moment penalty $l_2 = 3$. This forced the controller to kill the lateral and heading errors. The result was fast settling ($T_s \approx 0.62\,\text{s}$) but under-damped behaviour (overshoot $\approx 51.5\%$), and we achieved the steering angle limit of ($|\delta|_{\max} \approx 35.7°$). The u_δ subplot shows large rate bursts; that aggressiveness is exactly why it's quick but overshoots.

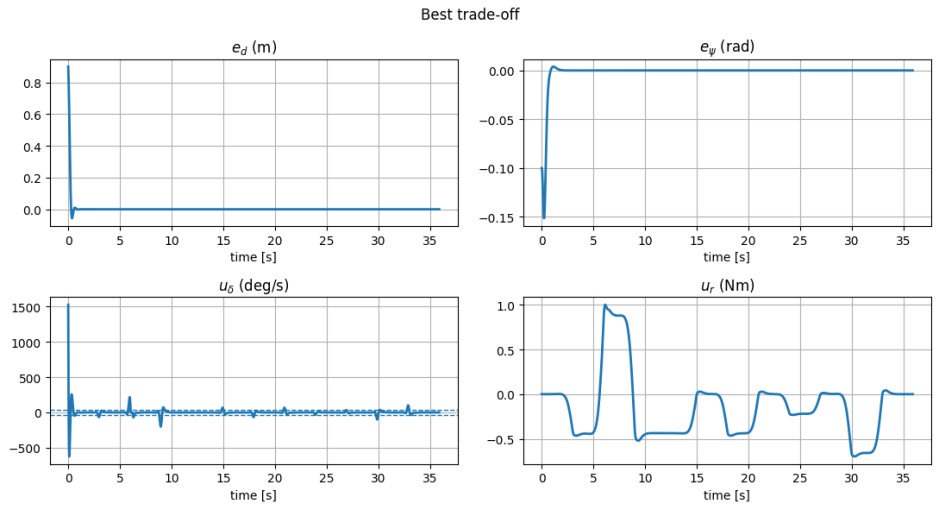


Figure 12: Best achieved performance after grid search of parameters.

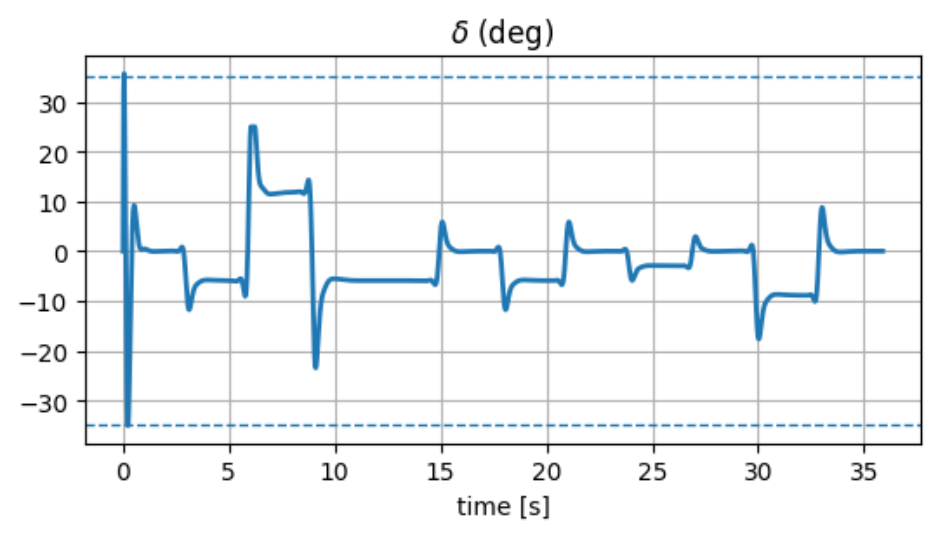


Figure 13: Steering Angle

To compensate for the overshoot, we had to add damping. To do so, we put larger weights on the velocity/yaw states $[p_{v_y}, p_r] = [6, 6]$, and increased the steering-rate penalty to $l_1 = 12$

to smooth the action, made heading error a bit costlier than lateral $[p_{e_d}, p_{e_\psi}] = [1.2, 2.0]$, and set $p_5 = 1.0$ so angle is still allowed. We also penalized yaw-moment more ($l_2 = 5$), so the controller relies more on steering than on u_r. That gave $|\delta|_{\max} \approx 17.3°$ (well less than the 35° limit), overshoot cut roughly in half ($\approx 27.9\%$) with respect to first design, and we achieved similar settling ($T_s \approx 0.68\,\text{s}$). The new plots also show smoother u_δ.

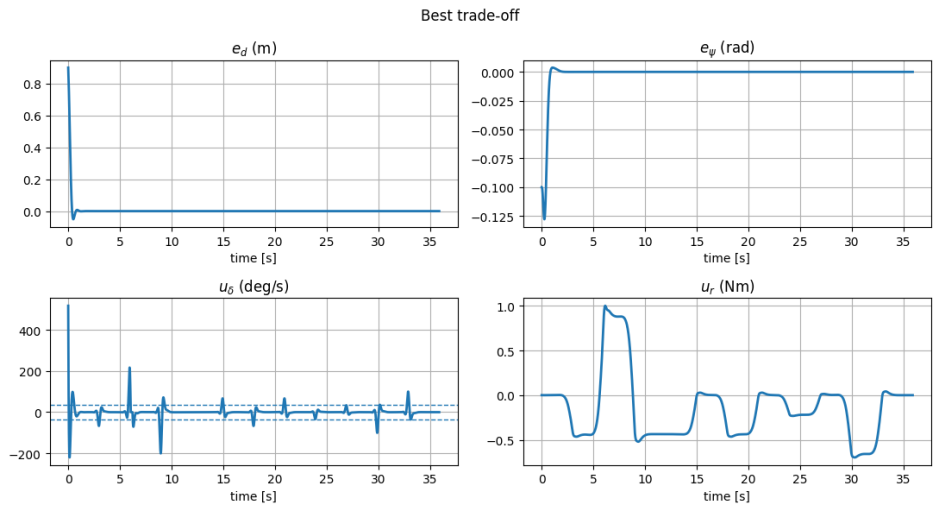


Figure 14: Best achieved performance after grid search of parameters and considering overshoot.

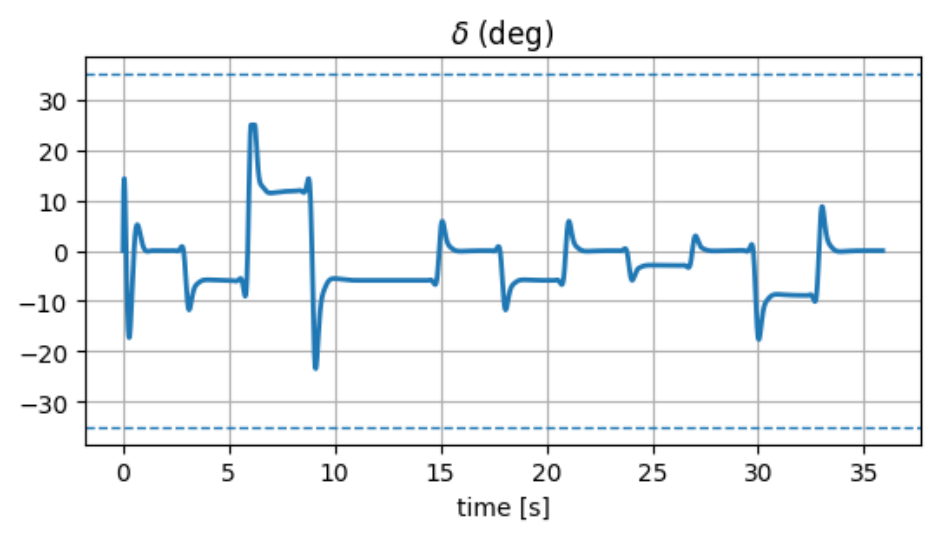
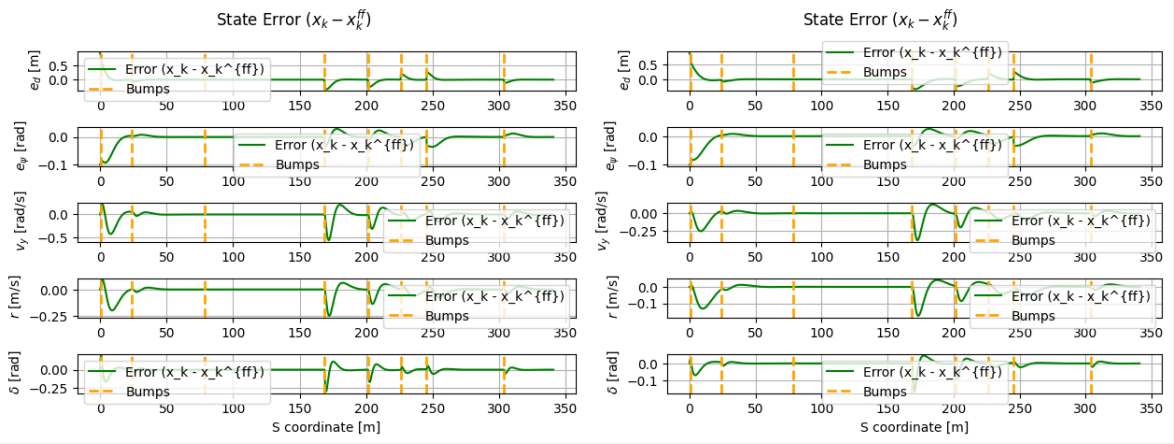


Figure 15: Steering angle.

At the end we can conclude that design 1 is the fast (great settling time), but big overshoot and angle right at the limit. Design 2 shows the effect of adding damping: the response is still quick, while the motion is smoother.



(a) First Design.

(b) Second design.