Model Predictive Control (H0E76A)

Exercise Session 1 | Unconstrained Optimal Control

mathijs.schuurmans@kuleuven.be
leander.hemelhof@kuleuven.be

2025–2026

1 Introduction

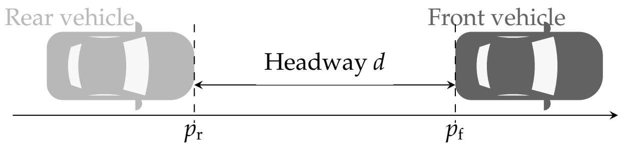


Figure 1

In this session, we will start experimenting with optimal control using a simple example from the automotive industry. Suppose we are interested in building a cruise control system, i.e., a system that automatically controls the throttle and brakes of our vehicle so that its velocity is as close as possible to some setpoint, all the while avoiding collisions with any vehicle that might be driving in front of it. Let us assume that we are working in a road-aligned coordinate system, and we only have to worry about longitudinal control (the steering wheel is still controlled by the human driver). For the sake of simplicity, let's also assume that we are in **traffic with a constant velocity**, which alleviates the need for a complicated prediction module for the preceding vehicle motion.

We can now define the following states for our dynamical system:

- $d \triangleq p_f - p_r - d_{ref}$: the headway (distance between the two vehicles), reduced by some desired value. Thus, $d = 0$ corresponds to the preferred safety distance, $d > 0$ means there is too much distance, $d < 0$ means we're too close.
- $v \triangleq v_f - v_r$: difference between the preceding vehicle's **(constant)** velocity v_f and the controlled vehicle's velocity v_r.

Exercise 1 | Let $x = \left[\begin{smallmatrix} d \\ v \end{smallmatrix}\right]$ denote the state vector and denote by u the acceleration of the rear car (which we will consider to be our control action). Write out the linear dynamics of this system in the form

$$\dot{x} = Ax + Bu. \tag{1}$$

Solution

Using basic kinematics, we can write $\dot{d} = v_f - v_r = v$ and because we have assumed constant velocity of the preceding vehicle, $\dot{v}_f = 0$. Thus, $\dot{v} = -\dot{v}_r = -u$. In matrix form, this becomes

$$\dot{x} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} x + \begin{bmatrix} 0 \\ -1 \end{bmatrix} u.$$

Exercise 2 | Discretize (1) using the forward Euler method[1] with a sample time T_s and write it in the form

$$x_{k+1} = Ax_k + Bu_k$$

Solution

The forward Euler update rule is given by $x_{k+1} = x_k + f(x_k, u_k)T_s$. Thus,

$$\begin{aligned} x_{k+1} &= x_k + T_s(Ax_k + Bu_k) \\ &= \begin{bmatrix} 1 & T_s \\ 0 & 1 \end{bmatrix} x_k + \begin{bmatrix} 0 \\ -T_s \end{bmatrix} u_k \end{aligned}$$

2 Finite horizon optimal control

We will now study the use of optimal control to drive the state of this system to the origin, that is: move to a steady state, in which we drive at the preceding vehicle's constant velocity at the desired distance from it.

Exercise 3 | For the discrete system obtained in Exercise 2, we will compute the (time-varying) optimal control law $\kappa_k(x) = K_k x$ (also referred to as a policy) that minimizes the following cost

$$V(x) = \min_{K_k} \sum_{k=0}^{N-1} (x_k^\top Q x_k + u_k^\top R u_k) + x_N^\top P_f x_N,$$

where $u_k = \kappa_k(x_k)$ and $x_{k+1} = Ax_k + Bu_k$ for all $k = 0, \dots, N-1$. We will use the following numerical values:

$$T_s = 0.5 \qquad C = [1, -2/3] \qquad Q = C^\top C + 0.001 I_{2\times 2} \qquad R = 0.1 \qquad P_f = Q.$$

As shown during the lectures, a solution for this problem can be derived either through dynamic programming or directly via the method of Lagrange multipliers. This yields a recursion in auxiliary matrices K_k and P_k. In this exercise, we will implement this solution numerically.

- Write out the backwards recursion relation for K_k and P_k derived during the lectures.

Solution

Following the derivation leading up to slide 3-30 of the lectures, we find that by writing the optimality conditions of the optimal control problem above, we can obtain K_k and P_k by the following backward-running recursion relations:

$$\begin{aligned} K_k &= -(R + B^\top P_{k+1} B)^{-1} B^\top P_{k+1} A \\ P_k &= Q + A^\top P_{k+1}(A + BK_k), \end{aligned}$$

which is initialized with $P_N = P_f$.

- Implement this recursion relation in Python.

Solution

See solution code.

- **(extra)** Derive the backwards recursion yourself.

[1]Since this is a linear system, you can of course also compute the exact discretization, (which you can do as an exercise) but for this session, forward Euler will suffice.

Solution

See slides 3-21 — 3-47.

Exercise 4 | Experiment with the resulting controller and the horizon length N:

- Simulate the closed-loop state trajectory in a **receding horizon** fashion starting from state $x = [\,10\;10\,]^\top$: at each time step, plot the predicted trajectory using the current state and the feedback matrices $K_0, \dots, K_{N-1}$ you obtain from the code you wrote for Exercise 3. Then, proceed to the next time step by using the actual feedback matrix K_0.

Solution

The procedure explained here corresponds to the diagram on slides 3-13 and 3-50 of the lectures. As mentioned there, for unconstrained linear systems, resolving the finite-horizon optimal control problem is redundant, since the solution of this problem yields a **constant** state feedback gain matrix K_0 (i.e., a linear controller $u = K_0 x$).

Warning

Simulating this controller in receding horizon fashion implies that at each simulation step t, you apply the **first** control action, i.e., $u_t = K_0 x_t$, not $u_t = K_t x_t$! The latter would not be equivalent to applying the solution of the N-horizon problem at each time step. Instead, it would be equivalent to solving the $(N - t)$-horizon optimal control problem at each time step t.

- Try this for multiple values of the horizon N and numerically look for the minimum horizon length N^* that stabilizes the system.
  Hint. For simplicity, we will simply detect instability through simulation: If the state norm $\|x_k\|$ exceeds some value (say 100), then we flag the system as unstable.

Solution

See the final plot in the solution code: the closed loop system is stable for $N \geq 5$.

- Intuitively, perhaps based on the plot of the prediction, motivate why increasing the horizon stabilizes the closed-loop system.

Solution

Two intuitive explanations are as follows.

1. For short horizons, the balance between the cost of the state trajectory and the input trajectory comes down in favor of the input. A dynamical system must have at least one integrator element, which delays the effects of the input on the state. This has as a result that for sufficiently small horizons, the effects of the choice of are not yet visible in the state within the prediction horizon. As an extreme example: in a one-step prediction the position is independent of the control action for the dynamics under consideration in this session. Therefore, the most effective way to decrease the cost would be to apply only very small inputs. If this were to lead to an unstable trajectory, this policy would cause the cost terms on the states to increase and eventually dominate the cost for larger horizons. This will force the inputs to increase in order to drive the states back to the origin.

2. A more straightforward argument would be that stability is an asymptotic

property and therefore inherently something that one can only evaluate over an infinite horizon. One could thus argue that penalizing the energy of the controls and states over a short horizon is therefore not a very accurate approximation of the goal we're trying to achieve. The longer the prediction horizon, the better our cost actually represents the desired behavior of the closed-loop system.

- Given a horizon length N^* that stabilizes the closed-loop system, can you be sure that the system will be stable for $N > N^*$?

Solution

Intuitively, one might presume that this should indeed hold. However, one can construct counterexamples showing that this is not the case. See code in the solution and the example in slide 3-61. In the particular case where the value function of the N^*-horizon problem is a Lyapunov function for the closed-loop system, then by monotonicity of the Bellman operator, such a result can be established. This, however, requires additional care in choosing the terminal costs and constraints, which is the subject of the next session.

Exercise 5 | Compare the finite horizon controller with the infinite horizon LQR controller.

- Implement the infinite horizon LQR controller $u = K_\infty x$.[2]
- Compare the simulated closed-loop trajectories of this controller with the trajectories of the controller of Exercise 3. What happens as you increase N?

Solution

The trajectories of the finite-horizon LQR controllers converge to those generated using the LQR controller (see solution code).

- Explain this observation based on the recursion relation used earlier and the discrete-time algebraic Riccati equation (DARE).

Solution

Suppose that the recursion relation converges. This means that as $k \to \infty$, $P_{k+1} = P_k = P$ and $K_{k+1} = K_k = K$, and thus

$$\begin{aligned} K &= -(R + B^\top P B)^{-1} B^\top P A \\ P &= Q + A^\top P (A + BK). \end{aligned}$$

Substituting the expression for K into expression for P, directly yields the discrete-time algebraic Riccati equation, the solution of which yields exactly the LQR controller.

- Approximate the infinite horizon cost for the closed-loop system numerically using a long state and input trajectory:

$$V_\infty = \sum_{k=0}^{\infty} (x_k^\top Q x_k + x_k^\top K^\top R K x_k) \approx \sum_{k=0}^{1000} (x_k^\top Q x_k + x_k^\top K^\top R K x_k).$$

[2]Use `solve_discrete_are` from `scipy.linalg`.

Solution

See the solution code: note that the infinite-horizon LQR controller will always have a lower closed-loop cost than the finite-horizon LQR controller, since it is the unique solution to the infinite horizon problem.

Appendix A Setting up your Python workspace

We will assume that you have either Python 3.8, 3.9 or 3.10 installed on your machine. Check your python version in your terminal using

```
    python --version
```

We recommend that you create a separate virtual environment for this course, to ensure that there are no conflicts with other installations of libraries you may have on your computer.

A.1 Creating a virtual environment

To create a virtual environment, open a terminal (or Powershell in Windows) in the workspace folder in which you plan to work. Then use:

```
    python3.x -m venv .venv
```

This will create a folder called `.venv` in your current folder. To activate the environment use

```
source .venv/bin/activate
```

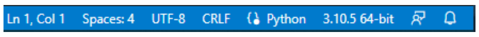
on Linux, or alternatively in Powershell:

```
.\.venv\Scripts\activate
```

You should see a `(.venv)` prompt at the start of the line in your terminal now. Make sure to activate your environment each time you want to execute Python scripts.

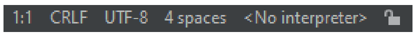
A.2 Setting up your IDE

If you use `VSCode` or `PyCharm` your environment can be handled automatically. In `VSCode`, after opening your workspace folder, click the area right of `Python` to select your interpreter (it is labeled `3.10.5 64-bit` below, but might be different depending on your Python installation).


Ln 1, Col 1   Spaces: 4   UTF-8   CRLF   {} Python   3.10.5 64-bit

Then select the Python executable in `(.venv)`. In Unix this is `.venv/bin/python` and in Windows it is `.venv\Scripts\python.exe`. This option should be available by default. If not you will have to browse to the correct file manually.

Similarly in PyCharm, after opening your workspace folder, you should click `<No interpreter>` depicted below:


1:1   CRLF   UTF-8   4 spaces   <No interpreter>

Then select `Add Interpreter`. The default option then should be to use an existing environment where your `.venv` is already selected.

A.3 Installing dependencies

To install the dependencies, make sure your environment is activated (i.e. you see `(.venv)` at the start of your terminal prompt). Then unzip `rcracers.zip` and execute

```
pip install <path-to-rcracers>
```

Now you should be set-up and ready to go.

Warning

You might encounter the following warning when installing rcracers:

```
fatal error: 'gmp.h' file not found
```

In that case, you need to first install GMP, which is a multiprecision arithmetic library. On Linux, this can be installed using

```
sudo apt-get install libgmp3-dev
```

and on MacOS using

```
brew install gmp
```