Model Predictive Control (H0E76A)

Exercise Session 3 | Terminal conditions, stability and recursive feasibility

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

2024–2025

1 Introduction

The exercises in this session build upon the same example as the previous session (see solution on Toledo). Recall that the controller from the previous session had a serious flaw: it had no guarantee of recursive feasibility or stability. As you observed in the final exercise, with some unfortunate settings, the controller could reach a state where it was no longer possible to find an input such that all the constraints were satisfied. The reason for this is that no terminal conditions were specified. The goal in this session is to construct a terminal constraint and a terminal cost that guarantee that the controller will remain feasible and stable.

To make this a bit easier, we provide a simple python module dedicated to representing and manipulating Polyhedra, which is part of the `rcracers` library. The sources can be found in `rcracers/utils/geometry/polyhedron.py`. See section 4.2 for more info on how to use the provided code.

2 Computing terminal conditions for linear systems

Consider the cost of the finite horizon optimal control problem from an initial state $x_0 = x$,

$$V_N(x) = \min_{u_0 \dots u_{N-1}} \sum_{k=0}^{N-1} \bar{\ell}(x_k, u_k) + \bar{V}_f(x_N),\ x_{k+1} = Ax_k + Bu_k,\ k = 0, \dots, N-1. \tag{1}$$

Here, all functions are extended-real valued, meaning that all **constraints** are encoded as their domains. For convenience, let us denote $Z = \operatorname{dom} \ell = X \times U \subset \mathbb{R}^{n_x+n_u}$. The sets X and U represent the sets of states and inputs that satisfy any constraint that we may impose.

Notice that V_N is exactly the value function obtained from N steps of *Value Iteration* (VI), starting from V_f, that is

$$V_{k+1} = \mathrm{T}\, V_k,\ V_0 = V_f,\ k = 0, \dots, N-1.$$

As discussed during the lectures, the key to establishing stability and recursive feasibility through dynamic programming lies in the connection with the infinite horizon problem, which has several nice properties that may be inhereted in the finite-horizon case through careful design choices. A crucial point is to guarantee that the VI converges. To do so, we need to choose the initial conditions V_f and X_f so that $\mathrm{T}\, V_f(x) \leq V_f(x),\ \forall x \in X_f$, and by **monotonicity** of the Bellman operator T, convergence of VI can be shown. This means that V_f is a control Lyapunov function See the slides of lecture 4 for more details. In particular 4–37.

Typically, finding such a control Lyapunov function is no simple task. For a linear system, however, we are in luck, as we can analytically obtain the infinite-horizon cost of the **unconstrained** problem: $V_{\mathrm{LQR}}(x) = x^\top P x$, where P is the solution to the algebraic Riccati equation. Since this is the optimal infinite-horizon cost, we know for all x, $\mathrm{T}\, V_{\mathrm{LQR}}(x) = V_{\mathrm{LQR}}(x)$. We'd like to use this information in the **constrained** case as well.

Exercise 1 | Formulate the defining condition on the set X_f of states under which the infinite-horizon cost of the unconstrained problem corresponds to the infinite horizon cost of the constrained optimal control problem: $\forall x \in X_f,\ V_{\mathrm{LQR}}(x) = V_\infty(x)$.

Exercise 2 | Consider the problem set-up we used in the previous session:

$$\begin{aligned} Q = \text{diag}([10,1]),\ R = 0.01 & \\ x_{k+1} = \begin{bmatrix} 1 & T_s \\ 0 & 1 \end{bmatrix} x_k + \begin{bmatrix} 0 \\ T_s \end{bmatrix} u_k, & \qquad k = 0 \dots N-1 \\ [-120, -50] \leq x_k \leq [1, 25]^\top, & \qquad k = 0 \dots N \\ -20 \leq u_k \leq 10, & \qquad k = 0 \dots N \end{aligned}$$

- Express the feasible set X as a polytope: $Hx \leq h$ and define it in python using the command

```
X = Polyhedron.from_inequalities(H, h)
```

- You can now plot this set in state space using

```
from given.visualization import plot_polytope
plot_polytope(X)   # you can provide axes, labels, etc. as usual in matplotlib
```

Exercise 3 | Because of the simplicity of the problem (only two states and only linear dynamics), computing the terminal set X_f you formulated in Exercise 1 is computationally tractable. Use the procedure explained in Section 4 below to compute this set. Here, it is advisable to call `canonicalize` on every iteration (Argue why!).

- Rewrite the input constraints in terms of x.
- Add these constraints to the state constraints to form one polyhedral set (similarly to Exercise 2).

Exercise 4 | Visualize the terminal set you obtained in state space. What drawback do you see to this approach?

Exercise 5 | Impose these terminal conditions in the situation with the bad road conditions from the previous session ($u_{min} = -10$).

Exercise 6 | Set $x_0 = (-4, 0)$ and set $N = 1$ and run the controller again. What do you observe? Is this what you expect?

3 Omitting the terminal cost

In practice, one might care more about having a large region of attraction, than having theoretical guarantees of stability. Therefore, techniques will be covered in the lectures that allow one to find a terminal cost, which is a control Lyapunov function in the largest possible region in state space.

So, ignoring stability, we might simply want to construct the maximum invariant set, and impose this as a terminal constraint, such that we have the largest possible operation range.

For the simple situation at hand, we can – similarly as before – iteratively construct the maximal positive invariant set.

Exercise 7 | Compute the maximum control invariant set in Python. Since we don't assume a specific local controller anymore, we now compute the pre-set as:

$$\begin{aligned} \text{pre}(X) &= \{x \ : \ \exists u \in U,\ Ax + Bu \in X\} \\ &= \{x \ : \ \exists u \in U,\ HAx + HBu \leq h\} \\ &= \left\{x \ : \exists u,\ \begin{bmatrix} HA & HB \\ 0 & G \end{bmatrix} \begin{bmatrix} x \\ u \end{bmatrix} \leq \begin{bmatrix} h \\ g \end{bmatrix}\right\}, \end{aligned}$$

assuming that $U = \{u : Gu \leq g\}$. This is a projection of a 3D polytope onto the first two dimensions, which we can compute using the provided code:

```
P.coordinate_projection([0,1])  # P is the 3D Polyhedron
```

Warning

Depending on your implementation, you may run into numerical inconsistencies when doing polyhedral computations. In this case, it turns out to be beneficial to include the intersection already in the computation of pre, before the projection. In general, however, this is a drawback inherent to working with polyhedral sets, which is very diffcult to solve.

Exercise 8 | Impose this terminal constraint now and observe how the system behaves for horizon $N \in \{1, 5, 10\}$.

Hint. Due to the many constraints added by the terminal set, not all QP solvers will successfully solve this problem in a reasonable amount of time. For this problem, it is recommended to use the ECOS solver. To do so, pass the argument `solver=cp.ECOS` when calling `solve()` on your problem.[1]

Do you notice anything in particular, and how can you explain this?

4 Tips on computing invariant sets

4.1 Basic procedure

Imagine we want to compute an invariant set $\Omega \subset X$ that lies within the feasible set X to ensure recursive feasibility. Algorithm 1 gives a simple iterative procedure to compute this (for small problems). The pre-set of a set X is the set of points x, that can reach X in one step. So for an autonomous system, $\text{pre}(X) = \{x : f(x) \in X\}$ and for a system with exogenous inputs, $\text{pre}(X) = \{x : \exists u \ : \ f(x, u) \in X\}$. So in a nutshell, Algorithm 1 initializes Ω to be equal to the feasible set (since this is the largest possible invariant set), then only keeps the points within Ω that again land in Ω after one time step.

Algorithm 1 Computing invariant sets

1: $\Omega_0 \leftarrow X$
2: **loop**
3: $\quad \Omega_{i+1} \leftarrow \text{pre}(\Omega_i) \cap X$
4: $\quad$ **if** $\Omega_{i+1} = \Omega_i$ **then**
5: $\quad\quad$ **return** Ω_{i+1}
6: $\quad$ **end if**
7: **end loop**

4.2 Representing polyhedral sets in Python

Use the `from_inequalities` function to construct a polyhedron from a system of linear inequalities:

$$X = \{x \in \mathbb{R}^n \mid Hx \leq h\}$$

For example, to represent the system

$$\begin{cases} x_1 \geq -1 \\ x_2 \geq -1 \\ x_1 + x_2 \leq 1 \end{cases}$$

you would write

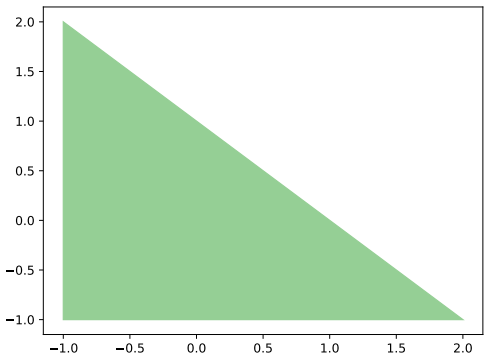
```
from rcracers.utils.geometry import Polyhedron
import numpy as np

H = np.vstack([-np.eye(2), np.ones((1,2))])
h = np.array([1.,1., 1.])
poly = Polyhedron.from_inequalities(H, h)
```

[1] see https://www.cvxpy.org/tutorial/advanced/index.html#choosing-a-solver

To visualize a polytope you can use the utility functions in `visualization`.

```
from rcracers.utils.geometry import plot_polytope
import matplotlib.pyplot as plt

# basic usage
plot_polytope(poly)

# Or with some styling
plt.figure()

plot_polytope(poly,
    color="tab:green", # Set the base color
    alpha=0.5,         # Set the transparency
)

plt.show()
```

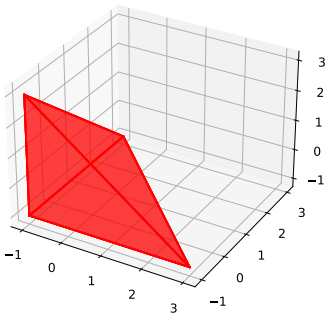
Which outputs:



It is also possible to plot 3d polytopes.

```
"""3d plot"""
import matplotlib.pyplot as plt
from rcracers.utils.geometry.polyhedron import Polyhedron
import numpy as np

H = np.vstack([-np.eye(3), np.ones((1,3))])
h = np.concatenate([np.ones(3), [1.]])
poly = Polyhedron.from_inequalities(H, h)

from rcracers.utils.geometry import plot_polytope

# basic usage
plot_polytope(poly)

plt.savefig("poly3d.pdf")
plt.show()
```

Which outputs:



To get the half-spaces back out, call `poly.H` and `poly.h`.

Warning

Very important: the method

```
Polyhedron.canonicalize
```

will remove redundant inequalities from the representation and will merge double-sided inequalities with opposing normals and equal offsets to single equalities , e.g.,

$$1 \leq x \leq 1 \rightarrow x = 1.$$

The equality constraints are represented as $H_{\text{eq}}x \leq h_{\text{eq}}$, which can be queried as `poly.H_eq` and `poly.h_eq`. Although this can become expensive in higher dimensions, it is advisable to use this method often, to ensure that the representation of the polyhedron remains numerically well-conditioned.
Other useful methods are:
`poly.intersect(opther)` Compute the intersection between `poly` and `other`.
`poly.coordinate_projection(coord)` Project a polyhedron on the provided coordinates.