cruise

May 26, 2019

1 Feedback Examples: Cruise Control

Richard M. Murray, 26 May 2019

The cruise control system of a car is a common feedback system encountered in everyday life. The system attempts to maintain a constant velocity in the presence of disturbances primarily caused by changes in the slope of a road. The controller compensates for these unknowns by measuring the speed of the car and adjusting the throttle appropriately.

This notebook explores the dynamics and control of the cruise control system, following the material presenting in Feedback Systems by Astrom and Murray. A ~~full~~ nonlinear model of the vehicle dynamics is used, with both PI and state space control laws. Different methods of constructing control systems are shown, all using the InputOutputSystem class (and subclasses).

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        from math import pi
        import control as ct
```

1.1 Process Model

1.1.1 Vehicle Dynamics

To develop a mathematical model we start with a force balance for the car body. Let v be the speed of the car, m the total mass (including passengers), F the force generated by the contact of the wheels with the road, and F_d the disturbance force due to gravity, friction, and aerodynamic drag.

```
In [2]: def vehicle_update(t, x, u, params={}):
            """Vehicle dynamics for cruise control system.

            Parameters
            ----------
            x : array
                 System state: car velocity in m/s
            u : array
                 System input: [throttle, gear, road_slope], where throttle is
                 a float between 0 and 1, gear is an integer between 1 and 5,
                 and road_slope is in rad.
```

Confusion about 1/2

\(\rho\)? (1/2)\rho

Jdr.

```
    # 1/\rho Cd A |v| v, where  is the density of air, Cd is the
    # shape-dependent aerodynamic drag coefficient, and A is the frontal area
    # of the car.

    Fa = 1/2 * rho * Cd * A * abs(v) * v

    # Final acceleration on the car
    Fd = Fg + Fr + Fa
    dv = (F - Fd) / m

    return dv
```

1.1.2 Engine model

The force F is generated by the engine, whose torque is proportional to the rate of fuel injection, which is itself proportional to a control signal 0 <= u <= 1 that controls the throttle position. The torque also depends on engine speed omega.

```
In [3]: def motor_torque(omega, params={}):
            # Set up the system parameters
            Tm = params.get('Tm', 190.)             # engine torque constant
            omega_m = params.get('omega_m', 420.)   # peak engine angular speed
            beta = params.get('beta', 0.4)          # peak engine rolloff

            return np.clip(Tm * (1 - beta * (omega/omega_m - 1)**2), 0, None)
```

Torque curves for a typical car engine. The graph on the left shows the torque generated by the engine as a function of the angular velocity of the engine, while the curve on the right shows torque as a function of car speed for different gears.

```
In [4]: plt.figure()
        plt.suptitle('Torque curves for typical car engine')

        # Figure 4.2a - single torque curve as function of omega
        omega_range = np.linspace(0, 700, 701)
        plt.subplot(2, 2, 1)
        plt.plot(omega_range, [motor_torque(w) for w in omega_range])
        plt.xlabel('Angular velocity $\omega$ [rad/s]')
        plt.ylabel('Torque $T$ [Nm]')
        plt.grid(True, linestyle='dotted')

        # Figure 4.2b - torque curves in different gears, as function of velocity
        plt.subplot(2, 2, 2)
        v_range = np.linspace(0, 70, 71)
        alpha = [40, 25, 16, 12, 10]
        for gear in range(5):
            omega_range = alpha[gear] * v_range
            plt.plot(v_range, [motor_torque(w) for w in omega_range],
                     color='blue', linestyle='solid')
```

```
u_min = 0; u_max = 2 if antiwindup else 1; u_ind = sys.find_output('u')

# Make sure the upper and lower bounds on v are OK
while max(y[v_ind]) > v_max: v_max += 1
while min(y[v_ind]) < v_min: v_min -= 1

# Create arrays for return values
subplot_axes = subplots.copy()

# Velocity profile
if subplot_axes[0] is None:
    subplot_axes[0] = plt.subplot(2, 1, 1)
else:
    plt.sca(subplots[0])
plt.plot(t, y[v_ind], linetype)
plt.plot(t, vref*np.ones(t.shape), 'k-')
plt.plot([t_hill, t_hill], [v_min, v_max], 'k--')
plt.axis([0, t[-1], v_min, v_max])
plt.xlabel('Time $t$ [s]')
plt.ylabel('Velocity $v$ [m/s]')

# Commanded input profile
if subplot_axes[1] is None:
    subplot_axes[1] = plt.subplot(2, 1, 2)
else:
    plt.sca(subplots[1])
plt.plot(t, y[u_ind], 'r--' if antiwindup else linetype)
plt.plot([t_hill, t_hill], [u_min, u_max], 'k--')
plt.axis([0, t[-1], u_min, u_max])
plt.xlabel('Time $t$ [s]')
plt.ylabel('Throttle $u$')

# Applied input profile
if antiwindup:
    # TODO: plot the actual signal from the process?
    plt.plot(t, np.clip(y[u_ind], 0, 1), linetype)
    plt.legend(['Commanded', 'Applied'], frameon=False)

return subplot_axes
```

1.2 State space controller

Förtydliga SS & PI

Construct a state space controller with integral action, linearized around an equilibrium point. The controller is constructed around the equilibrium point (x_d, u_d) and includes both feedforward and feedback compensation.

- Controller inputs - (x, y, r): system states, system output, reference
- Controller state - z: integrated error $(y - r)$

```
# To find the equilibrium point, we run an initial simulation (crude hack)
# TODO: replace with find_eqpt() calculation when updated to allow selections
t, y, x = ct.input_output_response(
    cruise_sf, T, [vref, gear, theta0], [20, 0], return_x=True,
    params={'K':0.5, 'ki':0.1, 'xd':vref[0], 'yd':vref[0]})
X0 = x[:, -1]
Yf = y[:, -1]

# Yf represents the steady state with PI control => we can use it to
# identify the steady state velocity and required throttle setting.
xd = Yf[1]
ud = Yf[0]
yd = Yf[1]

# Compute the linearized system at the eq pt
cruise_linearized = ct.linearize(vehicle, xd, [ud, gear[0], 0])
```

```
In [7]: # Construct the gain matrices for the system
        A, B, C = cruise_linearized.A, cruise_linearized.B[0, 0], cruise_linearized.C
        K = 0.5
        kf = -1 / (C * np.linalg.inv(A - B * K) * B)

        # Response of the system with no integral feedback term
        plt.figure()
        plt.suptitle('Cruise control with proportional and PI control')
        theta_hill = [
            0 if t <= 5 else
            4./180. * pi * (t-5) if t <= 6 else
            4./180. * pi for t in T]
        t, y = ct.input_output_response(
            cruise_sf, T, [vref, gear, theta_hill], [X0[0], 0],
            params={'K':K, 'kf':kf, 'ki':0.0, 'kf':kf, 'xd':xd, 'ud':ud, 'yd':yd})
        subplots = cruise_plot(cruise_sf, t, y, t_hill=5, linetype='b--')

        # Response of the system with state feedback + integral action
        t, y = ct.input_output_response(
            cruise_sf, T, [vref, gear, theta_hill], [X0[0], 0],
            params={'K':K, 'kf':kf, 'ki':0.1, 'kf':kf, 'xd':xd, 'ud':ud, 'yd':yd})
        cruise_plot(cruise_sf, t, y, t_hill=5, linetype='b-', subplots=subplots)

        # Add a legend
        import matplotlib.lines as mlines
        p_line = mlines.Line2D([], [], color='blue', linestyle='--', label='Proportional')
        pi_line = mlines.Line2D([], [], color='blue', linestyle='-', label='PI control')
        plt.legend(handles=[p_line, pi_line], frameon=False, loc='lower right');
```

Selecting pi gain

State feedback

(Compare with Section 1.3)

Example should use SS + integral a

Mention ≡ PI later

1.3.1 Robustness to change in mass

```
In [10]: # Define the time and input vectors
         T = np.linspace(0, 25, 101)
         vref = 20 * np.ones(T.shape)
         gear = 4 * np.ones(T.shape)
         theta0 = np.zeros(T.shape)

         # Now simulate the effect of a hill at t = 5 seconds
         plt.figure()
         plt.suptitle('Response to change in road slope')
         theta_hill = np.array([
             0 if t <= 5 else
             4./180. * pi * (t-5) if t <= 6 else
             4./180. * pi for t in T])

         subplots = [None, None]
         linecolor = ['red', 'blue', 'green']
         handles = []
         for i, m in zip([0, 1, 2], [1200, 1600, 2000]):
             # Compute the equilibrium state for the system
             # TODO: convert to call to find_eqpts (need to fix gear, theta0)
             t, y, x = ct.input_output_response(
                 cruise_tf, T*2, [vref, gear, theta0], [vref[-1], 0],
                 return_x=True, params={'m':m})
             X0 = x[:, -1]                      # Initial state (for disturbance response)

             t, y = ct.input_output_response(
                 cruise_tf, T, [vref, gear, theta_hill], X0, params={'m':m})

             subplots = cruise_plot(cruise_tf, t, y, t_hill=5, subplots=subplots,
                                    linetype=linecolor[i][0] + '-')
             handles.append(mlines.Line2D([], [], color=linecolor[i], linestyle='-',
                                          label="m = %d" % m))

         # Add labels to the plots
         plt.sca(subplots[0])
         plt.ylabel('Speed [m/s]')
         plt.legend(handles=handles, frameon=False, loc='lower right');

         plt.sca(subplots[1])
         plt.ylabel('Throttle')
         plt.xlabel('Time [s]');
```

```
def pi_output(t, x, u, params={}):
    # Get the controller parameters that we need
    kp = params.get('kp', 0.5)
    ki = params.get('ki', 0.1)

    # Assign variables for inputs and states (for readability)
    v = u[0]                    # current velocity
    vref = u[1]                 # reference velocity
    z = x[0]                    # integrated error

    # PI controller
    return kp * (vref - v) + ki * z

control_pi = ct.NonlinearIOSystem(
    pi_update, pi_output, name='control',
    inputs = ['v', 'vref'], outputs = ['u'], states = ['z'],
    params = {'kp':0.5, 'ki':0.1})

# Create the closed loop system
cruise_pi = ct.InterconnectedSystem(
    (vehicle, control_pi), name='cruise',
    connections=(
        ('vehicle.u', 'control.u'),
        ('control.v', 'vehicle.v')),
    inplist=('control.vref', 'vehicle.gear', 'vehicle.theta'),
    outlist=('control.u', 'vehicle.v'), outputs=['u', 'v'])
```

1.4.1 Response to a small hill

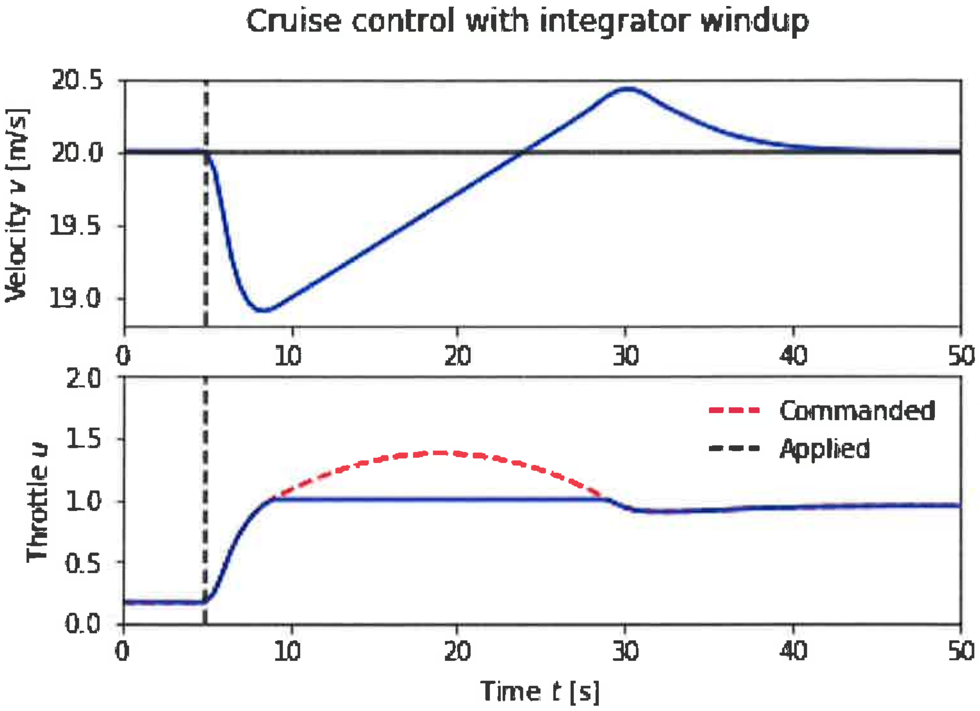
Figure 4.3b shows the response of the closed loop system. The figure shows that even if the hill is so steep that the throttle changes from 0.17 to almost full throttle, the largest speed error is less than 1 m/s, and the desired velocity is recovered after 20 s.

```
In [12]: # Define the time and input vectors
         T = np.linspace(0, 30, 101)
         vref = 20 * np.ones(T.shape)
         gear = 4 * np.ones(T.shape)
         theta0 = np.zeros(T.shape)

         # Compute the equilibrium throttle setting for the desired speed
         # TODO: convert to call to find_eqpts (need to fix gear, theta0)
         t, y, x = ct.input_output_response(
             cruise_pi, T, [vref, gear, theta0], [20, 0], return_x=True)
         X0 = x[:, -1]
         Yf = y[:, -1]

         # Now simulate the effect of a hill at t = 5 seconds
         plt.figure()
```

```
    cruise_pi, T, [vref, gear, theta_hill], X0,
    params={'kaw':0})
cruise_plot(cruise_pi, t, y, antiwindup=True);
```



Very nice pictures!

1.4.3 PI controller with anti-windup compensation

Anti-windup can be applied to the system to improve the response. Because of the feedback from the actuator model, the output of the integrator is quickly reset to a value such that the controller output is at the saturation limit.

```
In [14]: plt.figure()
         plt.suptitle('Cruise control with integrator anti-windup protection')
         t, y = ct.input_output_response(
             cruise_pi, T, [vref, gear, theta_hill], X0,
             params={'kaw':2.})
         cruise_plot(cruise_pi, t, y, antiwindup=True);
```