Model Predictive Control (H0E76A)

Exercise Session 2 | Linear MPC

leander.hemelhof@kuleuven.be
renzi.wang@kuleuven.be
brecht.evens@kuleuven.be

2024–2025

1 Introduction

The focus of this session will be the practical aspects of linear MPC. This mainly involves getting familiar with tools for building and solving convex optimization problems. One popular and very useful tool for prototyping such problems is called cvxpy[1]. The goal of today will be to get started with this tool and get some experience with linear MPC.

2 LQR using semidefinite programming

Consider again the LQR problem of the previous exercise session. There, we implemented an iterative algorithm for computing the horizon-N LQR controller using dynamic programming. Today, we will take this one step further and compute the infinite-horizon LQR solution.

To do so, we will formulate the LQR problem as a *semidefinite program* (SDP). These are convex optimization problems involving the constraint that some matrix which is linear in the decision variables is positive (or negative) semidefinite (hence the name semidefinite programming). Since these matrices are linear functions of the decision variables, the described constraints are also called *linear matrix inequalities* or LMIs. These problems can be formulated easily using cvxpy and efficient solvers exist for this problem class, which are ubiquitous in control applications.

2.1 Estimating the closed-loop cost of a given controller

In the homework assignment of the first exercise session, you are asked to estimate the infinite-horizon cost of your horizon N LQR controller by simply simulating it over a long horizon and summing the observed stage costs. Although this works to get a quick estimate, we can use semidefinite programming to compute this 'exactly' (up to tolerances of the SDP solver, of course)

To do so, consider again the setup of session 1. We have the dynamics

$$x_{t+1} = f(x_t, u_t) = Ax_t + Bu_t, \tag{1}$$

and the LQR cost associated with the linear control law $\kappa : x \mapsto Kx$

$$V_\infty(x_0, K) = \sum_{k=0}^{\infty} \ell(x_k, Kx_k), \tag{2}$$

where $\ell(x,u) = x^\top Qx + u^\top Ru$, with $Q \succeq 0$ and $R \succ 0$.

Suppose now that K is a fixed control gain that you want to evaluate. Our goal will be to compute $V_\infty(x_0, K)$. To do so, let's propose the quadratic candidate function $V_P(x) = x^\top Px$, where $P \succ 0$ is to be found. Now suppose that we know that we restrict our search to candidates V which are Lyapunov functions for the given controller, i.e., V_P satisfying

$$V_P(f(x, Kx)) - V_P(x) \leq -\ell(x, Kx), \ \forall x \in \mathbb{R}^n. \tag{3}$$

[1] https://www.cvxpy.org/

Exercise 1 | Let $(x_t)_{t\in\mathbb{N}}$ be a closed-loop state sequence satisfying $x_{t+1} = f(x_t, Kx_t)$, from $t = 0, \ldots, T-1$. Write down the sum of inequality (3) over this sequence of states and use this to show that this condition implies that V_P is an upper bound for V_∞.

The closest approximation is thus obtained by minimizing our upper bound:

$$\begin{aligned} \hat{V}(x_0) = \min_{P\succ 0} \quad & V_P(x_0) \\ \text{s. t.} \quad & V_P(f(x, Kx)) - V_P(x) \leq -\ell(x, Kx), \; \forall x \in \mathbb{R}^n. \end{aligned} \tag{4}$$

This problem is still not tractable numerically, since it has an infinite number of constraints (one for each x). However, let us write the constraint (3) more explicitly.

Exercise 2 | Write (3) explicitly as a function of P, K, A, B, Q, and R and show that it can be equivalently formulated in the form

$$\overline{A}^\top P\overline{A} - P + \overline{Q} \preceq 0, \tag{5}$$

for some $\overline{A}$ and $\overline{Q}$ (which don't depend on P).

Notice that the left-hand side of (5) is linear in P, and therefore, (5) is an LMI! Thus, we have finally obtained our SDP:

$$\begin{aligned} \hat{V}(x_0) = \min_{P\succ 0} \quad & V_P(x_0) \\ \text{s. t.} \quad & \overline{A}^\top P\overline{A} - P + \overline{Q} \preceq 0. \end{aligned} \tag{6}$$

Exercise 3 | Implement and solve the SDP using cvxpy. The general structure of such a cvxpy script is given below. Set $K = \begin{bmatrix} 1 & 2 \end{bmatrix}$ and $x_0 = (10, 10)$. All the other quantities are the same as in session 1. Compare the cost of this controller obtained from the SDP to the estimate of a long simulation. How long do you need to simulate to get an accurate estimate? Explain why this finite-horizon estimate is a good approximation.

```python
import cvxpy as cp
# Set up the numberical values
K = np.array([[1., 2.]])
Abar = ...
Qbar = ...
ns = ...  # State dimensions
x = ...   # Initial state
# cvxpy.org/api_reference/cvxpy.expressions.html#cvxpy.expressions.leaf.Leaf
P = cp.Variable((ns, ns), PSD=True)

cost = x.T@P@x  # Set the cost. Remember, P is the variable here!
# Hint: semidefinite constraints are implemented using `<<' and `>>'.
# `<=' and `>=' are element-wise!
constraints = [ ... ]  # Set the constraints
optimizer = cp.Problem(cp.Minimize(cost), constraints)  # build optimizer
solution = optimizer.solve()
```

3 Linear MPC using cvxpy

For the remainder of the session we will focus on implementing linear MPC, but now, we will add constraints.

We will continue experimenting with longitudinal control of the linearized vehicle model. In particular, we will cover the simple scenario of driving our vehicle from one intersection to the next over a straight road, as shown in fig. 1. By placing the origin of our coordinate system at the destination, we can model this problem as a regulation problem, (that is, we want to drive the state $\begin{bmatrix} p & v \end{bmatrix}^\top$ to the origin, where p, v denote the longitudinal position and velocity of the vehicle, respectively.)

For the vehicle dynamics, we will again use the double integrator model obtained in the previous session:

$$x_{k+1} = \begin{bmatrix} 1 & T_s \\ 0 & 1 \end{bmatrix} x_k + \begin{bmatrix} 0 \\ T_s \end{bmatrix} u_k$$

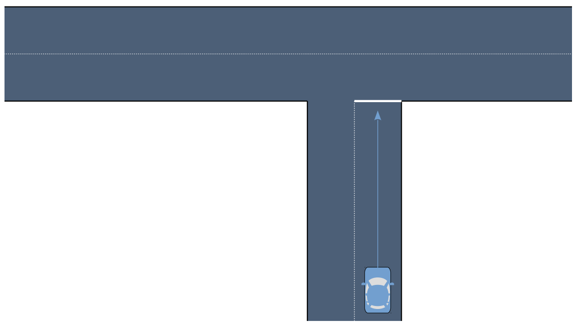

Figure 1: Problem setup

where we set $T_s = 0.3$. The input u is the longitudinal acceleration of the vehicle.

3.1 Exercises

Exercise 4 | Formulate this use case as a constrained finite-horizon optimal control problem. Use the standard quadratic cost, with $Q = \text{diag}(10, 1)$ and $R = 0.01$. Take into account the following constraints:

- The acceleration of the car is limited to $[u_{\min}, u_{\max}] = [-20, 10]$.
- The speed limit is $v_{\max} = 25\ m/s$.
- Overshoot past the stop line cannot exceed a small value $p_{\max} = 1\ m$ to guarantee that the vehicle doesn't accidentally cross the perpendicular road prematurely.

Exercise 5 | Implement this optimal control problem using cvxpy. Define the initial state x_0 as a parameter using `cvxpy.Parameter`. This way, you can prebuild the controller and rapidly solve the problem for different values of the initial state, as required for MPC purposes.

Hint. In order to define the quadratic stage costs, use the cvxpy atom `quad_form` instead of something like `x.T@P@x`. Otherwise, cvxpy may complain that the problem violates the disciplined convex programming (DCP) rules.

Hint. You can use the dataclass in `given.problem` to load the problem, so you don't have to type out the problem data.

Exercise 6 | Perform a closed-loop simulation of your MPC policy, with horizon $N = 5$ and $x_0 = [-100, 0]$. **Hint.** You can implement this yourself in the same way as you did in the previous exercise session, or you could simply use the `simulate` function from `rcracers.simulator`. It has essentially the same structure as the simulation code in the model solution for exercise 1. The usage is as follows

```python
from rcracers.simulator import simulate # Simulation function
from given.log import ControllerLog  # Data container for simulation data

def f(x, u): # Define the dynamics
    return problem.A @ x + problem.B @ u

logs = ControllerLog()
policy = ...
# `policy` should be a function (y, log) -> u
# taking (1) the current measured state and (2) a logging callback `log`
# and outputing a control action `u`.
# To add the state prediction `P` to the log, `policy' should
# execute the line
#   log("state_prediction", P)
# Note that the first argument must be a field of the `ControllerLog` class
# (see given/log.py)

x0 = ...  # initial_state
states = simulate(x0, f, n_steps=60, policy=policy, log=logs)
```

Exercise 7 | Now consider a slightly modified situation where the vehicle is not able to brake as effectively, e.g., due the road being wet. We may model this by setting $u_{\text{min}} = -10$ instead of -20. Now run the simulation again. What goes wrong and why?