Linear quadratic optimal control example

Richard M. Murray, 20 Jan 2022 (updated 17 Jan 2023)

This example works through the linear quadratic finite time optimal control problem. We assume that we have a linear system of the form

$$\dot{x} = Ax + Bu$$

and that we want to minimize a cost function of the form

$$\int_0^T (x^T Q_x x + u^T Q_u u) dt + x^T P_1 x.$$

We show how to compute the solution the the Riccati ODE and use this to obtain an optimal (time-varying) linear controller.

In [1]:
```
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt
import control as ct
import control.optimal as opt
import time
```

System dynamics

We use the linearized dynamics of the vehicle steering problem as our linear system. This is mainly for convenient (since we have some intuition about it).

In [2]:
```
# Use the linearized dynamics of the vehicle control problem
# (you can find kincar.py on the course website)
from kincar import kincar, plot_lanechange

# Initial conditions
x0 = np.array([-40, -2., 0.])
u0 = np.array([10, 0])                # only used for linearization
Tf = 4

# Linearized dynamics
sys = kincar.linearize(x0, u0)
print(sys)
```

```
<LinearIOSystem>: sys[2]
Inputs (2): ['u[0]', 'u[1]']
Outputs (3): ['y[0]', 'y[1]', 'y[2]']
States (3): ['x[0]', 'x[1]', 'x[2]']

A = [[ 0.0000000e+00  0.0000000e+00 -5.0004445e-06]
     [ 0.0000000e+00  0.0000000e+00  1.0000000e+01]
     [ 0.0000000e+00  0.0000000e+00  0.0000000e+00]]

B = [[1.         0.        ]
     [0.         0.        ]
     [0.         3.33333333]]

C = [[1. 0. 0.]
     [0. 1. 0.]
     [0. 0. 1.]]

D = [[0. 0.]
     [0. 0.]
     [0. 0.]]
```

Optimal trajectory generation

We generate an trajectory for the system that minimizes the cost function above. Namely, starting from some initial function $x(0) = x_0$, we wish to bring the system toward the origin without using too much control effort.

In [3]:
```
# Define the cost function and the terminal cost
# (try changing these later to see what happens)
Qx = np.diag([1, 1, 1])       # state costs
Qu = np.diag([1, 1])          # input costs
Pf = np.diag([1, 1, 1])       # terminal costs
```

Finite time, linear quadratic optimization

The optimal solution satisfies the following equations, which follow from the maximum principle:

$$
\begin{aligned}
\dot{x} &= \left(\frac{\partial H}{\partial \lambda}\right)^T = Ax + Bu, &\qquad x(0) &= x_0, \\
-\dot{\lambda} &= \left(\frac{\partial H}{\partial x}\right)^T = Q_x x + A^T \lambda, &\qquad \lambda(T) &= P_1 x(T), \\
0 &= \left(\frac{\partial H}{\partial u}\right)^T = Q_u u + B^T \lambda.
\end{aligned}
$$

The last condition can be solved to obtain the optimal controller

$$u = -Q_u^{-1} B^T \lambda,$$

which can be substituted into the equations for the optimal solution.

Given the linear nature of the dynamics, we attempt to find a solution by setting $\lambda(t) = P(t)x(t)$ where $P(t) \in \mathbb{R}^{n\times n}$. Substituting this into the necessary condition, we obtain

$$\begin{aligned}\dot\lambda = \dot P x + P \dot x &= \dot P x + P(Ax - BQ_u^{-1}B^T P)x, \\ \implies \qquad -\dot P x - PAx + PBQ_u^{-1}BPx &= Q_x x + A^T P x.\end{aligned}$$

This equation is satisfied if we can find $P(t)$ such that

$$-\dot P = PA + A^T P - PBQ_u^{-1}B^T P + Q_x, \qquad P(T) = P_1.$$

To solve a final value problem with $P(T) = P_1$, we set the "initial" condition to P_1 and then invert time, so that we solve

$$\frac{dP}{d(-t)} = -\frac{dP}{dt} = -F(P), \qquad P(0) = P_1$$

Solving this equation from time $t = 0$ to time $t = T$ will give us an solution that goes from $P(T)$ to $P(0)$.

In [4]:
```
# Set up the Riccatti ODE
def Pdot_reverse(t, x):
    # Get the P matrix from the state by resizing
    P = np.reshape(x, (sys.nstates, sys.nstates))

    # Compute the right hand side of Riccati ODE
    Prhs = P @ sys.A + sys.A.T @ P + Qx - \
        P @ sys.B @ np.linalg.inv(Qu) @ sys.B.T @ P

    # Return P as a vector, *backwards* in time (no minus sign)
    return Prhs.reshape((-1))

# Solve the Riccati ODE (converting from matrix to vector and back)
P0 = np.reshape(Pf, (-1))
Psol = sp.integrate.solve_ivp(Pdot_reverse, (0, Tf), P0)
Pfwd = np.reshape(Psol.y, (sys.nstates, sys.nstates, -1))

# Reorder the solution in time
Prev = Pfwd[:, :, ::-1]
trev = Tf - Psol.t[::-1]

print("Trange = ", trev[0], "to", trev[-1])
print("P[Tf] =", Prev[:,:,-1])
print("P[0] =", Prev[:,:,0])

# Internal comparison: show that initial value is close to algebraic solutic
_, P_lqr, _ = ct.lqr(sys.A, sys.B, Qx, Qu)
print("P_lqr =", P_lqr)
```

```
Trange =  0.0 to 4.0
P[Tf] = [[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
P[0] = [[ 1.00000000e+00  3.86208813e-07 -1.15917383e-07]
 [ 3.86208813e-07  2.64685426e-01  3.00060130e-01]
 [-1.15917383e-07  3.00060130e-01  7.93554820e-01]]
P_lqr = [[ 1.00000000e+00  3.86261437e-07 -1.15878431e-07]
 [ 3.86261437e-07  2.64575131e-01  3.00000000e-01]
 [-1.15878431e-07  3.00000000e-01  7.93725393e-01]]
```

For solving the x dynamics, we need a function to evaluate $P(t)$ at an arbitrary time (used by the integrator). We can do this with the SciPy `interp1d` function:

In [5]:
```
# Define an interpolation function for P
P = sp.interpolate.interp1d(trev, Prev)

print("P(0) =", P(0))
print("P(3.5) =", P(3.5))
print("P(4) =", P(4))
```

```
P(0) = [[ 1.00000000e+00  3.86208813e-07 -1.15917383e-07]
 [ 3.86208813e-07  2.64685426e-01  3.00060130e-01]
 [-1.15917383e-07  3.00060130e-01  7.93554820e-01]]
P(3.5) = [[ 1.00000000e+00  3.85128042e-07 -1.19928513e-07]
 [ 3.85128042e-07  2.73611300e-01  3.19711737e-01]
 [-1.19928513e-07  3.19711737e-01  8.38939788e-01]]
P(4) = [[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

We now solve the $\dot{x}$ equations *forward* in time, using $P(t)$:

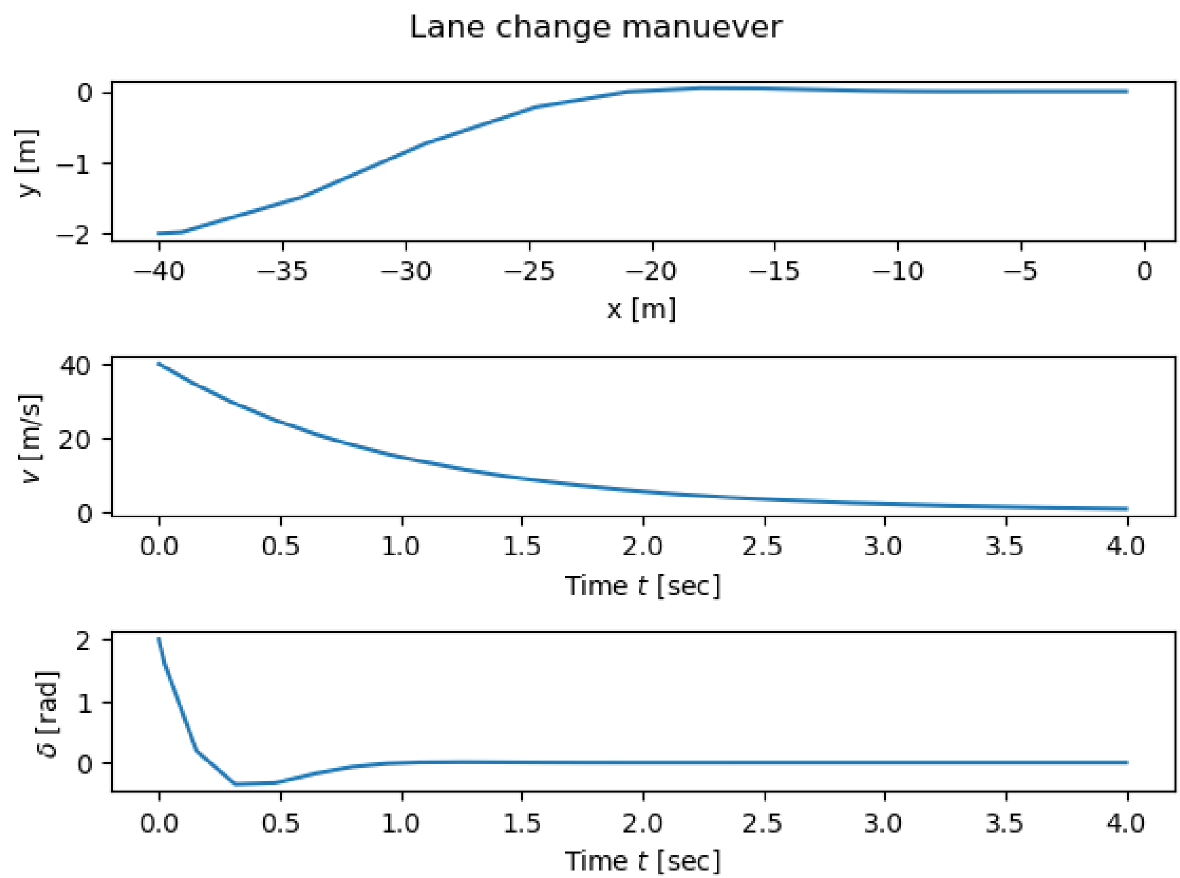
In [6]:
```
# Now solve the state forward in time
def xdot_forward(t, x):
    u = -np.linalg.inv(Qu) @ sys.B.T @ P(t) @ x
    return sys.A @ x + sys.B @ u

# Now simulate from a shifted initial condition
xsol = sp.integrate.solve_ivp(xdot_forward, (0, Tf), x0)
tvec = xsol.t
x = xsol.y
print("x[0] =", x[:, 0])
print("x[Tf] =", x[:, -1])
```

```
x[0] = [-40.  -2.   0.]
x[Tf] = [-7.32629521e-01 -2.56435711e-07 -3.40703665e-07]
```

In [7]:
```
# Finally compute the "desired" state and input values
xd = x
ud = np.zeros((sys.ninputs, tvec.size))
for i, t in enumerate(tvec):
  ud[:, i] = -np.linalg.inv(Qu) @ sys.B.T @ P(t) @ x[:, i]

plot_lanechange(tvec, xd, ud)
```



Note here that we are stabilizing the system to the origin (compared to some of other examples where we change langes and so the final y position is $y_{\mathrm{f}} = 2$.

In [ ]: