Murray Problem 3.10: Feedforward LQR with Integral Action

Source Filename: /main.py

Rico A. R. Picone

Begin by importing the necessary libraries and modules as follows:

```
import numpy as np
import matplotlib.pyplot as plt
import control
```

Define the linear system dynamics as follows:

```
A = np.array([[0, 2], [-1, -0.1]])
B = np.array([[0], [1]])
C = np.eye(2)
D = np.zeros((2,1))
sys = control.ss(A, B, C, D)
```

Extract the number of states, inputs, and outputs as follows:

```
n = A.shape[0]  # Number of states
m = B.shape[1]  # Number of inputs
r = C.shape[0]  # Number of outputs
```

Compute the equilibrium point as follows:

```
xd = np.array([[1], [0]])  # Desired equil. state (given)
ud = np.array([[1]])  # Desired equil. input (from 0 = A*xd + B*ud)
    # This could be computed automatically, but it's a little involved
    # because in general the system is overdetermined.
if (A @ xd + B @ ud != np.zeros_like(xd)).any():  # Check validity
    raise ValueError('The desired equilibrium point is not valid.')
```

Compute an LQR controller as follows:

```
Q = np.eye(n)  # State cost matrix
R = np.eye(m)  # Input cost matrix
K, _, _ = control.lqr(A, B, Q, R)
```

We can compute the closed-loop system dynamics with a feedforward input term as follows:

```
ctrl, clsys = control.create_statefbk_iosystem(sys, K)
print(clsys)
```

```
<LinearICSystem>: sys[0]_sys[1]
Inputs (3): ['xd[0]', 'xd[1]', 'ud[0]']
Outputs (3): ['y[0]', 'y[1]', 'u[0]']
States (2): ['sys[0]_x[0]', 'sys[0]_x[1]']

A = [[ 0.          2.        ]
     [-1.41421356 -1.6330506 ]]

B = [[0.         0.         0.        ]
     [0.41421356 1.5330506  1.        ]]

C = [[ 1.          0.        ]
     [ 0.          1.        ]
     [-0.41421356 -1.5330506 ]]

D = [[0.         0.         0.        ]
     [0.         0.         0.        ]
     [0.41421356 1.5330506  1.        ]]
```

The first two inputs to the closed-loop system are the desired state, and the third input is the feedforward input. The response of the closed-loop system to an input of

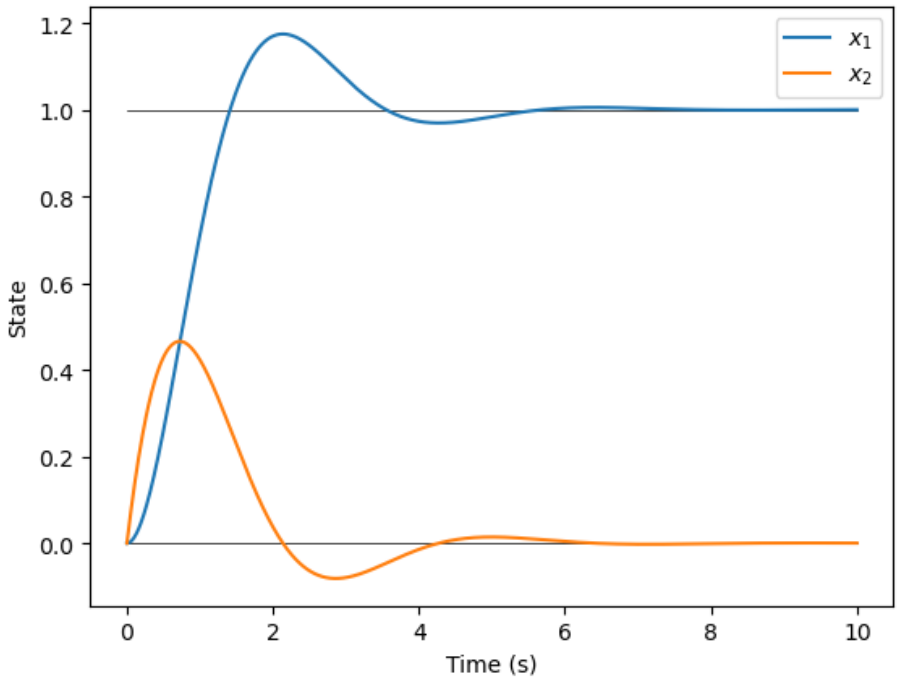
$$\begin{bmatrix} x_d \\ u_d \end{bmatrix}$$

can be simulated as follows:

```
t = np.linspace(0, 10, 1000)  # Simulation time
input_d = np.vstack([xd, ud]) @ \
    np.array([np.ones_like(t)])  # Desired state and input over time
_, output = control.forced_response(clsys, T=t, U=input_d)  # Simulate
```

Plot the response of the closed-loop system as follows:

```
fig, ax = plt.subplots()
ax.plot(t, input_d[0], 'k', linewidth=0.5)
ax.plot(t, input_d[1], 'k', linewidth=0.5)
ax.plot(t, output[0], label='$x_1$')
ax.plot(t, output[1], label='$x_2$')
ax.set_xlabel('Time (s)')
ax.set_ylabel('State')
ax.legend()
plt.draw()
```

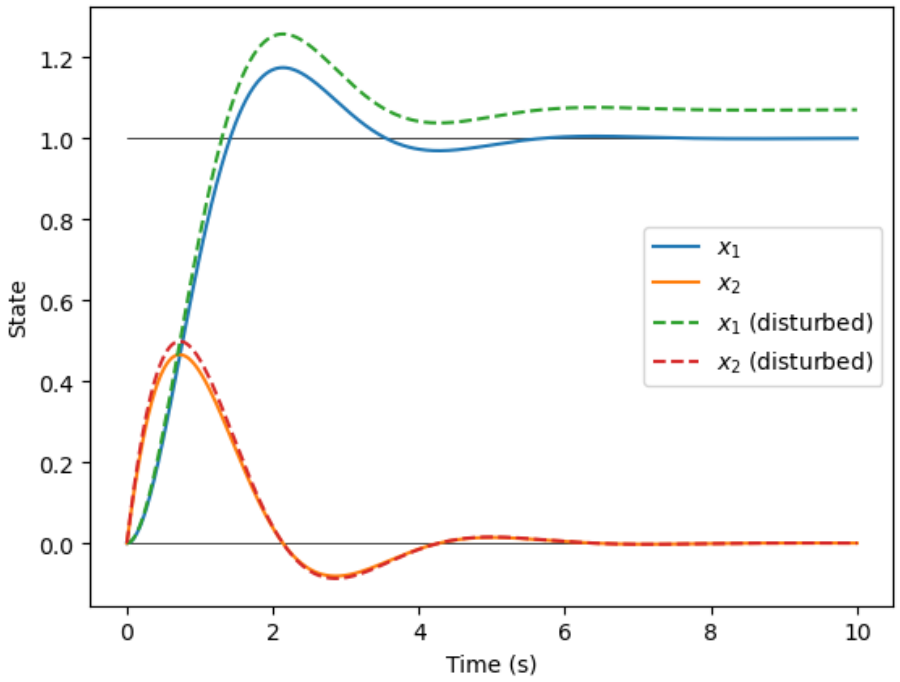


Now supposed there is an input disturbance $d = 0.1$ acting on the input to the plant. We can simulate the response of the closed-loop system to this disturbance as follows:

```
d = 0.1  # Input disturbance
input_d = np.vstack([xd, ud+d]) @ \
    np.array([np.ones_like(t)])  # Disturbed state and input over time
_, output_d = control.forced_response(clsys, T=t, U=input_d)  # Simulate
```

Plot the response of the closed-loop system to the disturbance along with the undisturbed response as follows:

```
ax.plot(t, output_d[0], '--', label='$x_1$ (disturbed)')
ax.plot(t, output_d[1], '--', label='$x_2$ (disturbed)')
ax.legend()
plt.draw()
fig
```




```
<Figure size 640x480 with 0 Axes>
```

The response of the closed-loop system to the disturbance is shown in the dashed lines. The disturbance causes the first state to deviate from the desired value. The feedback controller cannot reject this disturbance because it enters the system directly at the input. This is different than when the disturbance enters the system state or output, where the feedback controller can reject it.

One approach to rejecting the disturbance is to add integral action to the controller. This can be done by augmenting the state-space system with an integrator (see Astrom 2021, p. 8-27 and Murray 2023, pp. 3-18 through 3-19). The integral action is added to the controller as follows:

$$u = u_d - K_r(x - x_d) - K_i \int_0^t C(x - x_d)dt$$

where K_r is the state feedback gain, K_i is the integral gain, and C determines which states receive integral action (and how). To realize this, the state vector is augmented as follows:

$$x \mapsto \xi = \begin{bmatrix} x \\ z \end{bmatrix},$$

where z is the integral of the error state $x - x_d$. The dynamics of the augmented system are given by

$$A \mapsto \begin{bmatrix} A & 0 \\ C & 0 \end{bmatrix}, \quad \text{and} \quad B \mapsto \begin{bmatrix} B \\ 0 \end{bmatrix}.$$

The desired equilibrium point is augmented as

$$x_d \mapsto \begin{bmatrix} x_d \\ 0 \end{bmatrix}$$

because the integral state should be zero at the desired equilibrium point.

The `create_statefbk_iosystem` function can be used to create a state-space system with integral action by selecting a matrix C, augmenting A and B as shown above and augmenting K_r with an integral gain K_i; that is,

$$K_r \mapsto \begin{bmatrix} K_r \\ K_i \end{bmatrix}.$$

The greater the values of K_i, the faster the integral action will respond to errors. However, large values of K_i can affect system performance and stability. We will select K_i by including it in the augmented LQR design. Increasing the Q matrix values for the integral state will increase the integral gain and make the integral action faster.

The integral action can be added to the controller as follows:

```
C_int = np.array([[1, 0]])  # Select the state to get integrator
A_aug = np.block([[A, np.zeros((n, 1))], [C_int, 0]])
B_aug = np.vstack([B, np.zeros((1, m))])
Q_aug = np.eye(n+1)  # State+integral cost matrix
K_aug, _, _ = control.lqr(A_aug, B_aug, Q_aug, R)
ctrl_int, clsys_int = control.create_statefbk_iosystem(
    sys, K_aug, integral_action=C_int
)
print(clsys_int)
```

```
<LinearICSystem>: sys[0]_sys[3]
Inputs (3): ['xd[0]', 'xd[1]', 'ud[0]']
Outputs (3): ['y[0]', 'y[1]', 'u[0]']
States (3): ['sys[0]_x[0]', 'sys[0]_x[1]', 'sys[3]_x[0]']

A = [[ 0.          2.          0.        ]
     [-2.07460956 -2.30400483 -1.        ]
     [ 1.          0.          0.        ]]

B = [[ 0.          0.          0.        ]
     [ 1.07460956  2.20400483  1.        ]
     [-1.          0.          0.        ]]

C = [[ 1.          0.          0.        ]
     [ 0.          1.          0.        ]
     [-1.07460956 -2.20400483 -1.        ]]
```

```
D = [[0.         0.         0.        ]
     [0.         0.         0.        ]
     [1.07460956 2.20400483 1.        ]]
```

The response of the closed-loop system with integral action to a disturbed input of

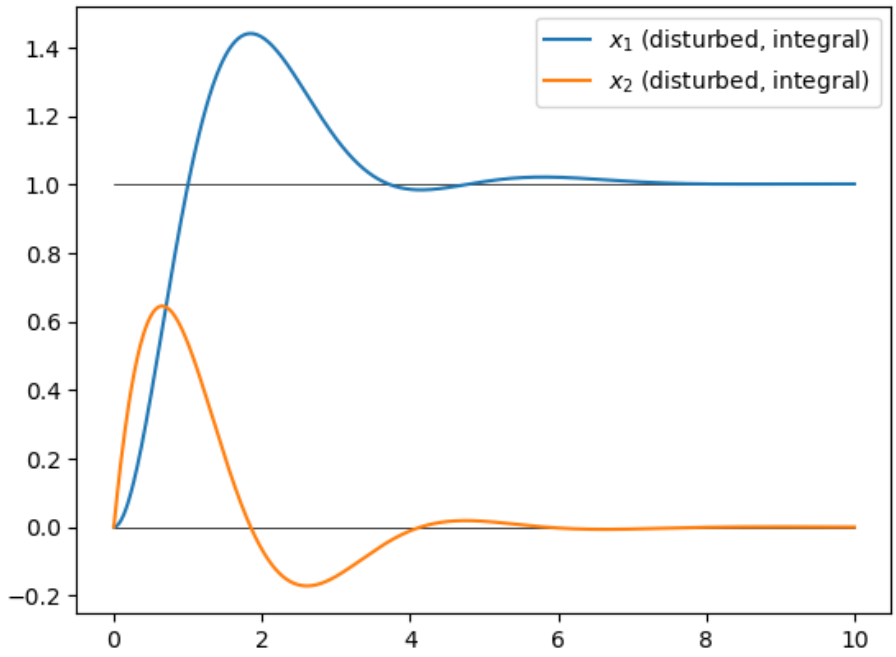
$$\begin{bmatrix} x_d \\ u_d + d \end{bmatrix}$$

can be simulated as follows:

```
_, output_int = control.forced_response(clsys_int, T=t, U=input_d)  # Simulate
```

Plot the response of the closed-loop system with integral action to the disturbance along with the undisturbed response as follows:

```
fig, ax = plt.subplots()
ax.plot(t, input_d[0], 'k', linewidth=0.5)
ax.plot(t, input_d[1], 'k', linewidth=0.5)
ax.plot(t, output_int[0], label='$x_1$ (disturbed, integral)')
ax.plot(t, output_int[1], label='$x_2$ (disturbed, integral)')
ax.legend()
plt.draw()
```



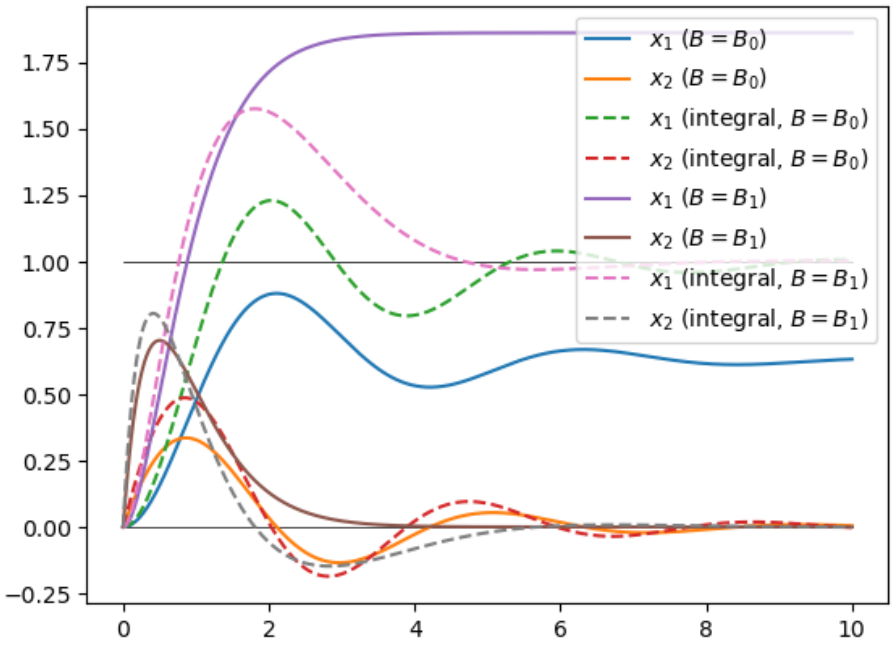
The steady-state error in the first state is eliminated by the integral action. However, note that the overshoot is increased.

Now consider two different variations of the B matrix. For each, we will try the original controller and the controller with integral action.

```
Bs = [
    np.array([[0], [0.5]]),
    np.array([[0], [2.5]])
]
outputs = []
outputs_int = []
for B in Bs:
    sys = control.ss(A, B, C, D)
    ctrl, clsys = control.create_statefbk_iosystem(sys, K)
    ctrl_int, clsys_int = control.create_statefbk_iosystem(
        sys, K_aug, integral_action=C_int
    )
    _, output = control.forced_response(clsys, T=t, U=input_d)
    _, output_int = control.forced_response(clsys_int, T=t, U=input_d)
    outputs.append(output)
    outputs_int.append(output_int)
```

Plot the responses of the closed-loop systems with and without integral action for the two different B matrices as follows:

```
fig, ax = plt.subplots()
ax.plot(t, input_d[0], 'k', linewidth=0.5)
ax.plot(t, input_d[1], 'k', linewidth=0.5)
for i in range(len(Bs)):
    ax.plot(t, outputs[i][0], label=f'$x_1$ ($B = B_{i}$)')
    ax.plot(t, outputs[i][1], label=f'$x_2$ ($B = B_{i}$)')
    ax.plot(t, outputs_int[i][0], '--', label=f'$x_1$ (integral, $B = B_{i}$)')
    ax.plot(t, outputs_int[i][1], '--', label=f'$x_2$ (integral, $B = B_{i}$)')
ax.legend()
plt.show()
```



The response of the closed-loop system with integral action is shown in the dashed lines. The integral action eliminates the steady-state error in the first state for both B matrices. The transient performance is affected, but the system remains stable. This demonstrates the effectiveness of integral action in rejecting disturbances that enter the system directly at the input or when the system model is uncertain. Note that in the latter case, without feedforward control, the steady-state error would likely have been eliminated. The reason it is not here is that we are using feedforward control based on an incorrect model. This is a common issue in practice, and integral action can help to mitigate it.

The value of the feedforward control is that it can be used to more accurately track a desired trajectory. In this case, we are providing a static desired equilibrium point, which is not a feasible trajectory, so it cannot be tracked exactly.