EGH445 - Modern Control

Discrete-Time Control Design 3

Optimal Control

Overview

- Quick review of (some of) the content so far.
- When does pole placement work well? And when does it fail?
- What is *Optimal* Control?
- Why should we do optimal control instead of just pole placement?
- The Optimal Control Problem.
- The Linear Quadratic Regulator (LQR).
- Model Predictive Control (MPC).

Quick review of (some of) the content so far

Continuous-time system:

$$\dot{x}(t) = f(x, u),$$
$$y(t) = g(x, u)$$

Linearised system:

$$\delta\dot{x}(t) = A\delta x(t) + B\delta u(t),$$
$$\delta y(t) = C\delta x(t) + D\delta u(t),$$

where $\delta x = x - \bar{x}$, $\delta u = u - \bar{u}$, $\delta y = y - \bar{y}$, and the A, B, C, D are the matrices

$$A = \left.\frac{\partial f}{\partial x}\right|_{\bar{x},\bar{u}} \quad B = \left.\frac{\partial f}{\partial u}\right|_{\bar{x},\bar{u}} \quad C = \left.\frac{\partial g}{\partial x}\right|_{\bar{x},\bar{u}} \quad D = \left.\frac{\partial g}{\partial u}\right|_{\bar{x},\bar{u}}$$

Discrete-time system:

$$x(kT+T) = Gx(kT) + Hu(kT),$$
$$y(kT) = Cx(kT) + Du(kT),$$

where

$$G = e^{AT}, \quad H = \left[\int_0^T e^{A\tau} d\tau\right] B \quad \left(\text{or, if } A \text{ is invertible, } H = A^{-1}(G-I)B\right)$$

State-feedback controller: $u(kT) = -Kx(kT)$,

where K is the feedback gain matrix that is designed to arbitrarily move the poles of the closed-loop system

$$x(kT+T) = (G - HK)x(kT)$$

This is achieved, for example, by equating the characteristic polynomial of the closed-loop system to a desired polynomial,

$$\det(zI - (G - HK)) = (z - z_1)(z - z_2) \cdots (z - z_n)$$

> **Important**
>
> The solution exists if the system is controllable, i.e. if the **controllability matrix** $\mathcal{C} = \begin{bmatrix} H, & GH, & G^2H, & \ldots, & G^{n-1}H \end{bmatrix}$ has full rank ($\text{rank}(\mathcal{C}) = n$).

How do you choose the poles?

- t_s - settling time
- t_r - rise time
- $\%OS$ - percent overshoot
- ω_n - natural frequency
- ζ - damping ratio (or through percent overshoot)

- $\zeta = \dfrac{\ln(\%OS/100)}{\sqrt{\pi^2 + \ln^2(\%OS/100)}}$
- $\omega_n = \frac{4}{\zeta t_s}$
- $s_{1,2} = -\zeta\omega_n \pm j\omega_n\sqrt{1-\zeta^2}$
- $z_{1,2} = e^{s_{1,2}T}$

We also saw the **Internal Model Principle** (e.g. **Integral Action**) to reject disturbances (or follow references) with a known model (e.g. a step input, a ramp input, a sinusoidal input, etc.). Those still relied on the **pole placement** approach.

Limitations of Pole placement

Works well for:

- SISO systems (Single Input Single Output)
- Low order systems (2nd order or when 2nd order *dominant*, etc.)

Does not work well for:

- **Higher-order systems** (4th order, 5th order, etc.)
- **MIMO systems** (Multiple Input Multiple Output)
- *Stiff* systems (e.g. with very different time constants, e.g. 1ms and 1s)
- **Highly nonlinear systems** (i.e. linearisation quickly becomes invalid.)

Higher-Order Systems

$$x(kT+T) = Gx(kT) + Hu(kT),$$
$$y(kT) = Cx(kT) + Du(kT),$$

where $x(kT) \in \mathbb{R}^n, \quad u(kT) \in \mathbb{R}, \quad y(kT) \in \mathbb{R}, \quad n > 3$

Important

The link between pole locations and desired time-domain response becomes less clear. Arbitrary choices can lead to poor performance or excessive control effort.

How do you choose the best pole locations for a 5th, 10th, or higher-order system?

MIMO Systems

Let's start with an example.

Consider a simple MIMO system with two states and two inputs:

$$x(kT+T) = \begin{bmatrix} 1 & 0.1 \\ 0 & 0.8 \end{bmatrix} x(kT) + \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} u(kT)$$

We want to place the poles at 0.4 and 0.6, by using $u(kT) = -Kx(kT)$, where $K = \begin{bmatrix} k_1 & k_2 \\ k_3 & k_4 \end{bmatrix}$.

The characteristic polynomial is given by:

$$\det(zI - (G - HK)) = \det\left(\begin{bmatrix} z - 1 + k_1 & k_2 - 0.1 \\ k_3 & z - 0.8 + k_4 \end{bmatrix}\right)$$
$$= z^2 + (k_1 + k_4 - 1.8)z + (0.1k_3 - k_4 - 0.8k_1 + k_1k_4 - k_2k_3 + 0.8)$$

The desired characteristic polynomial is given by:

$$(z-0.4)(z-0.6) = z^2 - 1z + 0.24$$

When we equate the coefficients, we get:

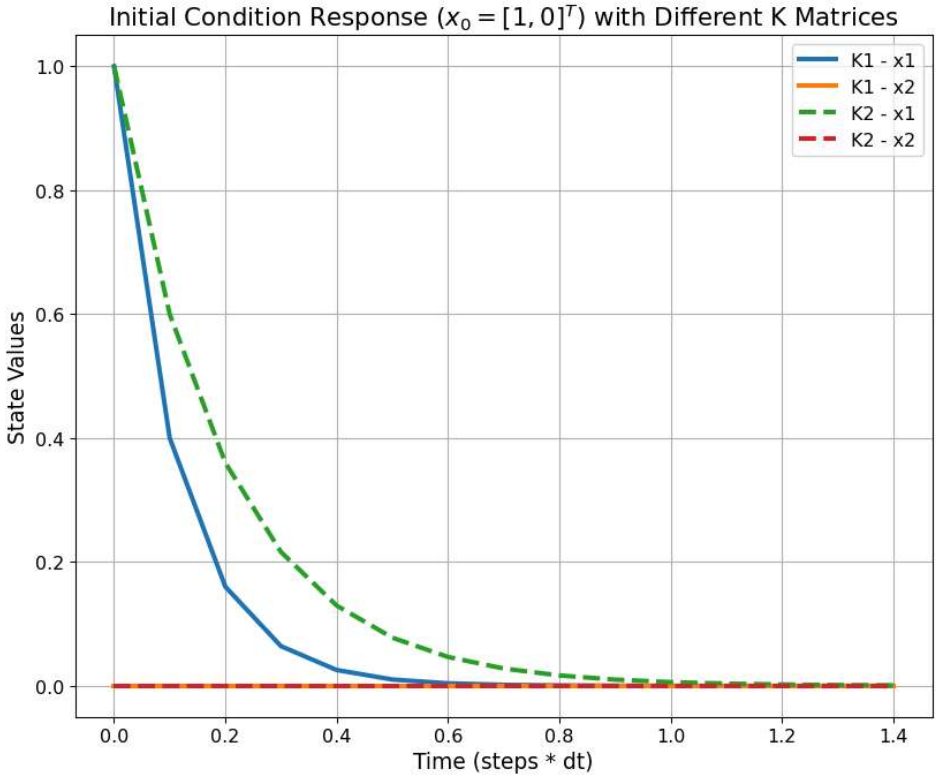
$$\begin{cases} (k_1 + k_4 - 1.8) = -1 \\ 0.1k_3 - k_4 - 0.8k_1 + k_1k_4 - k_2k_3 + 0.8 = 0.24 \end{cases}$$

> **Important**
>
> Note that we have **two equations** and **four unknowns**. This means that we have **degrees of freedom** in the design. We can choose two of the four variables arbitrarily, and then solve for the other two.

This seems like a good idea, but it is not. The problem is that different choices of K, even if they lead to the same **eigenvalues**, can lead to different **eigenvectors**. The closed-loop system's **eigenvectors** directly affect the **response** of the system.

Let $k_2 = k_3 = 0$ and consider the following two cases of gains matrices K_1 and K_2, both of which lead assign the closed-loop eigenvalues to the desired values:

```python
import numpy as np
import matplotlib.pyplot as plt
import control as ctrl
# Create the system matrices
G = np.array([[1, 0.1], [0, 0.8]])
H = np.array([[1, 0], [0, 1]])
C = np.array([[1, 0], [0, 1]]) # Outputs are the states
D = np.array([[0, 0], [0, 0]])
K1 = np.array([[0.6, 0], [0, 0.2]])
K2 = np.array([[0.4, 0], [0, 0.4]])
Gcl1 = G - H @ K1
Gcl2 = G - H @ K2

Ts = 0.1 # Sampling time (seconds)
# Create the closed-loop LTI systems
sys1 = ctrl.ss(Gcl1, H, C, D, Ts)
sys2 = ctrl.ss(Gcl2, H, C, D, Ts)

# Simulate for initial condition x0 = [1, 0]
x0 = np.array([1, 0])
t_end = 1.5
t = np.arange(0, t_end, Ts)
t1, y1 = ctrl.initial_response(sys1, T=t, X0=x0)
t2, y2 = ctrl.initial_response(sys2, T=t, X0=x0)

```



MIMO Systems

$$x(kT+T) = Gx(kT) + Hu(kT),$$
$$y(kT) = Cx(kT) + Du(kT),$$

$$x(kT) \in \mathbb{R}^n, \quad u(kT) \in \mathbb{R}^m, \quad y(kT) \in \mathbb{R}^p, \quad n, m, p > 1$$

Important

For MIMO systems, pole placement is significantly more complex. Specifying only the eigenvalues leaves **degrees of freedom** in the **eigenvectors**, which also **affect the response**. The design process becomes **non-unique** and less intuitive.

How do you systematically handle interactions between different inputs and outputs?

Performance trade-off

Finally, pole placement does not consider the **control effort**, not addressing the trade-off between:

- regulating the state, by making $x(kT)$, or $\delta x(kT)$[1], small

  [1] For linearised systems, we defined $\delta x = x - \bar{x}$.

- *regulating* required control effort $u(kT)$.

> **Important**
>
> You might achieve **desired poles** but with impractically **large control signals**.
>
> Which, like in the following example, can lead to **instability**.

Linear Controller Failure for a Nonlinear System

Consider a **scalar** system with **cubic nonlinearity**, $\dot{x}(t) = -x(t) + x(t)^3 + u(t)$.

$-x$ represents a stabilizing linear dynamic.

x^3 is a destabilizing nonlinearity that becomes significant for larger values of $|x|$.

We want to design a controller $u(kT)$ to stabilize the system at $\bar{x} = 0$ (requiring $\bar{u} = 0$).

1. Linearise the system around the equilibrium point $(\bar{x} = 0, \bar{u} = 0)$:

$$A = \frac{\partial}{\partial x}(-x + x^3 + u)\Big|_{x=0,u=0} = (-1 + 3x^2)\Big|_{x=0} = -1$$

$$B = \frac{\partial}{\partial u}(-x + x^3 + u)\Big|_{x=0,u=0} = 1$$

The linearised continuous-time system is $\boxed{\delta\dot{x}(t) = -\delta x(t) + \delta u(t)}$.

2. Discretise the system with a sampling time $T = 0.1$s:

$$x(kT + T) = Gx(kT) + Hu(kT), \quad y(kT) = x(kT)$$

where $G = e^{-AT} = 0.905$, and $H = A^{-1}(G - I)B = 0.095$.

The discrete-time (linearised) system is $\boxed{x(kT + T) = 0.905x(kT) + 0.095u(kT)}$.

3. Design $u(kT) = -Kx(kT)$ to place the closed-loop pole of this *linearised* system at $p = 0.5$.

3.1 Design through pole placement. The closed-loop is $x(kT + T) = (G - HK)x(kT)$.

We want the closed-loop pole to be equal to $p = 0.5$:

Try to do this through equating the characteristic polynomials: $\det(zI - (G - HK)) = (z - p)$.

$$
\begin{aligned}
0.905 - 0.095K &= 0.5 \\
0.095K &= 0.905 - 0.5 = 0.405 \\
K &= \frac{0.405}{0.095} = 4.263
\end{aligned}
$$

So, the linear controller designed for the linearized system is $\boxed{u(kT) = -4.263x(kT)}$.

4. Test the controller on the original nonlinear system.

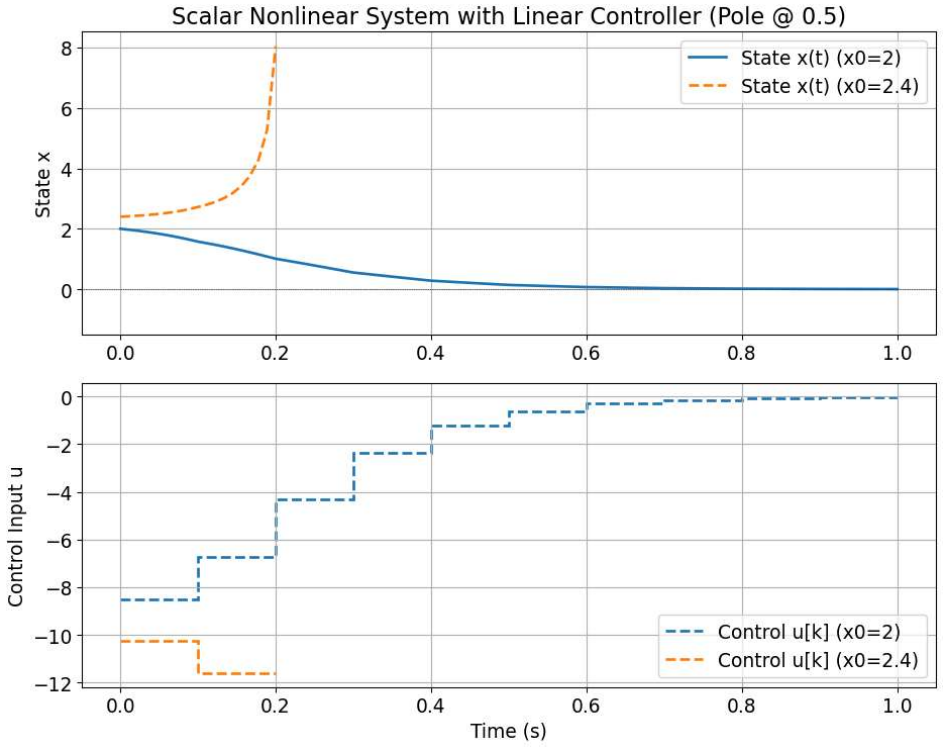
Simulate the continuous nonlinear dynamics, but apply the control input $u(kT) = -Kx(kT)$ constant over the period $[kT, kT + T)$ (Zero-Order Hold).

Important

Note that the controller was derived for the **linearised** system, which **ignores** the x^3 term. The linear model is only accurate for small x (close to the equilibrium $x = 0$). When $|x|$ becomes large, the x^3 term can dominate, and the linear controller may fail.

```python
import numpy as np
from scipy.integrate import solve_ivp
import matplotlib.pyplot as plt

# Set default font size for plots
plt.rcParams['font.size'] = 14

# 1. Nonlinear System Dynamics
def nonlinear_system(x, u):
    # Scalar system dynamics: dx/dt = -x + x^3 + u
    return -x + x**3 + u

# 2. Linear Controller Parameters (derived above)
Ad = 0.905
Bd = 0.095
K = 4.263
Ts = 0.1
x_eq = 0.0 # Equilibrium state

# 3. Simulation Setup
t_start = 0
t_end = 1.0 # Shorter simulation time might be enough
n_steps = int(t_end / Ts)

# Initial Conditions to compare
```

```
Simulation failed at step 2 (t=0.20) for x(0)=2.4
```



Optimal Control

What is optimal?

Optimal means **best**. But best in terms of what?

- **Best** in terms of *performance*?
- **Best** in terms of *low control effort*?
- **Best** in terms of a ***(generalised) cost***?

> **Key Idea:**
>
> Instead of choosing pole locations, you define what constitutes good **performance** via a **cost function** J.

The Discrete-Time Optimal Control Problem

Consider the **System Dynamics**[1], $x(kT+T) = Gx(kT) + Hu(kT), \quad x(0) = x_0.$

[1]The general concept extends to nonlinear systems, but through methods that are beyond the scope of this course.

We want to find $u(kT) = -Kx(kT)$ that minimizes the **Cost Function / Performance Index**[2]:

$$J = \sum_{k=0}^{\infty} \left(x(kT)^T Q x(kT) + u(kT)^T R u(kT) \right), \quad Q \succeq 0, \quad R \succ 0$$

[2] This is an *infinite horizon* cost function, where the **state penalty matrix** Q and the **control penalty matrix** R define the cost.

The solution to this problem is the Linear Quadratic Regulator (LQR).

> **Additional Reading**
>
> For more information about the general optimal control problem, see the additional readings on Canvas.

$A \succeq 0$ means that A is positive semi-definite, i.e. $x^T A x \geq 0$ for all $x \in \mathbb{R}^n$.

$A \succ 0$ means that A is positive definite, i.e. $x^T A x > 0$ for all $x \in \mathbb{R}^n$ and $x \neq 0$.

The Discrete Linear Quadratic Regulator (DLQR)

The gain K that minimizes the cost function[1], subject to the system dynamics[2] is given by:

$$K = (R + H^\intercal P H)^{-1} H^\intercal P G,$$

which depends on the *unique, symmetric, positive semi-definite* solution P of the **Discrete Algebraic Riccati Equation (DARE)**:

$$P = G^\intercal P G - (G^\intercal P H)(R + H^\intercal P H)^{-1}(H^\intercal P G) + Q,$$

Remember that P **must** be *unique, symmetric, positive semi-definite*, that is $P = P^\intercal \succeq 0$.

> **Note**
>
> The full derivation of the optimal control gain K formula is beyond the scope of the course. If you're curious, start by investigating the **Bellman's Principle of Optimality (Dynamic Programming)**.

1. $J = \sum_{k=0}^{\infty} \left(x(kT)^T Q x(kT) + u(kT)^T R u(kT) \right), \quad Q \succeq 0, \quad R \succ 0$
2. $x(kT + T) = Gx(kT) + Hu(kT), \quad x(0) = x_0$

Design of DLQR Controllers

The design of DLQR controllers is based on the following steps:

1. Check controllability of $x(KT+T) = Gx(kT) + Hu(kT)$.
2. Select the cost matrices $Q \succeq 0$ and $R \succ 0$.

   $J = \sum_{k=0}^{\infty} \left(x(kT)^T Q x(kT) + u(kT)^T R u(kT) \right)$
3. Solve the DARE to find the **positive semi-definite symmetric** P.
4. Compute the gain matrix $K = (R + H^\intercal P H)^{-1} H^\intercal P G$.
5. Implement the controller $u(kT) = -Kx(kT)$.
6. Test the controller performance and adjust Q and R as necessary.

Existence and Stability of the Solution

We need to check if the solution to the DARE exists and is stable.

Questions

- Does a *unique* solution to the DARE always exist?
- Does the resulting controller guarantee stability?

Turns out that we need the following conditions.

Conditions for Stabilizing DLQR Solution:

- **Controllability**: Can the controller move all unstable poles of the system?
- **Observability**: Can all unstable models be observed **by the cost function**?

The DLQR Stability Theorem

Theorem: DLQR Stability Theorem

Assume $R \succ 0$ and $Q \succeq 0$. Let $Q = V^T V$ (where V might not be unique).

If the following conditions hold:

1. The pair (G, H) is controllable.
2. The pair (G, V) is observable[1].

[1]Note that this is not the same as the observability of the system, as what's important is observability of states **by the cost function**.

Then:

1. A unique solution $P = P^\intercal \succeq 0$ to the DARE exists.
2. The resulting controller closed-loop system is stable.

Simple Example of DLQR Design

Let’s consider a simple example of a DLQR design and solve it “by hand”. Consider again the scalar system with cubic nonlinearity and its discrete linearised version,

$$\dot{x}(t) = -x(t) + x(t)^3 + u(t) \quad \rightarrow \quad x(kT+T) = 0.905x(kT) + 0.095u(kT)$$

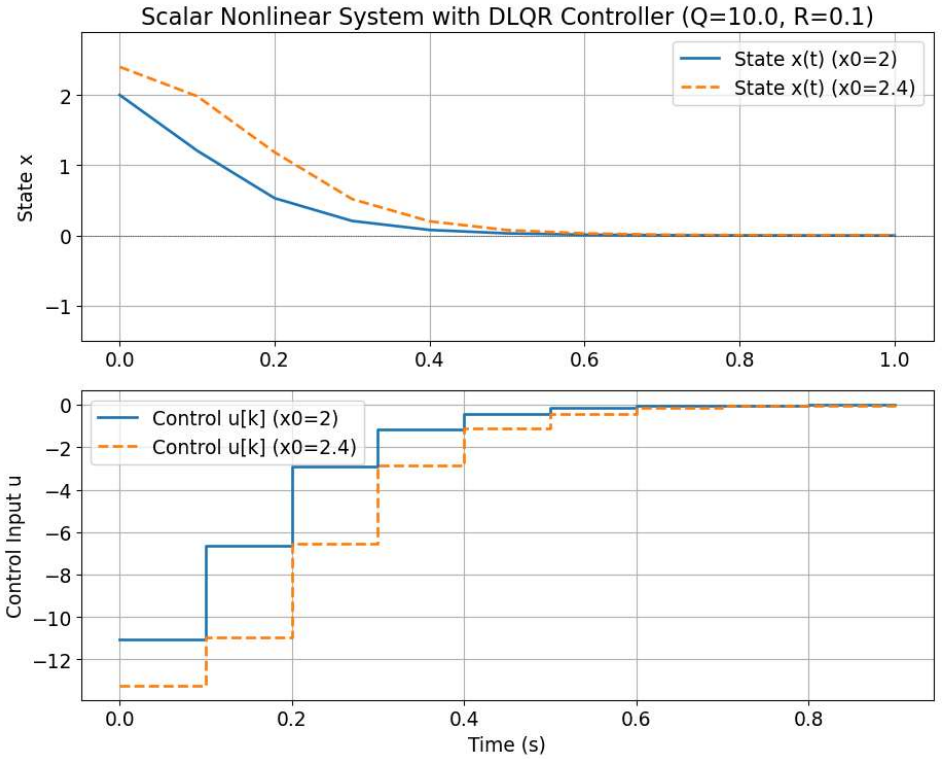
- **Controllability**: $\mathcal{C} = [H,\ GH] = [0.095,\ 0.086475]$ has full rank ($\text{rank}(\mathcal{C}) = 1$).
- **Cost Matrices**: Select $Q = 10.0$ and $R = 0.1$.
- **Observability**: $\mathcal{O} = [V,\ VG]^\intercal$, where $V = \sqrt{Q}$. Then $\mathcal{O} = [3.162,\ 2.857]^\intercal$, which has full rank.
- **Solution $P = P^\intercal \succeq 0$ of the DARE**: $P - G^\intercal PG + (G^\intercal PH)(R + H^\intercal PH)^{-1}(H^\intercal PG) = Q$

$$
\begin{aligned}
P - G^2P + \frac{(GHP)^2}{R + H^2P} &= Q \\
(P - G^2P)(R + H^2P) + G^2H^2P^2 &= Q(R + H^2P) \\
PR + H^2P^2 - G^2PR - QR - QH^2P &= 0 \\
(H^2)P^2 + (R - G^2R - QH^2)P + (-QR) &= 0 \\
0.009P^2 - 0.0722P - 1 &= 0 \\
P = 15.2571 \text{ or } P &= -7.2624
\end{aligned}
$$

- **Pick the *positive semi-definite* solution**: $P = 15.2571$.
- **Compute the gain** $K = (R + H^\intercal PH)^{-1}H^\intercal PG = 5.519$.

```python
import numpy as np
from scipy.integrate import solve_ivp
from scipy.linalg import solve_discrete_are # Import DARE solver
import matplotlib.pyplot as plt

# Set default font size for plots
plt.rcParams['font.size'] = 14

# 1. Nonlinear System Dynamics
def nonlinear_system(x, u):
    # Scalar system dynamics: dx/dt = -x + x^3 + u
    return -x + x**3 + u

# 2. Linear Controller Parameters (derived above)
Ad = 0.905
Bd = 0.095
K = 4.263
Ts = 0.1
x_eq = 0.0 # Equilibrium state

# 3. Simulation Setup
t_start = 0
t_end = 1.0 # Shorter simulation time might be enough
n_steps = int(t_end / Ts)

```

```
DARE Solution S = 15.257 (for Q=10.0, R=0.1)
DLQR Gain K = 5.519
Linear Closed-Loop Pole = 0.381. (STABLE)
```



MIMO Example of DLQR Design

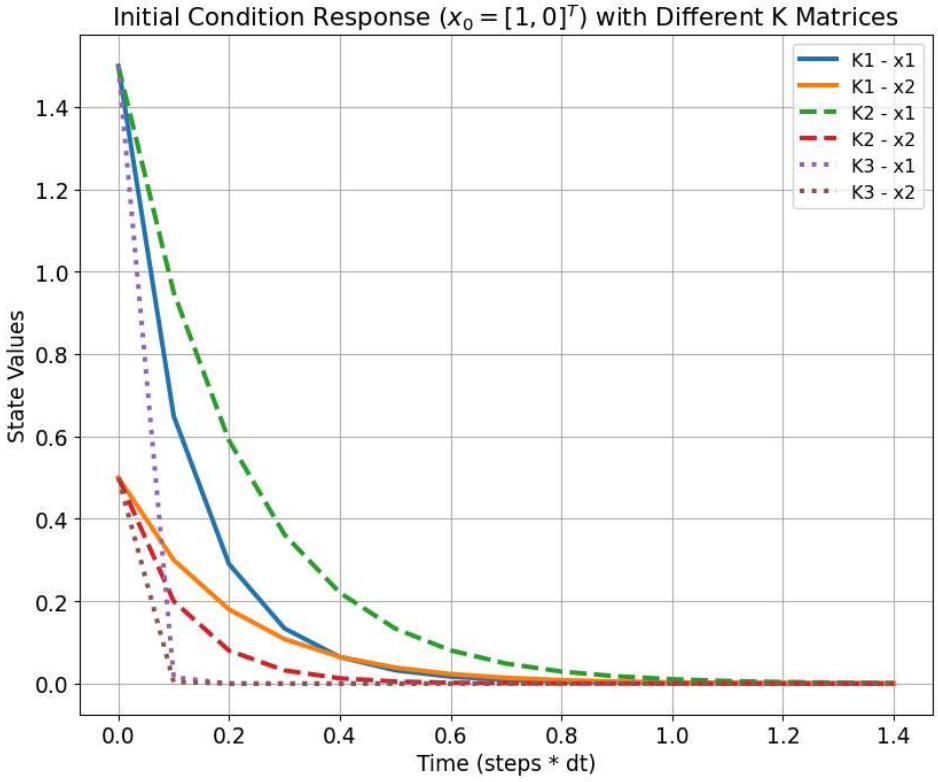
Consider the MIMO system with two states and two inputs:

$$x(kT+T) = \begin{bmatrix} 1 & 0.1 \\ 0 & 0.8 \end{bmatrix} x(kT) + \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} u(kT)$$

$$y(kT) = Cx(kT) + Du(kT)$$

```python
import numpy as np
import matplotlib.pyplot as plt
import scipy.linalg as linalg
import control as ctrl
# Create the system matrices
G = np.array([[1, 0.1], [0, 0.8]])
H = np.array([[1, 0], [0, 1]])
C = np.array([[1, 0], [0, 1]]) # Outputs are the states
D = np.array([[0, 0], [0, 0]])

# Pole placement for MIMO system
K1 = np.array([[0.6, 0], [0, 0.2]])
K2 = np.array([[0.4, 0], [0, 0.4]])

# QLQR design for MIMO system
Q = np.array([[10, 0], [0, 10]]) # State weight matrix
R = np.array([[0.1, 0], [0, 0.1]]) # Control weight matrix

# Solve the Discrete Algebraic Riccati Equation (DARE)
S = linalg.solve_discrete_are(G, H, Q, R) # S is the solution to the
# Calculate DLQR gain K
K3 = np.linalg.inv(R + H.T @ S @ H) @ (H.T @ S @ G)
print(f"K3 = [{K3[0, 0]:.3f}, {K3[0, 1]:.3f}; {K3[1, 0]:.3f}, {K3[1,
print(f"eig(G - H @ K3) = {np.linalg.eigvals(G - H @ K3)}")

```

```
K3 = [0.990, 0.099; 0.000, 0.792]
eig(G - H @ K3) = [0.00980006 0.0078745 ]
```

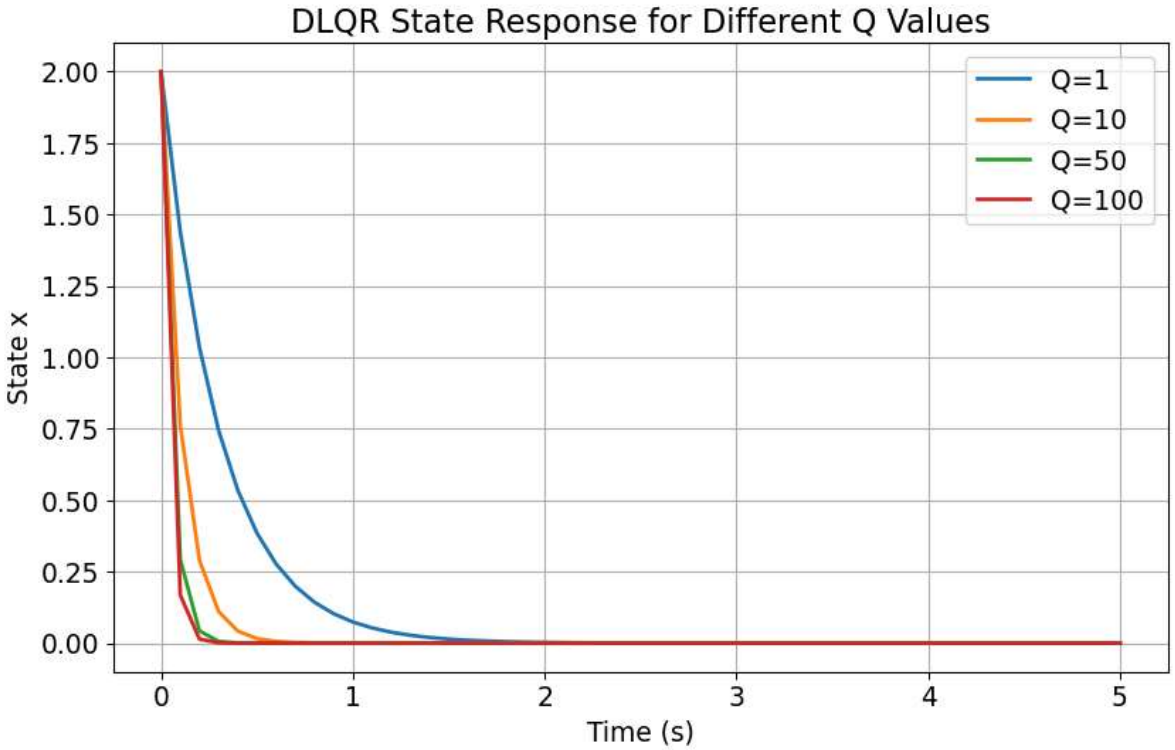


Tuning Q and R: The Trade-off

- The **relative size** (Q/R ratio) affects the **trade-off** between state regulation and control effort.
- Increasing Q relative to $R \rightarrow$ increases the **penalty on state deviation**
  - **faster state convergence**,
  - **higher control effort**.
- Increasing R relative to $Q \rightarrow$ increases the **penalty on control effort**
  - **slower state convergence**,
  - **lower control effort**.

Quick Example of DLQR Tuning

- Code



Limitations of LQR: Control Saturation

The *Linear* Quadratic Regulator may not be suitable for all systems, however.

Consider the system $\ddot{x} = x - c\dot{x} + u$, where $u_{\min} \leq u(kT) \leq u_{\max}$.

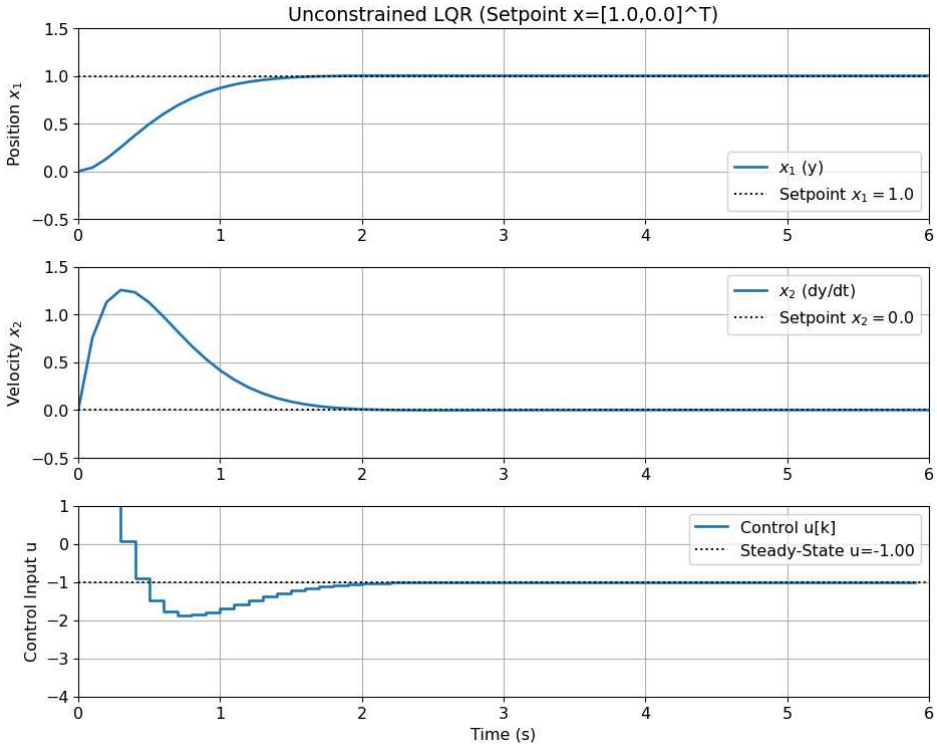
Let's compare the simulations:

- **Unconstrained LQR**: No control saturation applied.
- **Saturated LQR**: Control effort limited to $u_{\min} \leq u(kT) \leq u_{\max}$.
- **Model Predictive Control (MPC)**: Control effort limited to $u_{\min} \leq u(kT) \leq u_{\max}$, but with a prediction horizon.

Unsaturated LQR

```python
# Scenario 1: Unconstrained LQR for Nonzero Regulation (Damped 2nd Or

import numpy as np
from scipy.integrate import solve_ivp
from scipy.linalg import solve_discrete_are, expm
from scipy.signal import cont2discrete
import matplotlib.pyplot as plt
import time

# print("--- Running Scenario 1: Unconstrained LQR (Nonzero Setpoint,

# --- 1. System Definition ---
# Continuous system: dx/dt = Ac*x + Bc*u
# x = [y, y_dot]^T = [x1, x2]^T
# d(x1)/dt = x2
# d(x2)/dt = x1 - c*x2 + u
c_damping = 0.5
Ac = np.array([[0., 1.], [1., -c_damping]])
Bc = np.array([[0.], [1.]])
n_states = Ac.shape[0]
m_inputs = Bc.shape[1]

def system_ode(t, x, u, Ac_mat, Bc_mat):
    u_val = u if np.isscalar(u) else u[0]
    dxdt = Ac_mat @ x + Bc_mat @ np.array([u_val])
```

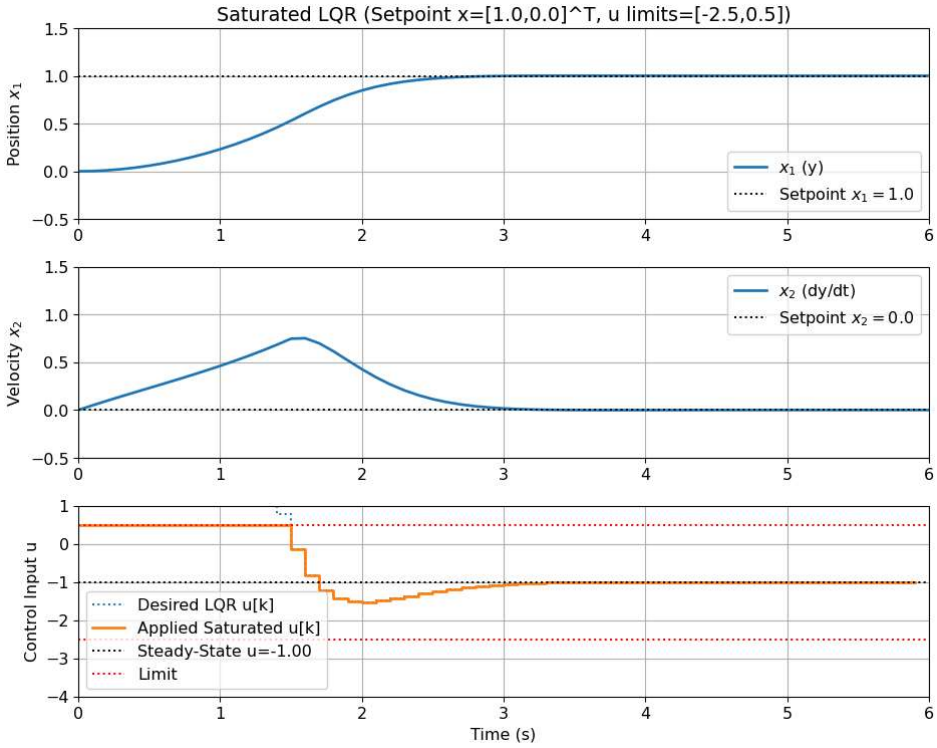
```
DLQR Gain for error K = [[8.76224795 4.36193866]]
Running simulation...
Simulation loop time: 0.02 s
```



Saturated LQR

```python
# Scenario 2: Saturated LQR for Nonzero Regulation (Damped 2nd Order

import numpy as np
from scipy.integrate import solve_ivp
from scipy.linalg import solve_discrete_are, expm
from scipy.signal import cont2discrete
import matplotlib.pyplot as plt
import time

# print("--- Running Scenario 2: Saturated LQR (Nonzero Setpoint, Dam

# --- 1. System Definition ---
c_damping = 0.5
Ac = np.array([[0., 1.], [1., -c_damping]])
Bc = np.array([[0.], [1.]])
n_states = Ac.shape[0]
m_inputs = Bc.shape[1]
def system_ode(t, x, u, Ac_mat, Bc_mat):
    u_val = u if np.isscalar(u) else u[0]
    dxdt = Ac_mat @ x + Bc_mat @ np.array([u_val])
    return dxdt.flatten()

# --- 2. Discretization ---
Ts = 0.1
Ad, Bd, _, _, _ = cont2discrete((Ac, Bc, np.eye(n_states), np.zeros((
```

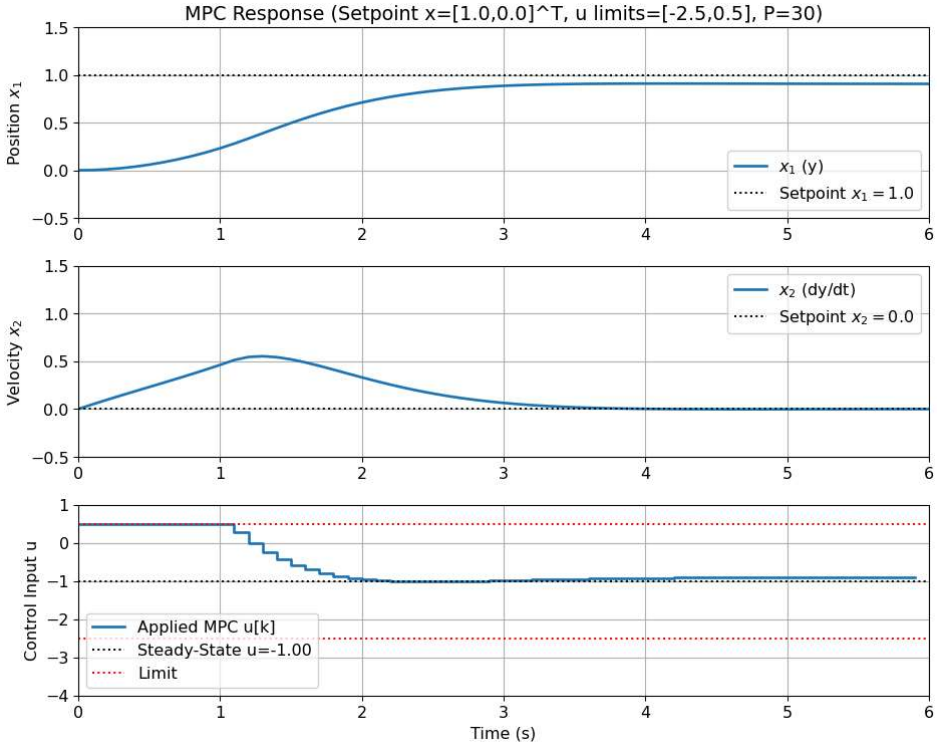
```
DLQR Gain for error K = [[8.76224795 4.36193866]]
Control Limits: [-2.5, 0.5]
Simulation loop time: 0.02 s
```



MPC

```python
# Scenario 3: MPC for Nonzero Regulation (Damped 2nd Order System)

import numpy as np
from scipy.integrate import solve_ivp
from scipy.linalg import solve_discrete_are, expm
from scipy.signal import cont2discrete
import cvxpy as cp # Requires cvxpy and a solver like OSQP
import matplotlib.pyplot as plt
import time

# print("--- Running Scenario 3: MPC (Nonzero Setpoint, Damped System

# --- 1. System Definition ---
c_damping = 0.5
Ac = np.array([[0., 1.], [1., -c_damping]])
Bc = np.array([[0.], [1.]])
n_states = Ac.shape[0]
m_inputs = Bc.shape[1]
def system_ode(t, x, u, Ac_mat, Bc_mat):
    u_val = u if np.isscalar(u) else u[0]
    dxdt = Ac_mat @ x + Bc_mat @ np.array([u_val])
    return dxdt.flatten()

# --- 2. Discretization ---
Ts = 0.1
```

```
Control Limits: [-2.5, 0.5]
MPC CVXPY Problem defined.
MPC Simulation loop time: 12.74 s
```



Model Predictive Control (MPC)

MPC is a powerful control strategy that uses an **optimization problem** to determine the control input at each time step. It is particularly useful for systems with constraints and nonlinearities.

> **Tip**
>
> While LQR solves the infinite-horizon optimal control problem, MPC solves a **finite-horizon** optimization problem at each time step (this horizon is MPC's **prediction horizon**, i.e. how many steps ahead it looks).

How MPC Works

1. Finds an optimal control sequence for a finite-horizon problem.
2. The first control input of the optimal sequence is applied to the system.
3. Repeats at the next time step, using the current state as the new initial condition.

MPC vs LQR:

- MPC handles **constraints** (saturation, **state limits**, rates of change, etc).
- MPC is more complex and computationally intensive than LQR.
- MPC's performance depends on the accuracy of the system model.

Annexes

Quadratic Forms and Positiveness

For a square matrix P and a vector x, the product $x^\intercal Px$ is a **scalar** called a **quadratic form**.

For example, $[x_1, x_2]\begin{bmatrix}1 & 2\\ 3 & 4\end{bmatrix}\begin{bmatrix}x_1\\ x_2\end{bmatrix} = 1x_1^2 + 2x_1x_2 + 3x_1x_2 + 4x_2^2.$

$P \succeq 0$ means that P is positive semi-definite, i.e. $x^T Px \geq 0$ for all $x \in \mathbb{R}^n$.

$P \succ 0$ means that P is positive definite, i.e. $x^T Px > 0$ for all $x \in \mathbb{R}^n$ and $x \neq 0$.

- A symmetric matrix P is positive definite if (and only if)
  - all its **eigenvalues** are positive.
  - all its **principal minors** are positive.

$$P = \begin{bmatrix}p_{1,1} & p_{1,2} & p_{1,3}\\ p_{2,1} & p_{2,2} & p_{2,3}\\ p_{3,1} & p_{3,2} & p_{3,3}\end{bmatrix} \succ 0 \iff p_{1,1} > 0,\ det\begin{bmatrix}p_{1,1} & p_{1,2}\\ p_{2,1} & p_{2,2}\end{bmatrix} > 0,\ det(P) > 0$$

LQR Cost Function and Tuning

Note that the cost function is a **quadratic form** in the state and control variables.

$$J = \sum_{k=0}^{\infty} (x(k)^\mathsf{T} Q x(k) + u(k)^\mathsf{T} R u(k))$$

Consider $x(k) = [x_1(k), x_2(k)]^\mathsf{T}$ and $u(k) = [u_1(k), u_2(k)]^\mathsf{T}$. The cost function can be expressed as:

$$\begin{aligned} J = \sum_{k=0}^{\infty} (&q_{1,1} x_1(k)^2 + q_{1,2} x_1(k) x_2(k) + q_{2,1} x_2(k) x_1(k) + q_{2,2} x_2(k)^2 \\ &+ r_{1,1} u_1(k)^2 + r_{1,2} u_1(k) u_2(k) + r_{2,1} u_2(k) u_1(k) + r_{2,2} u_2(k)^2) \end{aligned}$$

Notice that you have freedom to **tune** the individual weights $q_{i,j}$ and $r_{i,j}$ in the cost function.