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

Problem: Q1.

In this question, the method `setup_mpc_problem` was implemented within the class `MPC`. The goal was to formulate the Model Predictive Control problem in `cvxpy` according to the generic recipe provided in the assignment.
The state and input trajectories over the prediction horizon were defined as optimization variables, while the current state and the feedforward trajectories were introduced as parameters. The error dynamics were then expressed in terms of the deviation between the system variables and the feedforward reference.
The cost function was constructed as the sum of quadratic stage costs over the horizon, penalizing both state and input deviations, plus a quadratic terminal cost on the final state deviation. The constraints included the system dynamics, input and state bounds, the terminal bounds, and the initial condition constraint.
The final code is presented below.

```
def setup_mpc_problem(self):
        """
        Set up the MPC optimization problem given the current state
           and reference trajectory.

        :param x0: Current state of the system.
        :type x0: np.ndarray
        :param xr: Reference trajectory over the prediction horizon.
        :type xr: np.ndarray

        """

        ############################################################
        ## Extract system information
        ############################################################
        A = self.model.A
        B = self.model.B
        n = self.n
        m = self.m
        N = self.N

        # Initialize cost matrices
        constraints = []
        cost = 0

        ############################################################
        ## Define Optimization Problem Variables and Parameters.
        ############################################################

        ## Variables definition
        x =  cp.Variable((n,N)) # todo: State variables over the
           horizon nxN
        u =  cp.Variable((m,N-1)) # todo: Control inputs over the
```

```
            horizon  mx(N-1)

        ## Define initial state parameter
        x_init = cp.Parameter(n) #todo: initial state parameter
        x_ff   = cp.Parameter((n,N)) # todo:feedforward reference
            state parameter nxN
        u_ff   = cp.Parameter((m,N-1)) # todo:feedforward reference
            input parameter mx(N-1)

        ## Define state error and input error
        e_x = x - x_ff # todo: create error as state difference
        e_u = u - u_ff # todo:create error as input difference


        #############################################################
        ## Define Cost Function.
        #############################################################
        for k in range(N-1):
            cost += cp.quad_form(e_x[:, k], self.Q) + cp.quad_form(e_u
                [:, k], self.R) # todo stage cost

        cost += cp.quad_form(e_x[:, N-1], self.Qt) # todo terminal
            cost

        #############################################################
        ## Define Constraints.
        #############################################################

        for k in range(N-1):

            # Define system dynamics constraints (e_x_{k+1} = A*e_x_k
                + B*e_u_k)
            constraints += [e_x[:, k+1] == A @ e_x[:, k] + B @ e_u[:,
                k]] # #todo:dynamic constraints

            # Define input constraints (self.lbu <= u_k <= self.ubu)
            for jj in range(self.m) :
                if self.lbu[jj] != -np.inf :
                    constraints += [u[jj, k] >= self.ulb[jj]] # todo:
                        fill bounds
                if self.ubu[jj] != np.inf :
                    constraints += [u[jj, k] <= self.ubu[jj]] # todo:
                        fill bounds

            # Define state constraints (self.lbx <= x_k <= self.ubx)
            for ii in range(self.n) :
                if self.lbx[ii] != -np.inf :
                    constraints += [x[ii, k] >= self.lbx[ii]] # todo:
                        fill bounds
                if self.ubx[ii] != np.inf :
                    constraints += [x[ii, k] <= self.ubx[ii]] # todo:
                        fill bounds
```

```
        # terminal constraint
        for ii in range(self.n) :
            if self.lbx[ii] != -np.inf :
                constraints += [x[ii, N-1] >= self.lbx[ii]] # todo:
                    fill bounds
            if self.ubx[ii] != np.inf :
                constraints += [x[ii, N-1] <= self.ubx[ii]] # todo:
                    fill bounds

        # Define initial state constraint
        constraints += [x[:, 0] == x_init] # todo: fill constraint

        ############################################################
        ## Define Optimization Problem and store variables in self.
        ############################################################


        self.problem = cp.Problem(cp.Minimize(cost), constraints)
        self.x      = x
        self.u      = u
        self.e_u    = e_u
        self.e_x    = e_x
        self.x_ff   = x_ff
        self.u_ff   = u_ff
        self.x_init = x_init

        self.is_setup = True
```

Problem: Q2.

In this question, the method `solve_mpc` was implemented inside the `MPC` class. This function takes as input the current state of the system, the state feedforward trajectory, and the input feedforward trajectory. These values are assigned to the corresponding parameters of the optimization problem defined in Question 1. The problem is then solved using the chosen solver, and the optimal control input sequence is obtained. Finally, the first input $u_0^\star$ is returned, which represents the action to be applied to the system at the current step.
The code of the function is presented below.

```
    def solve_mpc(self, x0: np.ndarray, x_ff: np.ndarray, u_ff: np.
        ndarray):
        """
        Solve the MPC optimization problem for the current state and
            reference trajectory.

        :param x0: Current state of the system.
        :type x0: np.ndarray
        :param xr: Reference trajectory over the prediction horizon (N
            ,n).
        :type xr: np.ndarray
        :param ur: Reference input over the prediction horizon (N-1,m)
```

```
        .
    :type ur: np.ndarray
    :return: Optimal control input for the current time step.
    :rtype: np.ndarray
    """

    if not self.is_setup:
        raise ValueError("MPC problem is not set up. Call
            setup_mpc_problem() before solving.")

    # Set parameter values
    self.x_init.value = x0 # todo:fill here
    self.x_ff.value   = x_ff # todo:fill here
    self.u_ff.value   = u_ff # todo:fill here

    # Solve the optimization problem
    self.problem.solve(solver=self.solver_type, warm_start=self.
        warm_start)

    if self.problem.status not in ["optimal", "optimal_inaccurate"
        ]:
        raise ValueError(f"MPC problem did not solve to optimality
            . Status is {self.problem.status}")

    # Return the first control input
    return self.u.value[:, 0]
```

Problem: Q3.

In this question, the given MPC controller implementation was executed using the provided code, with the specified initial condition and reference trajectory. The results confirm that the MPC correctly applies the receding horizon strategy: at each time step, the optimization problem is solved, and only the first optimal control input is applied.
Some representative plots are presented below, together with the Python code used to generate them. Particular attention is given to the comparison between the LQR and MPC control inputs. The difference is effectively zero, with only minor deviations attributable to numerical precision. This behavior is expected, as both controllers rely on the same quadratic cost function and linear system dynamics. When no additional constraints are active, the MPC solution coincides with that of the LQR.

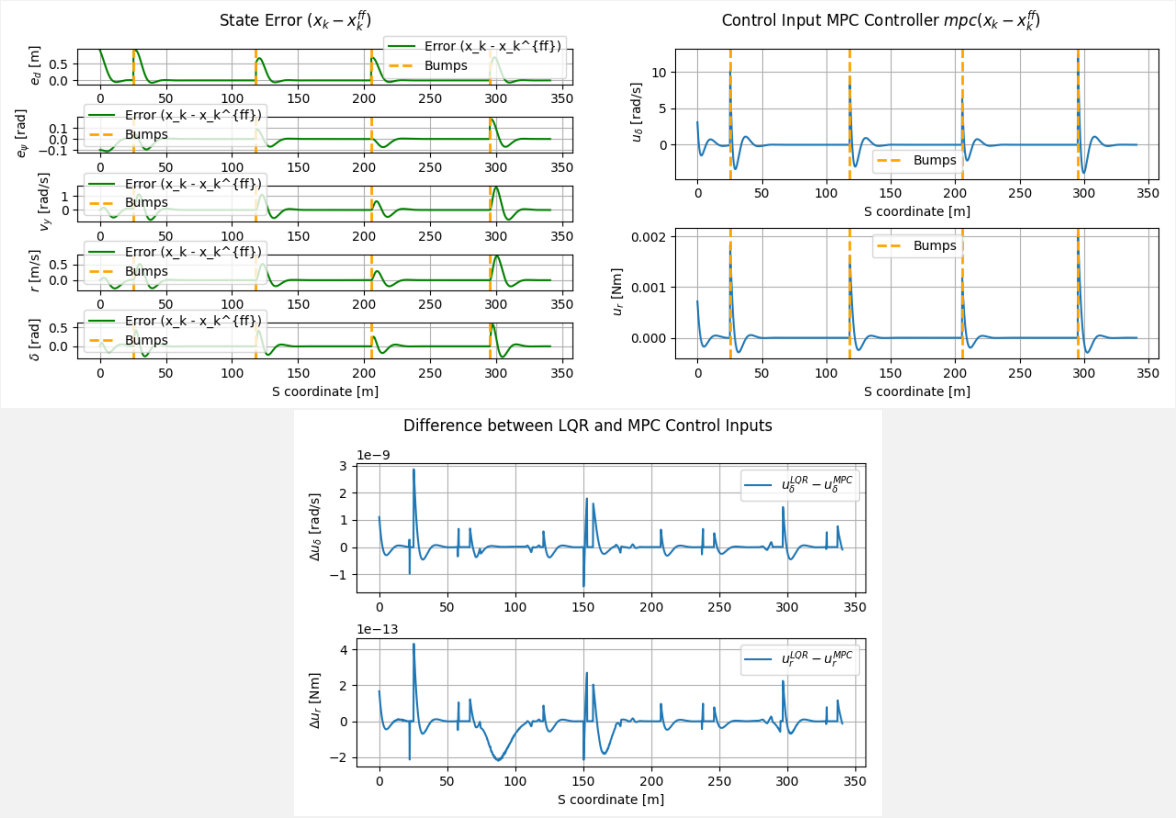


Figure 1: Results obtained using the MPC controller.

In the above Figure, it is important to mention the plot where there is a commparison between the LQR and MPC control inputs. The difference is essentially zero, with only very small deviations at the level of numerical precision. This outcome is expected, since both controllers are based on the same quadratic cost and linear system dynamics. The MPC reduces to the LQR solution when no additional constraints are active.

```
####################################
# Q3
####################################

Q = np.diag([10, 1, 0.1, 0.1, 1])
R = np.diag([0.1, 0.1])


L = system.get_lqr_controller(Q, R)
P = system.get_lqr_cost_matrix(Q, R)


##  Define MPC controller
mpc_controller = MPC(model      = system,
                     N          = 20,
                     Q          = Q,
                     R          = R,
                     Qt         = P*1,
```

```
                        warm_start = True,
                        solver     = 'CLARABEL')
mpc_controller.setup_mpc_problem()

x0 = np.array([0.9, -0.1, 0, 0, 0])

simulate_system_run(x0          = x0,
                    system      = system,
                    mpc_controller = mpc_controller,
                    lqr_controller = L,
                    racetrack   = racetrack,
                    raceline    = raceline,
                    ds          = ds,
                    x_ff        = x_ff,
                    u_ff        = u_ff)
```

Problem: Q4.

In this question, Q and R were tuned using Bryson's rule as a principled baseline. Each diagonal entry of Q is defined as p_i/q_i^2, where q_i is the maximum magnitude of the corresponding state along the feedforward trajectory, and each diagonal entry of R is defined as ℓ_j/r_j^2, where r_j is the maximum magnitude of the corresponding input along the feedforward trajectory. The multipliers were selected as $p = [1.2, 2.0, 6.0, 6.0, 1.0]$ for the states and $\ell = [12.0, 5.0]$ for the inputs, reusing the same values as in the previous assignment. This choice balances tracking performance and actuator smoothness, emphasizing lateral-velocity and yaw-rate damping while moderating steering-rate peaks.
Some of the resulting plots are shown below, along with the Python code used to generate them.

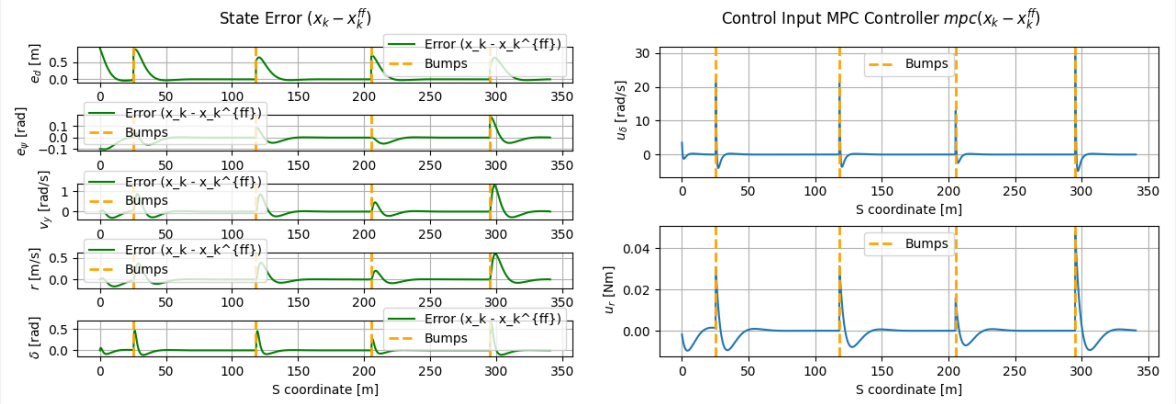


Figure 2: Results obtained using the Bryson's rule.

```
#####################################
# Q4
#####################################
```

```
q1   = np.max(np.abs(x_ff[:, 0]))
q2   = np.max(np.abs(x_ff[:, 1]))
q3   = np.max(np.abs(x_ff[:, 2]))
q4   = np.max(np.abs(x_ff[:, 3]))
q5   = np.max(np.abs(x_ff[:, 4]))

r1 = np.max(np.abs(u_ff[:, 0]))
r2 = np.max(np.abs(u_ff[:, 1]))

p1 = 1.2
p2 = 2.0
p3 = 6.0
p4 = 6.0
p5 = 1.0

l1 = 12.0
l2 = 5.0

Q = np.diag([p1/q1**2, p2/q2**2, p3/q3**2, p4/q4**2, p5/q5**2])
R = np.diag([l1/r1**2, l2/r2**2])

L = system.get_lqr_controller(Q, R)
P = system.get_lqr_cost_matrix(Q, R)


##  Define MPC controller
mpc_controller = MPC(model      = system,
                     N          = 20,
                     Q          = Q,
                     R          = R,
                     Qt         = P*1,
                     warm_start = True,
                     solver     = 'CLARABEL')
mpc_controller.setup_mpc_problem()

x0 = np.array([0.9, -0.1, 0, 0, 0])

simulate_system_run(x0         = x0,
                    system     = system,
                    mpc_controller = mpc_controller,
                    lqr_controller = L,
                    racetrack  = racetrack,
                    raceline   = raceline,
                    ds         = ds,
                    x_ff       = x_ff,
                    u_ff       = u_ff)
```

Problem: Q5.

In this step, constraints were introduced into the MPC formulation to ensure realistic and safe vehicle behavior. The steering angle was limited to $|\delta| \leq 35\pi/180$ rad, the lateral velocity was restricted to $|v_y| \leq 3.1$ m/s, the steering input was bounded by $|u_\delta| \leq 50$, and the longitudinal input was constrained by $|u_r| \leq 60$. These bounds were implemented using the controller's functions for setting upper and lower limits on both states and inputs. The code with these constraints implemented is provided below:

```
# ####################################
# # Q5
# ####################################

Q = np.diag([100, 10, 10, 10, 1])
R = np.diag([10, 0.1])
lambda_val = 100.


L = system.get_lqr_controller(Q, R)
P = system.get_lqr_cost_matrix(Q, R)


##  Define MPC controller
mpc_controller = MPC(model      = system,
                     N          = 20,
                     Q          = Q,
                     R          = R,
                     Qt         = P*lambda_val,
                     warm_start = True,
                     solver     = 'MOSEK')

mpc_controller.add_lower_bound_on_input(0, -50)
mpc_controller.add_upper_bound_on_input(0,  50)

mpc_controller.add_lower_bound_on_input(1, -60)
mpc_controller.add_upper_bound_on_input(1,  60)

mpc_controller.add_lower_bound_on_state(4, -35*np.pi/180)  # u_delta
   >= -50 deg
mpc_controller.add_upper_bound_on_state(4,  35*np.pi/180) # u_delta <=
    50 deg

mpc_controller.add_lower_bound_on_state(2, -3.1)
mpc_controller.add_upper_bound_on_state(2,  3.1)
mpc_controller.setup_mpc_problem()


x0 = np.array([0.9, -0.1, 0, 0, 0])

simulate_system_run(x0         = x0,
                    system     = system,
                    mpc_controller = mpc_controller,
                    lqr_controller = L,
```

```
                racetrack   = racetrack,
                raceline    = raceline,
                ds          = ds,
                x_ff        = x_ff,
                u_ff        = u_ff)
```

After including these constraints, the MPC optimization problem was solved at each step while guaranteeing that all predicted trajectories respected the imposed limits.
Figure 3 shows the state errors (left) and control inputs (right) along the racetrack. The steering angle and lateral velocity remain within their specified bounds, while the control inputs also satisfy the imposed limits. The results confirm that the MPC constraints were correctly implemented and actively enforced throughout the simulation.

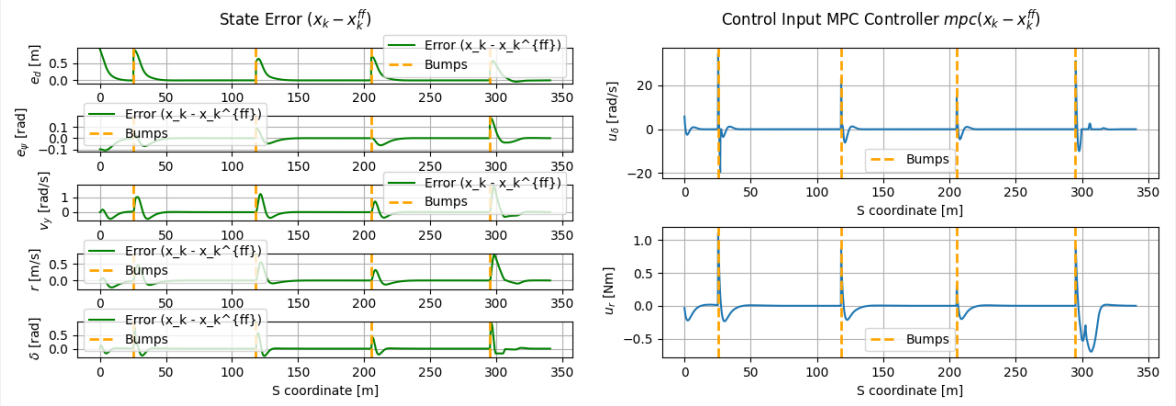


Figure 3: Results obtained using the MPC controller with constraints.

Problem: Q6.

Figures 4 to 7 show the effect of increasing the terminal cost weight λ on the performance of the MPC controller. For $\lambda = 2$ (Figure 4), the vehicle follows the trajectory but with noticeable tracking errors and oscillatory control inputs. With $\lambda = 10$ (Figure 5), the performance improves, though some deviations and input oscillations remain. At $\lambda = 50$ (Figure 6), the controller achieves the best balance: the vehicle path closely matches the racing line, state errors are minimal, and the control inputs are smooth and well within limits. Finally, for $\lambda = 100$ (Figure 7), the controller is also stable, but the control actions become unnecessarily large and conservative. Overall, $\lambda = 50$ provides the most effective compromise between stability, tracking accuracy, and control effort.

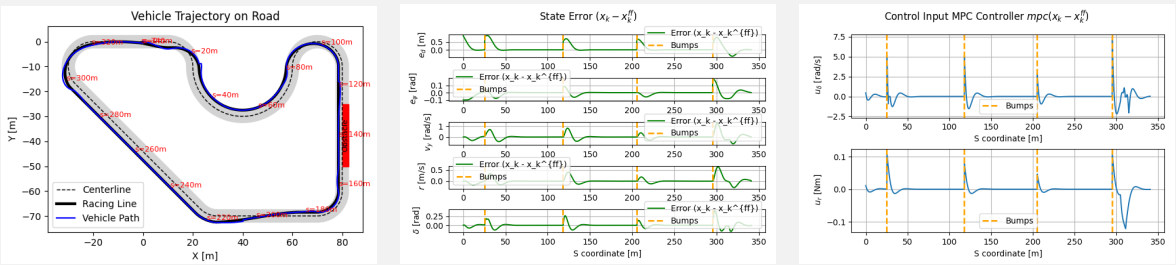


Figure 4: Results obtained using the MPC controller with constraints and $\lambda = 2$.

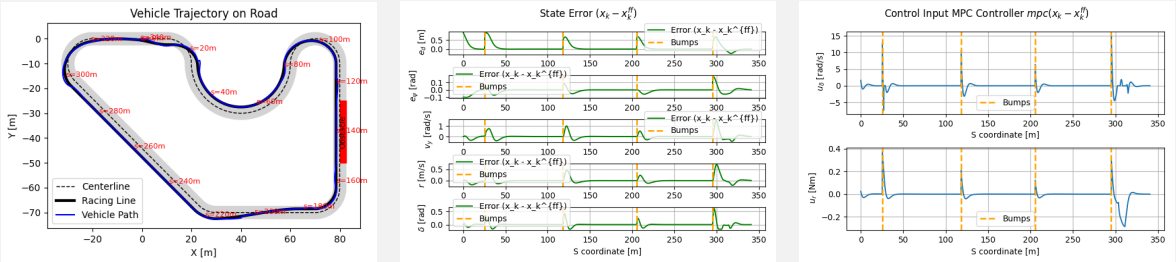


Figure 5: Results obtained using the MPC controller with constraints and $\lambda = 10$.

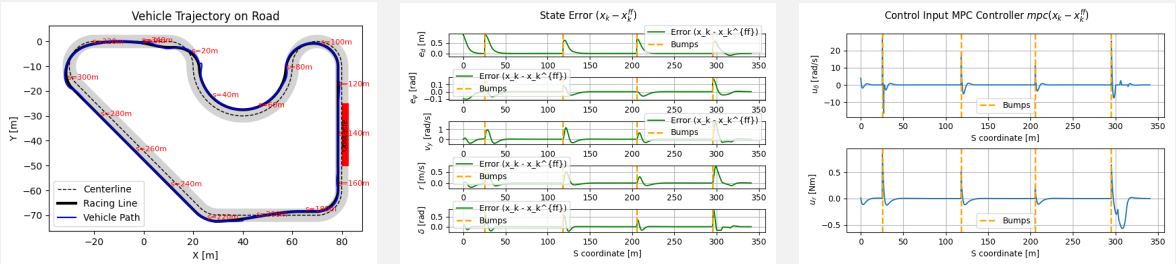


Figure 6: Results obtained using the MPC controller with constraints and $\lambda = 50$.

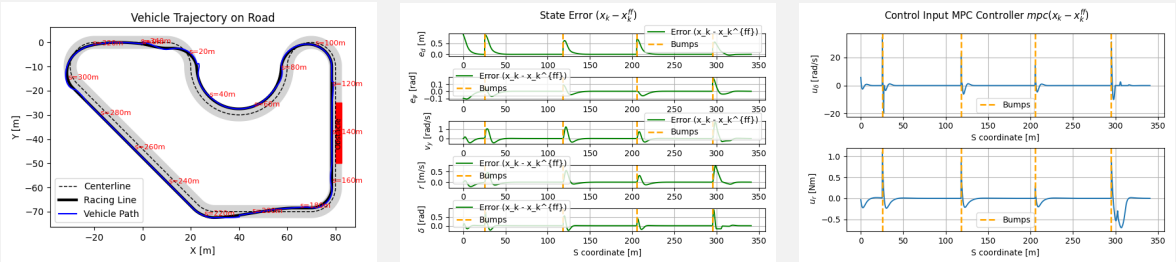


Figure 7: Results obtained using the MPC controller with constraints and $\lambda = 100$.

Problem: Q7.

7.1

For this part, we used the configuration as Table 1.

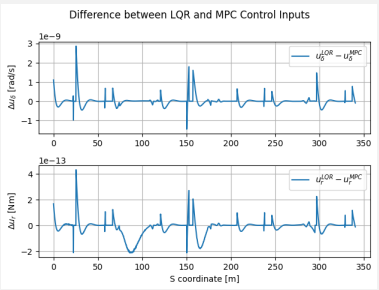
Case	Solver	Warm start	Q_T	N	Avg time(ms)	Std(ms)	Max(ms)	10 Hz
1	CLARABEL	True	P	20	0.82	0.35	1.46	Yes
2	CLARABEL	True	P	40	1.23	0.49	2.20	Yes
3	CLARABEL	True	P	80	2.18	0.86	3.67	Yes

Table 1: Solver performance for different values of N with warm start enabled.

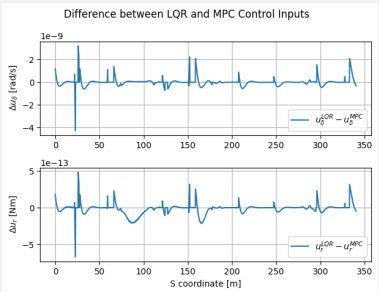
Across $N = \{20, 40, 80\}$ the MPC–LQR input difference remains at numerical noise ($\approx 10^{-9}$ rad/s and 10^{-13} Nm). This matches the theory that implies with no active constraints and a terminal weight equal to the LQR cost-to-go P, finite-horizon MPC reproduces the LQR

first move exactly, so changing N doesn't alter the control law. The tiny spikes seen in the difference plots are consistent with solver tolerances and are not real discrepancies. so the implementation is behaving as we expected.
Computation time grows roughly linearly with the horizon: for $N = 20, 40, 80$ we measured average/maximum per-step times of about (0.82/1.46) ms, (1.23/2.20) ms, and (2.18/3.67) ms, all well below the 10 Hz budget (100 ms). Within each lap the time trace increases gradually and shows small peaks near the marked "bumps", this indicates a few extra solver iterations.
To find the maximal N, we did a simple linear fit of the worst-case times versus N that yielded approximately $N_{\max} \approx 2.7 \times 10^3$ for a 100 ms budget; Of course, this is a theoretical upper bound and one can try values between 2500 and 2700 to evaluate it. (The code is as follows.)

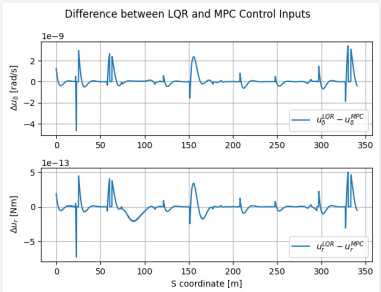
```
import numpy as np
Ns    = np.array([20, 40, 80])
t_max = np.array([1.46, 2.20, 3.67])

a, b = np.polyfit(Ns, t_max, 1)
N_max_est = int(np.floor((100 - b) / a))
print("Estimated N_max for 10 Hz:", N_max_est)
```



(a) N = 20

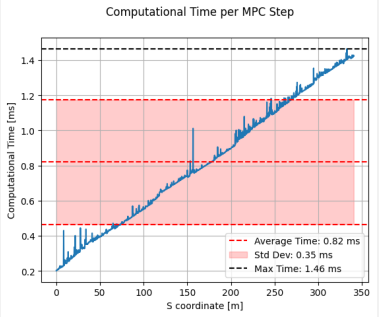


(b) N = 40

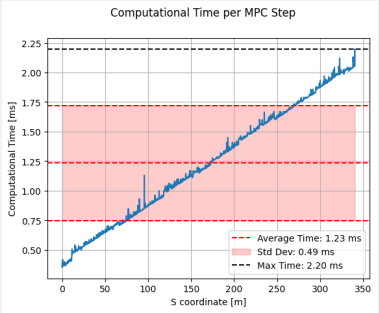


(c) N = 80

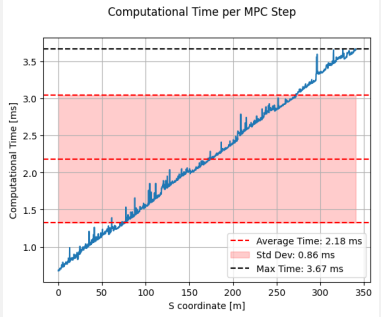
Figure 8: Difference between LQR and MPC controller inputs for different horizons



(a) N = 20



(b) N = 40

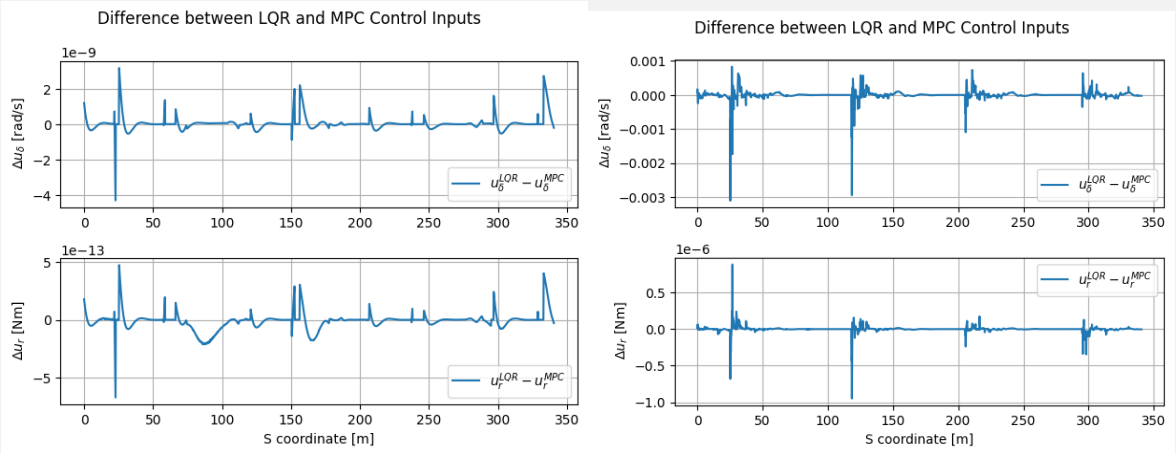


(c) N = 80

Figure 9: Computational Time per MPC step for different horizons.

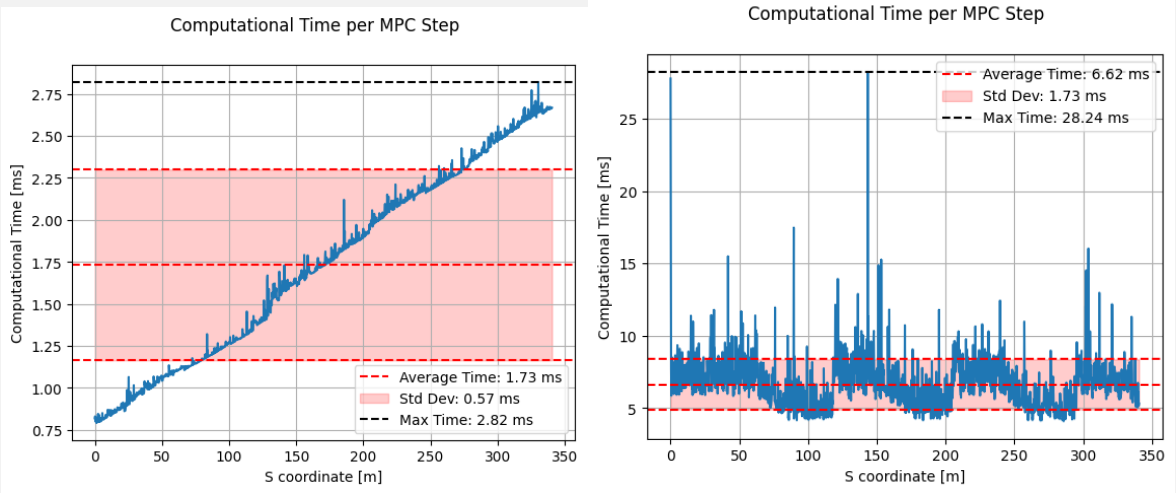
7.2

With the same controller weights $Q = \text{diag}(10, 1, 0.1, 0.1, 1)$, $R = \text{diag}(0.1, 0.1)$, terminal weight $Q_T = P$, no constraints, warm start enabled, and $N = 40$ from the initial state $x_0 = [0.9, -0.1, 0, 0, 0]$, both solvers produce control inputs that match LQR essentially exactly (the MPC–LQR difference remains small and looks like solver noise).
CLARABEL is clearly faster and more consistent: **avg 1.73 ms, std 0.57 ms, max 2.82 ms** per MPC step. MOSEK also meets the 10 Hz budget but is slower and spikier: **avg 6.62 ms, std 1.73 ms, max 28.24 ms**. We believe those spikes likely come from different internal scaling/termination criteria, not from control law differences.



(a) CLARABEL **(b)** MOSEK

Figure 10: Difference between LQR and MPC controller inputs for different solvers.

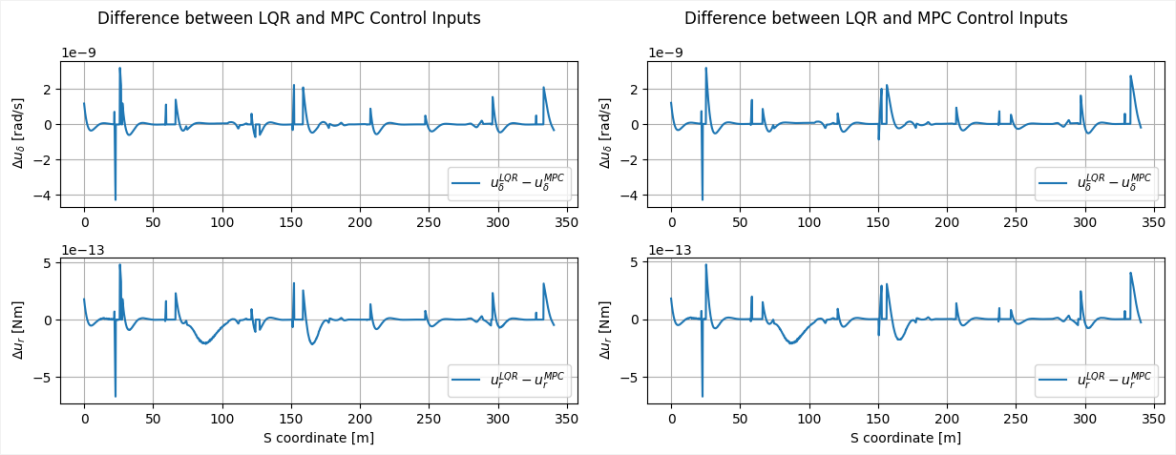


(a) CLARABEL **(b)** MOSEK

Figure 11: Computational Time per MPC step for different solvers.

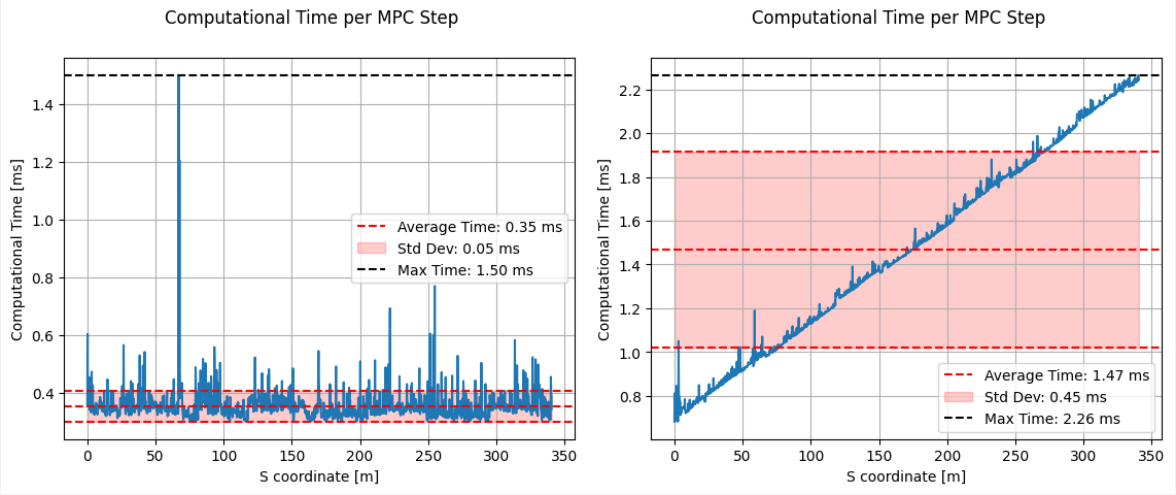
7.3

Using the same settings as before, the difference plots are identical, implying that warm starting changes speed but not the control law when constraints are inactive and $Q_T = P$. Timing is where the effect shows. In our runs, the warm-started case was actually slower, with higher average and maximum solve times compared to the cold start. This can occur because the solver's default initialization is already well suited for these problems, so enabling warm start does not provide an advantage and may even add some overhead.
The state-error plots, however, show a small advantage with warm start. Around curves, the warm-started run exhibits lower peaks and quicker decay in e_d, e_ψ, v_y, and r, and the profiles appear slightly smoother. In practice, this suggests that while warm starting did not improve computation time in our setup, it can still help smooth the transient response when the reference changes more abruptly or when constraints become active.



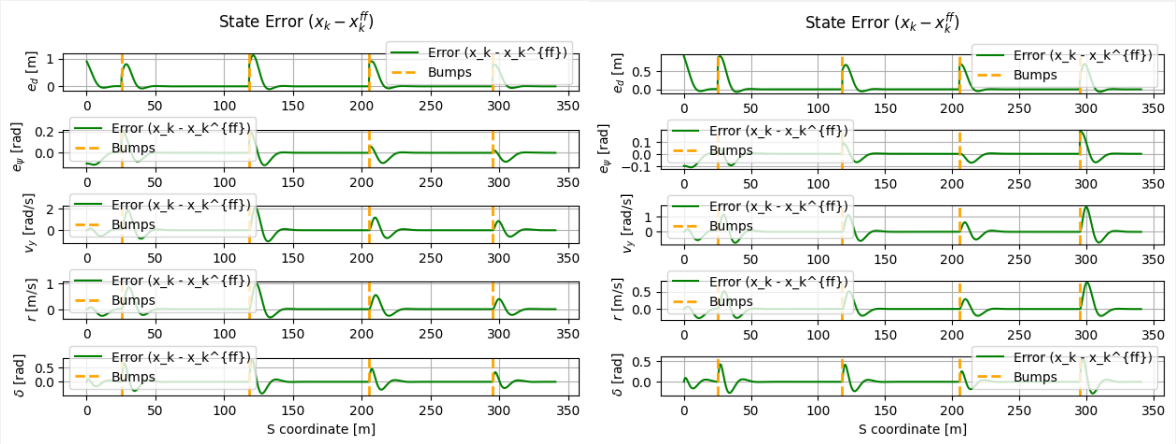
(a) Cold start **(b)** Warm start

Figure 12: Difference between LQR and MPC controller inputs for different starts.



(a) Cold start **(b)** Warm start

Figure 13: Computational Time per MPC step for different starts.



(a) Cold start **(b)** Warm start

Figure 14: State error for different starts.

```
#####################################
# Q7
#####################################

# Weights
Q = np.diag([10, 1, 0.1, 0.1, 1])
R = np.diag([0.1, 0.1])

# LQR and terminal weight
L = system.get_lqr_controller(Q, R)
P = system.get_lqr_cost_matrix(Q, R)    # QT = P
```

```
# Initial state from the Q3
x0 = np.array([0.9, -0.1, 0.0, 0.0, 0.0])

def run_case(N, solver='CLARABEL', warm_start=True):
    print(f"\n--- Q7: N={N}, solver={solver}, warm_start={warm_start}
        ---")
    mpc = MPC(model=system, N=N, Q=Q, R=R, Qt=P, warm_start=warm_start
        , solver=solver)
    mpc.setup_mpc_problem()
    simulate_system_run(
        x0=x0,
        system=system,
        mpc_controller=mpc,
        lqr_controller=L,
        racetrack=racetrack,
        raceline=raceline,
        ds=ds,
        x_ff=x_ff,
        u_ff=u_ff
    )

# 1) Horizon sweep
for N in [20, 40, 80]:
    run_case(N, solver='CLARABEL', warm_start=True)

2) Solver comparison at fixed N
run_case(40, solver='CLARABEL', warm_start=True)
try:
    run_case(40, solver='MOSEK', warm_start=True)
except Exception as e:
    print("MOSEK not available:", e)

3) Warm start vs cold start (same N & solver)
run_case(40, solver='CLARABEL', warm_start=True)
run_case(40, solver='CLARABEL', warm_start=False)
```