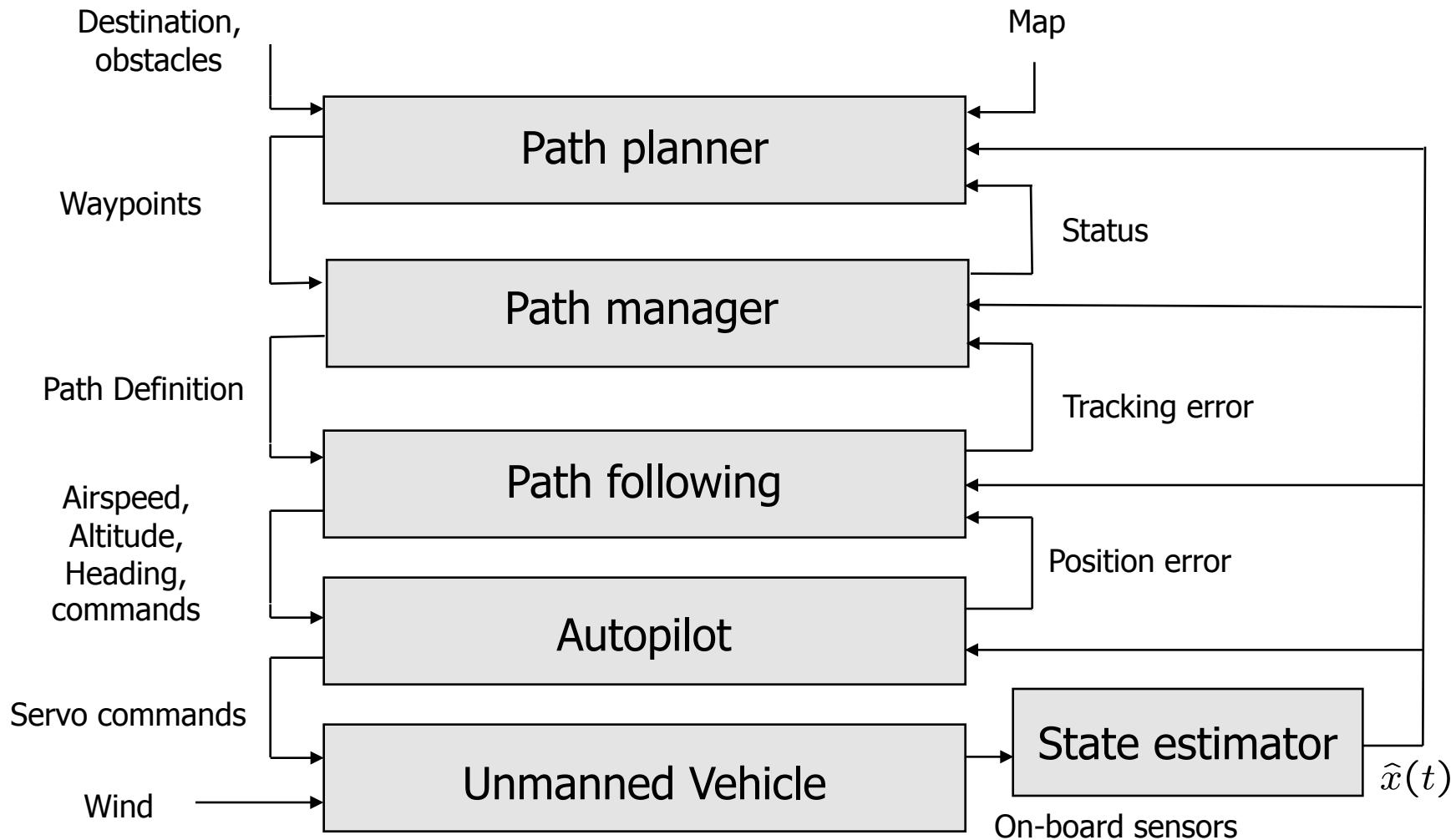




Chapter 6

Successive Loop Closure

Architecture

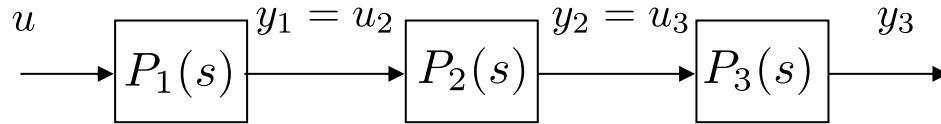


Outline

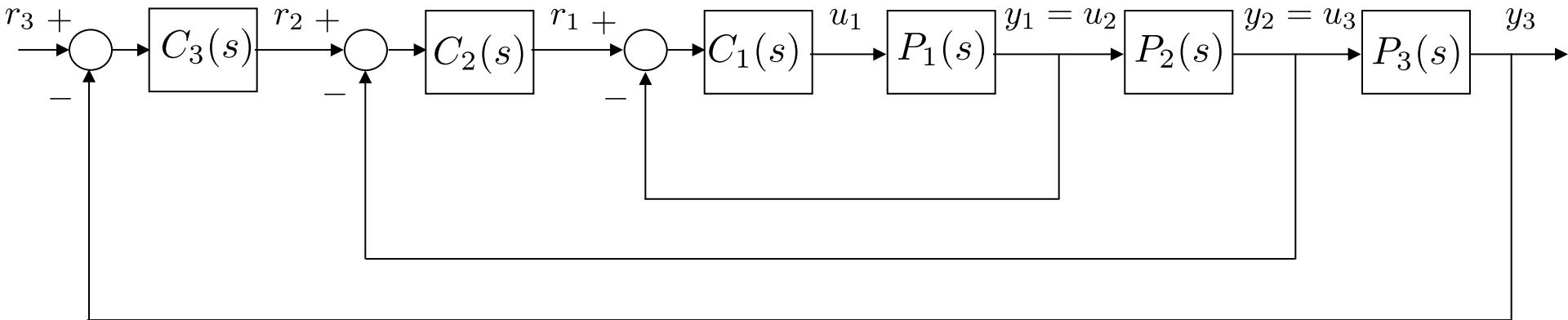
- Different Options for Autopilot Design
 - Successive Loop Closure
 - Total Energy Control
 - LQR Control

Successive Loop Closure

Open-loop system

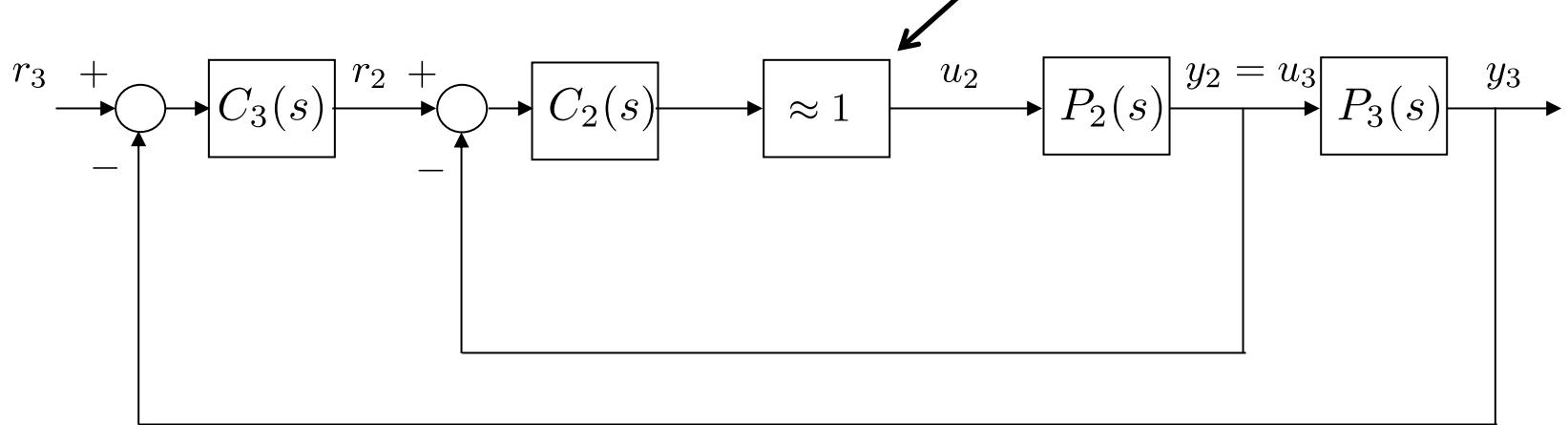


Closed-loop system

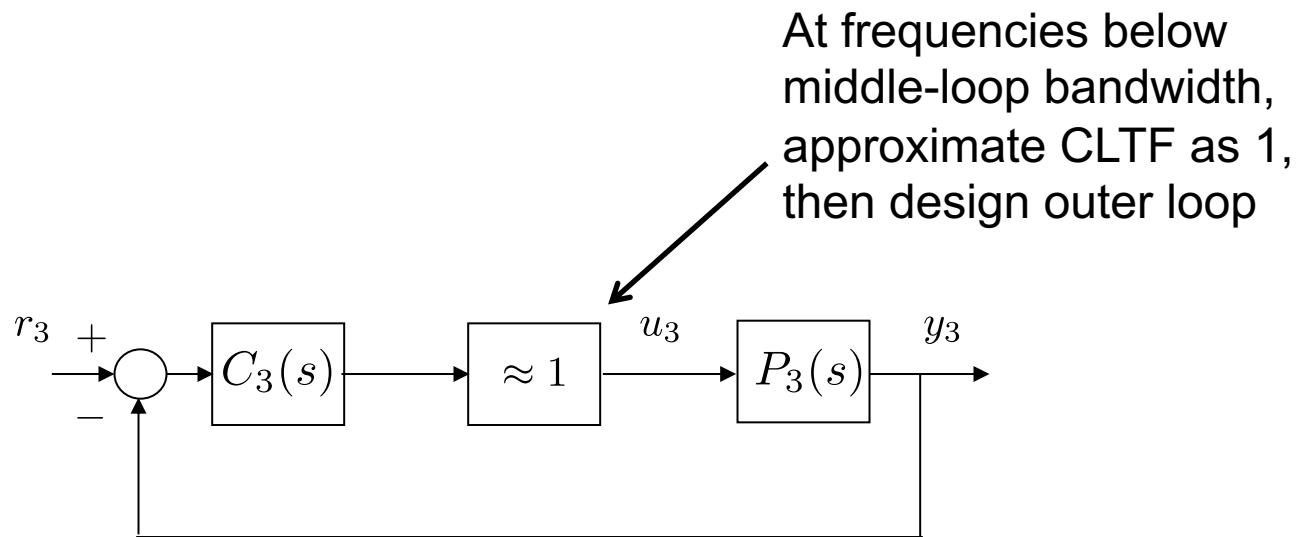


SLC: Inner Loop Closed

At frequencies below inner-loop bandwidth, approximate CLTF as 1, then design middle loop



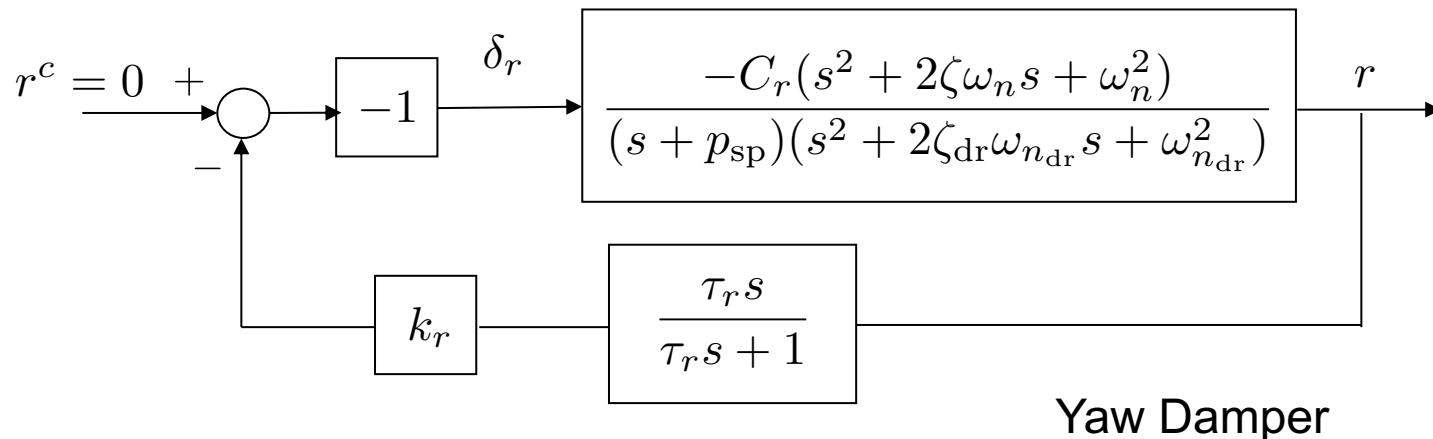
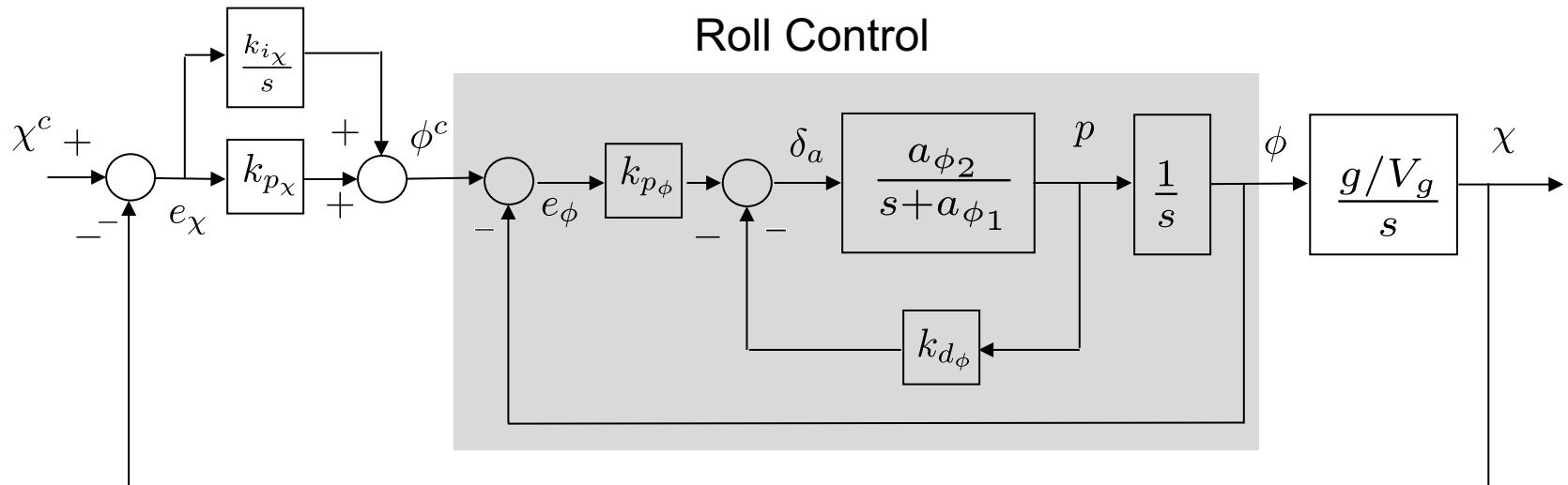
SLC: Two Loops Closed



Key idea: Each successive loop must be lower in bandwidth
--- typically by a factor of 10 or more

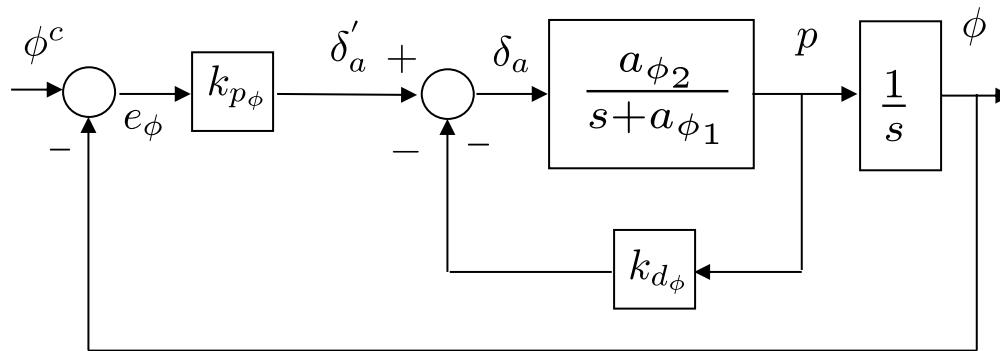
Lateral-directional Autopilot

Course Control



Yaw Damper

Roll Autopilot



$$H_{\phi/\phi^c}(s) = \underbrace{\frac{k_{p_\phi} a_{\phi_2}}{s^2 + (a_{\phi_1} + a_{\phi_2} k_{d_\phi})s + k_{p_\phi} a_{\phi_2}}}_{\text{Closed Loop TF}} = \underbrace{\frac{\omega_{n_\phi}^2}{s^2 + 2\zeta_\phi \omega_{n_\phi} s + \omega_{n_\phi}^2}}_{\text{Canonical } 2^{nd}\text{-order TF}}$$

Design parameters are ω_{n_ϕ} and ζ_ϕ

Gains are given by

$$k_{p_\phi} = \frac{\omega_{n_\phi}^2}{a_{\phi_2}}$$

$$k_{d_\phi} = \frac{2\zeta_\phi \omega_{n_\phi} - a_{\phi_1}}{a_{\phi_2}}$$

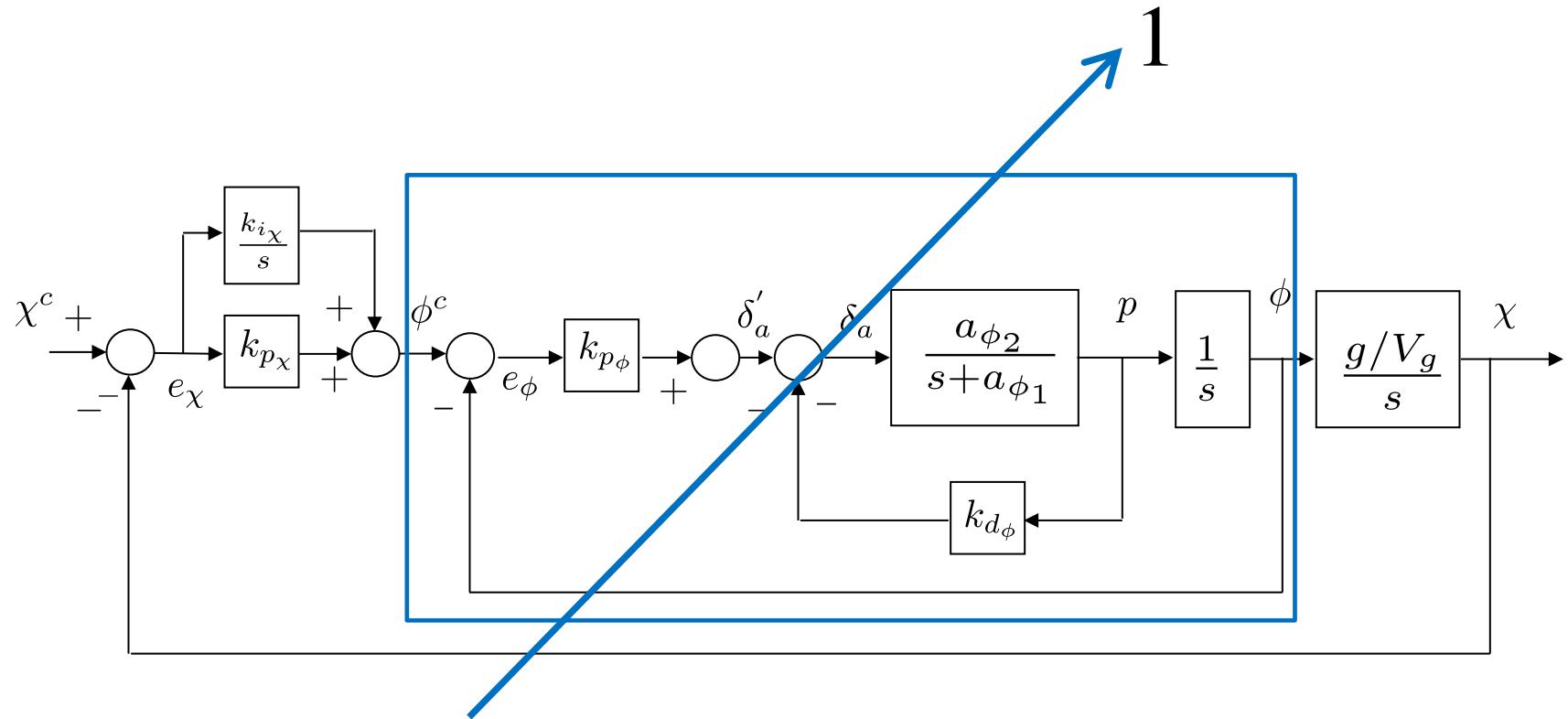
Implementation:

$$\delta_a(t) = k_{p_\phi}(\phi^c(t) - \phi(t)) - k_{d_\phi}p(t).$$

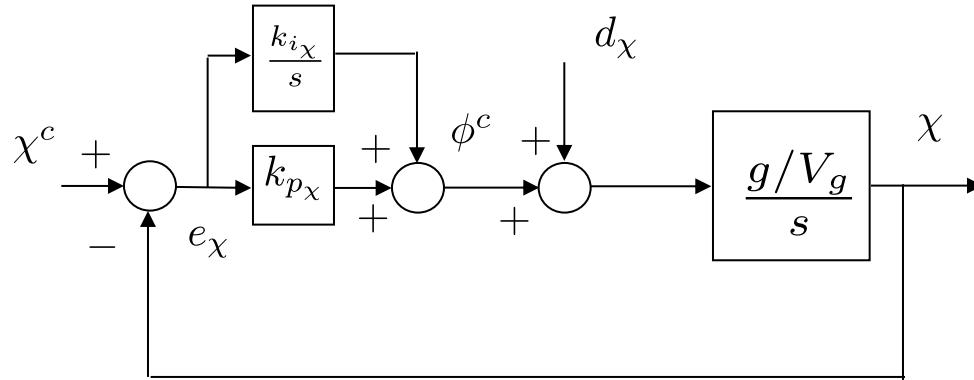
Roll Autopilot

- The book suggests using an integrator on roll in the roll loop to correct for any steady-state error due to disturbances
- Our current suggestion is to not have an integrator on inner loops, including the roll loop
 - Integrators add delay and instability -> not a good idea for inner-most loops
 - An integrator will be used on the course loop to correct for steady-state errors

Lateral-directional Autopilot



Course Hold Loop



For the course loop, note the presence of the input disturbance

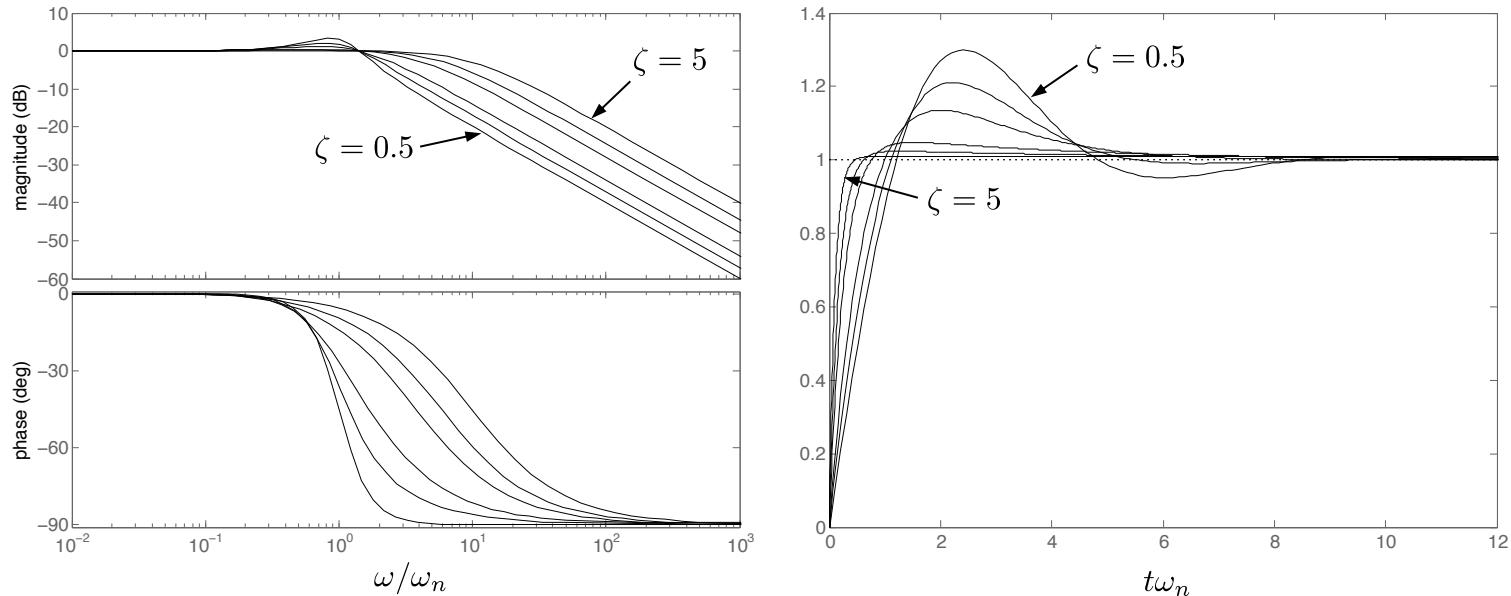
Using a PI controller for course, the response to the course command and disturbance is given by

$$\chi = \frac{k_{p_\chi}g/V_g s + k_{i_\chi}g/V_g}{s^2 + k_{p_\chi}g/V_g s + k_{i_\chi}g/V_g} \chi^c + \frac{g/V_g s}{s^2 + k_{p_\chi}g/V_g s + k_{i_\chi}g/V_g} d_\chi$$

Note:

- There is a zero in the response to the course command χ^c
- The presence of the zero at the origin ensures rejection of low-frequency disturbances

TF Zero Affects Response

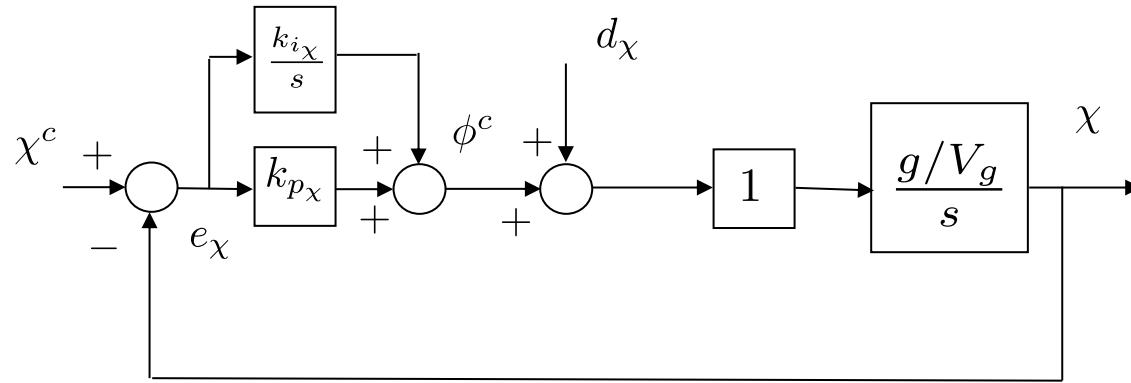


With a zero, the canonical second-order TF is given by

$$H(s) = \frac{2\zeta\omega_n s + \omega_n^2}{s^2 + 2\zeta\omega_n s + \omega_n^2} = \left(\frac{2\zeta}{\omega_n}\right) \underbrace{\frac{\omega_n^2 s}{s^2 + 2\zeta\omega_n s + \omega_n^2}}_{d/dt \text{ of } 2^{\text{nd}}\text{-order TF}} + \underbrace{\frac{\omega_n^2}{s^2 + 2\zeta\omega_n s + \omega_n^2}}_{2^{\text{nd}}\text{-order TF}}$$

Note that ζ has a different effect when the zero is present

Course Hold Loop



$$\chi = \underbrace{\frac{(k_{p_\chi}g/V_g)s + (k_{i_\chi}g/V_g)}{s^2 + (k_{p_\chi}g/V_g)s + (k_{i_\chi}g/V_g)}\chi^c}_{\text{Response to course command}} + \underbrace{\frac{(g/V_g)s}{s^2 + (k_{p_\chi}g/V_g)s + (k_{i_\chi}g/V_g)}d_\chi}_{\text{Response to disturbance}}$$

Equating coefficients to canonical TF gives:

$$\omega_{n_\chi}^2 = k_{i_\chi}g/V_g \quad \text{and} \quad 2\zeta_\chi\omega_{n_\chi} = k_{p_\chi}g/V_g$$

or

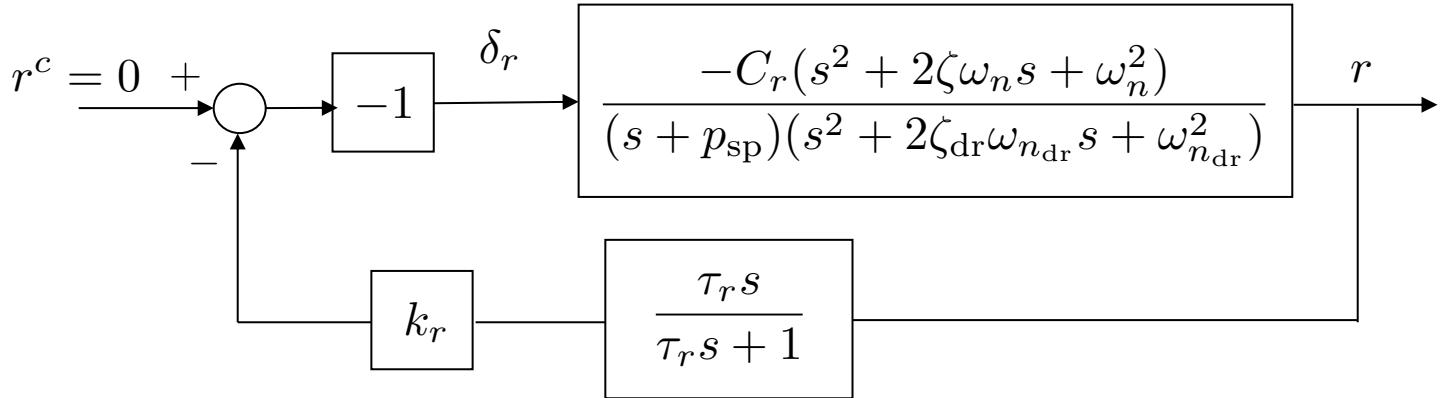
$$\omega_{n_\chi} = \frac{1}{W_\chi}\omega_{n_\phi}$$

$$k_{p_\chi} = 2\zeta_\chi \omega_{n_\chi} V_g/g$$

$$k_{i_\chi} = \omega_{n_\chi}^2 V_g/g$$

Design parameters are bandwidth separation W_χ and damping ratio ζ_χ

Yaw Damper



- The rudder is used to counter act the yaw rate caused by adverse yaw
- The washout filter makes it so that the yaw damper only counteracts high-frequency yaw rate
- The washout filter is similar to a dirty derivative with gain τ_r

Lateral Autopilot - Summary

If model is known, the the design parameters are

Inner Loop (roll attitude hold)

- ω_{n_ϕ} - Error in roll when aileron just saturates
- ζ_ϕ - Damping ratio for roll attitude loop

Outer Loop (course hold)

- $W_x > 1$ - Bandwidth separation between roll and course loops
- ζ_x - Damping ratio for course hold loop

Yaw damper (if rudder is available)

- τ_r - cut off frequency for wash-out filter
- k_r - gain for yaw damper

Lateral Autopilot – In Flight Tuning

If model is not known, and autopilot must be tuned in flight, then the following gains are tuned one at a time, in this specific order:

Inner Loop (roll attitude hold)

- k_{d_ϕ} - Increase k_{d_ϕ} until onset of instability, and then back off by 20%
- k_{p_ϕ} - Tune k_{p_ϕ} to get acceptable step response

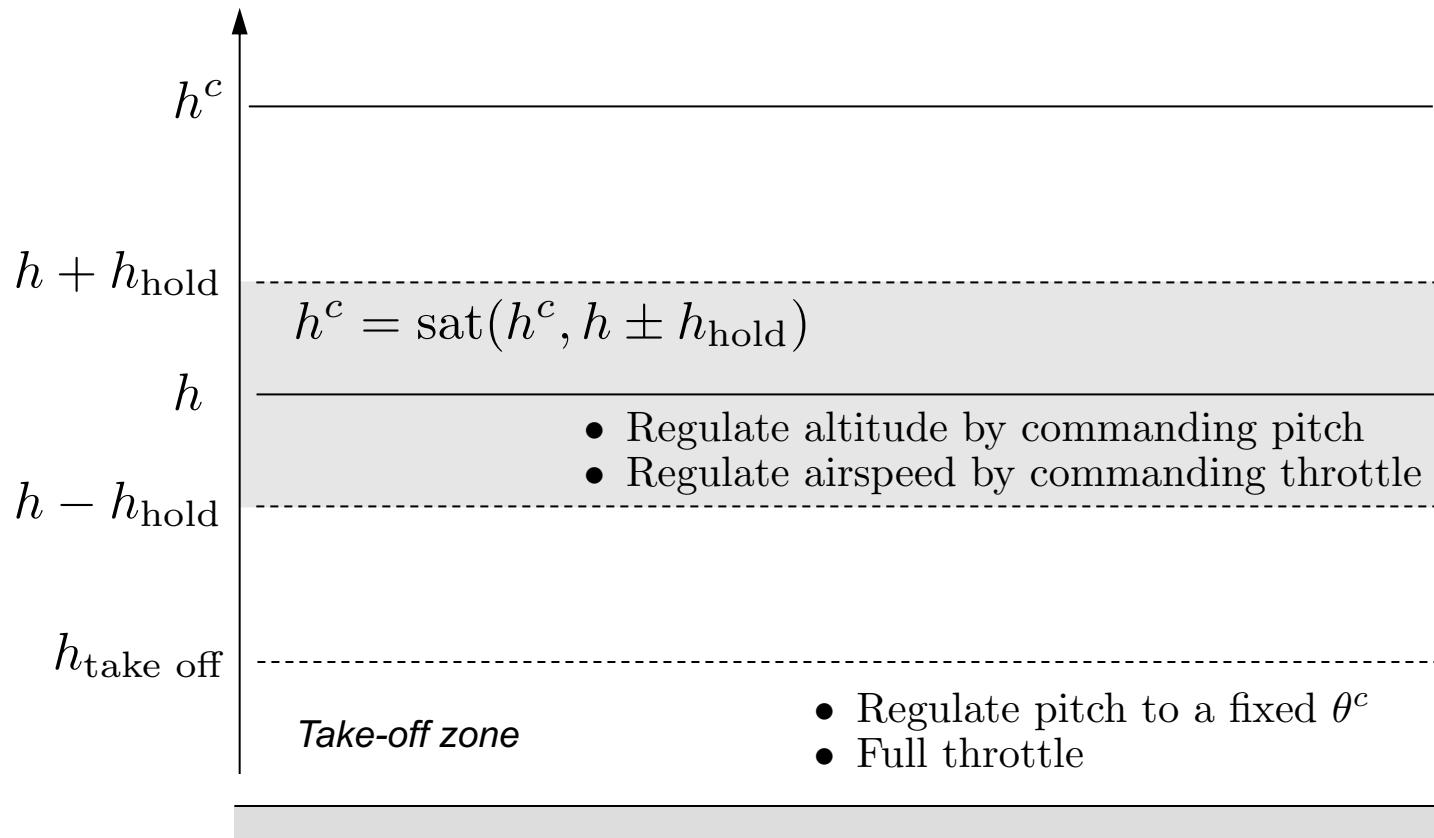
Outer Loop (course hold)

- k_{p_x} - Tune k_{p_x} to get acceptable step response
- k_{i_x} - Tune k_{i_x} to remove steady state error

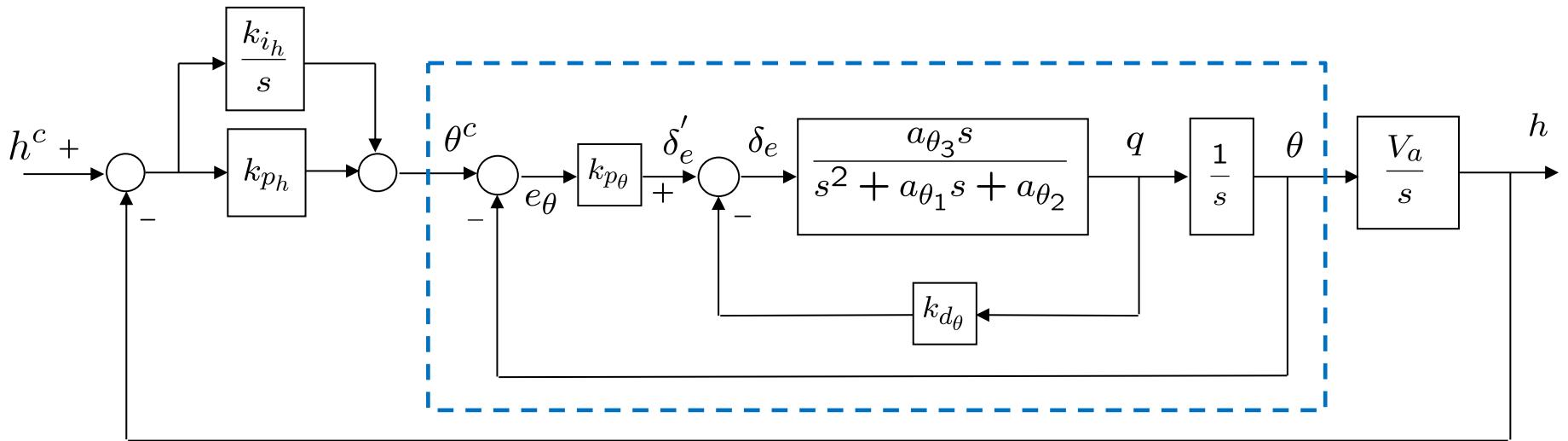
Sideslip hold (if rudder is available)

- τ_r - Tune τ_r to get acceptable step response
- k_r - Tune k_r to remove steady state error

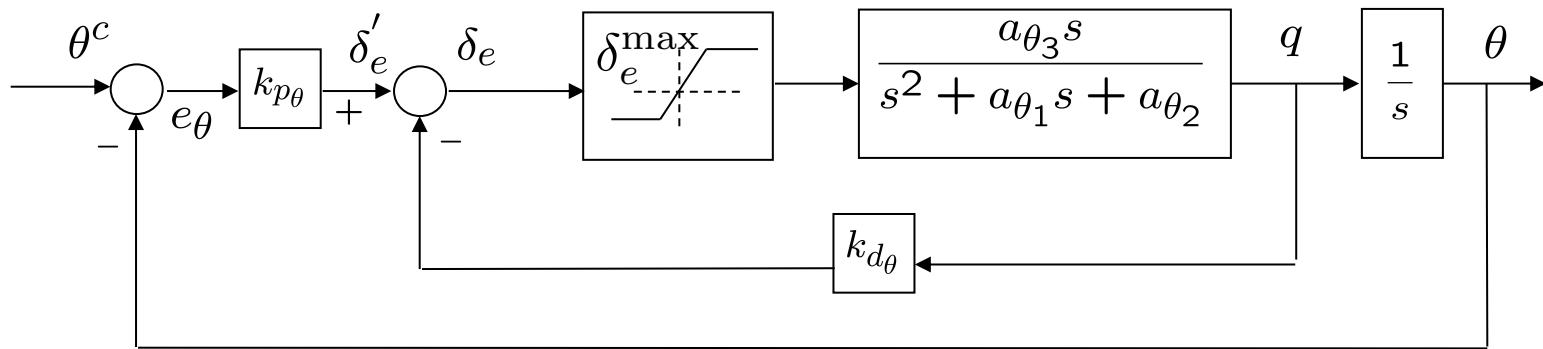
Longitudinal Flight Regimes



Altitude Hold Using Commanded Pitch



Pitch Attitude Hold



$$H_{\theta/\theta^c}(s) = \underbrace{\frac{k_{p\theta} a_{\theta_3}}{s^2 + (a_{\theta_1} + k_{d\theta} a_{\theta_3})s + (a_{\theta_2} + k_{p\theta} a_{\theta_3})}}_{\text{Closed Loop TF}} = \underbrace{\frac{K_{\theta_{DC}} \omega_{n_\theta}^2}{s^2 + 2\zeta_\theta \omega_{n_\theta} s + \omega_{n_\theta}^2}}_{\text{Note: Non-unity DC Gain}}$$

Equating coefficients, the gains are given by

$$k_{p\theta} = \frac{\omega_{n_\theta}^2 - a_{\theta_2}}{a_{\theta_3}}$$

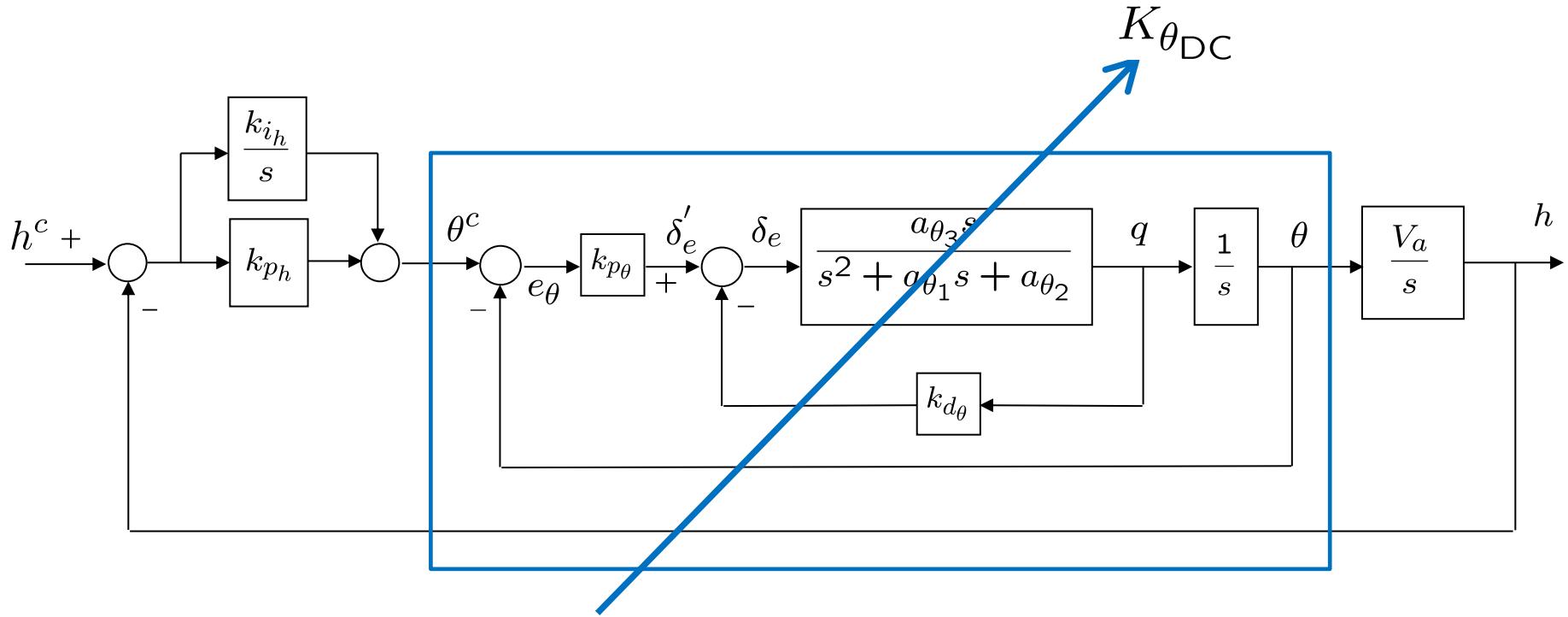
$$k_{d\phi} = \frac{2\zeta_\theta \omega_{n_\theta} - a_{\theta_1}}{a_{\theta_3}}$$

Design parameters are ω_{n_θ} and ζ_θ

The DC gain is

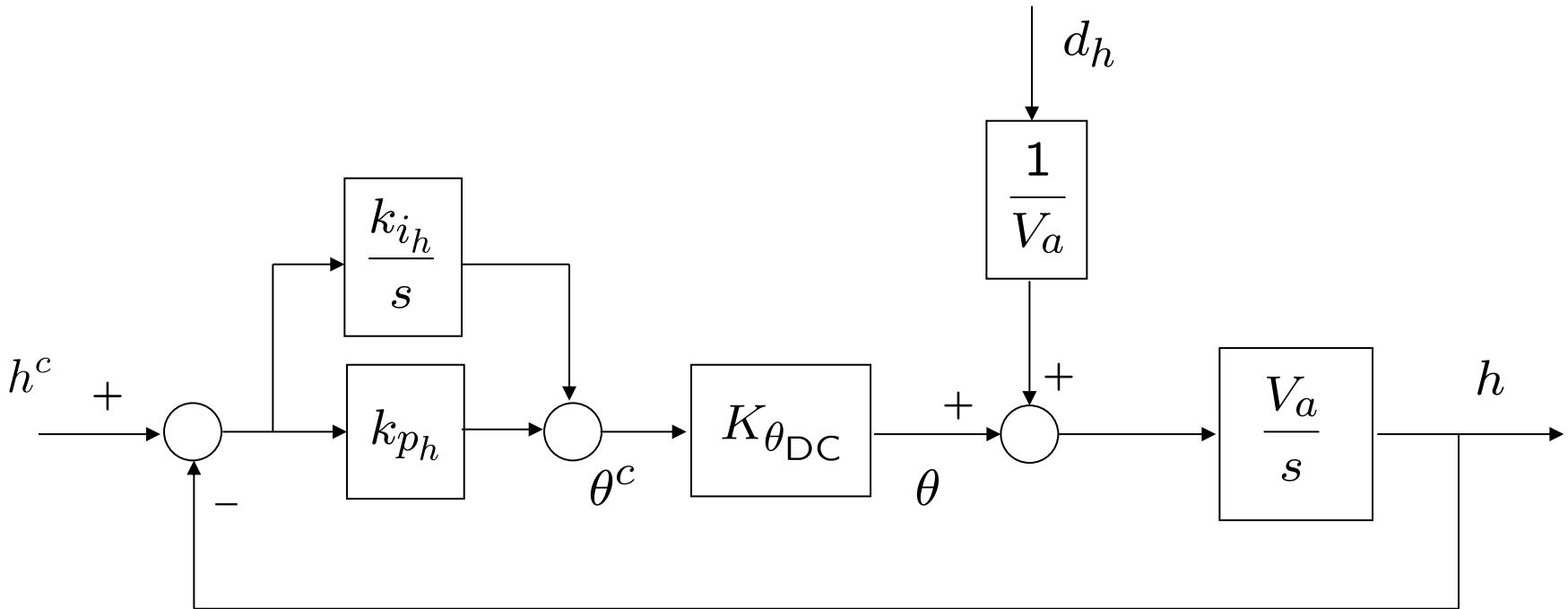
$$K_{\theta_{DC}} = \frac{k_{p\theta} a_{\theta_3}}{a_{\theta_2} + k_{p\theta} a_{\theta_3}}$$

Altitude Hold Using Commanded Pitch



Provided pitch loop functions as intended, we can simplify the inner-loop dynamics to $\theta \approx K_{\theta_{DC}} \theta^c$

Altitude from Pitch – Simplified



$$H(s) = \left(\frac{K_{\theta_{DC}} V_a k_{ph} s + K_{\theta_{DC}} V_a k_{ih}}{s^2 + K_{\theta_{DC}} V_a k_{ph} s + K_{\theta_{DC}} V_a k_{ih}} \right) h^c(s) + \left(\frac{s}{s^2 + K_{\theta_{DC}} V_a k_{ph} s + K_{\theta_{DC}} V_a k_{ih}} \right) d_h(s)$$

A PI control on altitude ensures that h tracks constant h^c with zero steady-state error and rejects constant disturbances

Altitude from Pitch Gain Calculations

Equating the transfer functions

$$H_{h/h^c}(s) = \frac{K_{\theta_{DC}} V_a k_{p_h} s + K_{\theta_{DC}} V_a k_{i_h}}{s^2 + K_{\theta_{DC}} V_a k_{p_h} s + K_{\theta_{DC}} V_a k_{i_h}} \quad \text{and} \quad H_c(s) = \frac{2\zeta\omega_n s + \omega_n^2}{s^2 + 2\zeta\omega_n s + \omega_n^2}$$

gives the coefficients

$$k_{i_h} = \frac{\omega_{n_h}^2}{K_{\theta_{DC}} V_a}$$

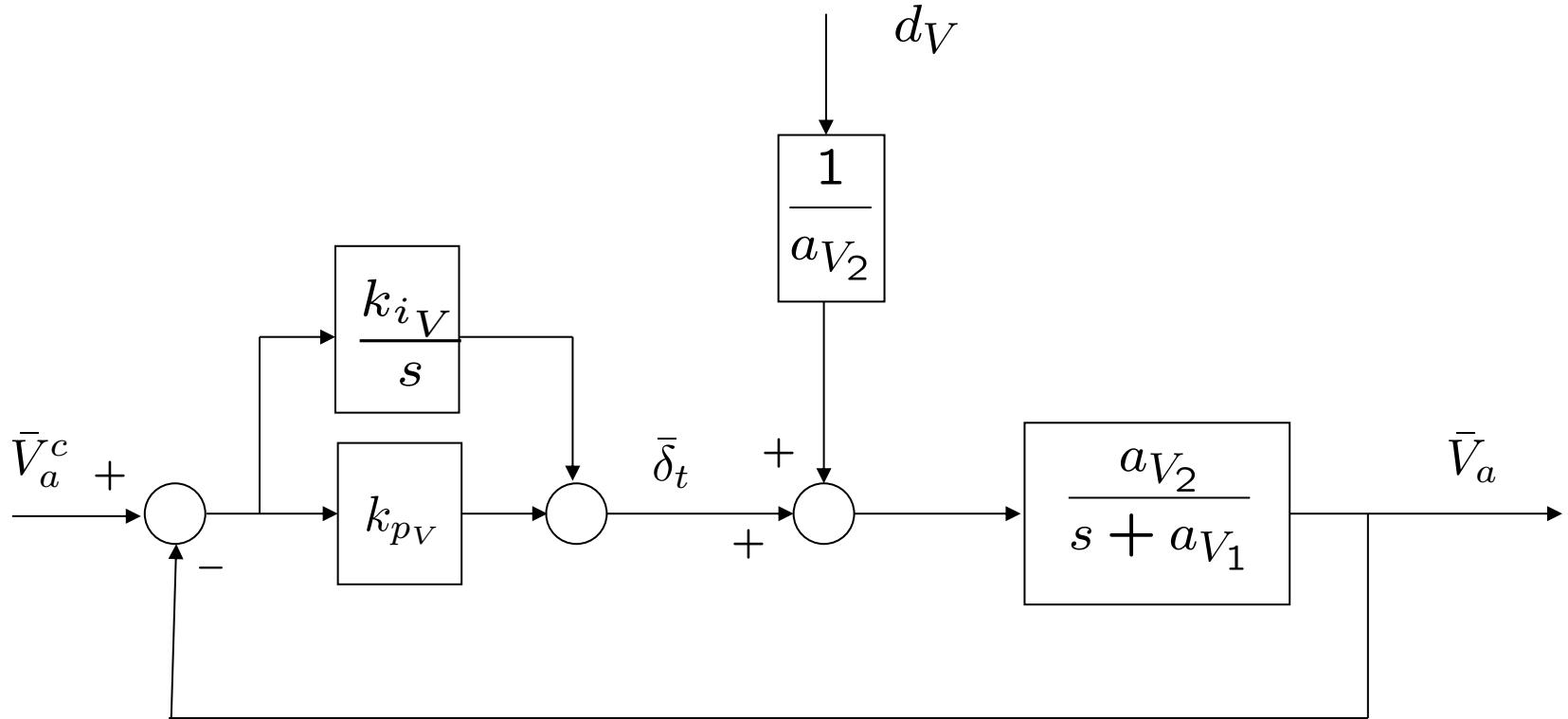
$$k_{p_h} = \frac{2\zeta_h \omega_{n_h}}{K_{\theta_{DC}} V_a}$$

where bandwidth separation is achieved by selecting

$$\omega_{n_h} = \frac{1}{W_h} \omega_{n_\theta}$$

Design parameters are bandwidth separation W_h and damping ratio ζ_θ

Airspeed Hold Using Throttle



$$V_a = \left(\frac{a_{V_2}(k_{p_V}s + k_{i_V})}{s^2 + (a_{V_1} + a_{V_2}k_{p_V})s + a_{V_2}k_{i_V}} \right) V_a^c + \left(\frac{s}{s^2 + (a_{V_1} + a_{V_2}k_{p_V})s + a_{V_2}k_{i_V}} \right) d_V$$

A PI control on the throttle-to-airspeed loop ensures that V_a tracks a constant V_a^c with zero steady-state error and rejects constant disturbances

Airspeed from Throttle Gain Calculations

Equating the transfer functions and their coefficients

$$H_{V_a/V_a^c}(s) = \left(\frac{a_{V_2} k_{p_V} s + a_{V_2} k_{i_V}}{s^2 + (a_{V_1} + a_{V_2} k_{p_V})s + a_{V_2} k_{i_V}} \right) = \frac{\varrho 2\zeta \omega_n s + \omega_n^2}{s^2 + 2\zeta \omega_n s + \omega_n^2}$$

gives the gains

$$k_{i_V} = \frac{\omega_{n_V}^2}{a_{V_2}}$$

$$k_{p_V} = \frac{2\zeta_V \omega_{n_V} - a_{V_1}}{a_{V_2}}$$

Design parameters are natural frequency ω_{n_V} and damping ratio ζ_V

The control signal is

$$\begin{aligned}\delta_t &= \delta_t^* + \bar{\delta}_t \\ &= \delta_t^* + k_{p_V} (V_a^c - V_a) + k_{i_V} \int (V_a^c - V_a) dt\end{aligned}$$

Note that if δ_t^* is imprecisely known, the integrator will compensate.

Python Implementation

```
class autopilot:
    def __init__(self, ts_control):
        self.roll_from_aileron = pdControlWithRate(kp, kd, limit)
        self.course_from_roll = piControl(kp, ki, Ts, limit)
        self.yaw_damper = transferFunction(num, den, Ts)
        self.pitch_from_elevator = pdControlWithRate(kp, kd, limit)
        self.altitude_from_pitch = piControl(kp, ki, Ts, limit)
        self.airspeed_from_throttle = piControl(kp, ki, Ts, limit)

    def update(self, cmd, state):
        # lateral autopilot
        chi_c = wrap(cmd.course_command, state.chi)
        phi_c = self.saturate(
            cmd.phi_feedforward + self.course_from_roll.update(chi_c, state.chi)
            -np.radians(30), np.radians(30))
        delta_a = self.roll_from_aileron.update(phi_c, state.phi, state.p)
        delta_r = self.yaw_damper.update(state.r)

        # longitudinal autopilot
        # saturate the altitude command
        h_c = self.saturate(cmd.altitude_command,
                            state.h - AP.altitude_zone, state.h + AP.altitude_zone)
        theta_c = self.altitude_from_pitch.update(h_c, state.h)
        delta_e = self.pitch_from_elevator.update(theta_c, state.theta, state.q)
        delta_t = self.airspeed_from_throttle.update(cmd.airspeed_command,
                                                    state.Va)
        delta_t = self.saturate(delta_t, 0.0, 1.0)

    return delta, self.commanded_state
```

Wrap Function

```
import numpy as np

def wrap(chi_1, chi_2):
    while chi_1 - chi_2 > np.pi:
        chi_1 = chi_1 - 2.0 * np.pi
    while chi_1 - chi_2 < -np.pi:
        chi_1 = chi_1 + 2.0 * np.pi
    return chi_1
```

Longitudinal Autopilot - Summary

If model is known, the the design parameters are

Inner Loop (pitch attitude hold)

- $\omega_{n\theta}$ - natural frequency for pitch.
- ζ_θ - Damping ratio for pitch attitude loop.

Altitude Hold Outer Loop

- $W_h > 1$ - Bandwidth separation between pitch and altitude loops.
- ζ_h - Damping ratio for altitude hold loop.

Airspeed Hold Outer Loop

- $W_{V_2} > 1$ - Bandwidth separation between pitch and airspeed loops.
- ζ_{V_2} - Damping ratio for airspeed hold loop.

Throttle hold (inner loop)

- ω_{n_V} - Natural frequency for throttle loop.
- ζ_V - Damping ratio for throttle loop.

Longitudinal Autopilot – In Flight Tuning

If model is not known, and autopilot must be tuned in flight, then the following gains are tuned one at a time, in this specific order:

Inner Loop (pitch attitude hold)

- $k_{d\theta}$ - Increase $k_{d\theta}$ until onset of instability, and then back off by 20%.
- $k_{p\theta}$ - Tune $k_{p\theta}$ to get acceptable step response.

Altitude Hold Outer Loop

- k_{ph} - Tune k_{ph} to get acceptable step response.
- k_{ih} - Tune k_{ih} to remove steady state error.

Airspeed Hold Outer Loop

- k_{pv_2} - Tune k_{pv_2} to get acceptable step response.
- k_{iv_2} - Tune k_{iv_2} to remove steady state error.

Throttle hold (inner loop)

- k_{pv} - Tune k_{pv} to get acceptable step response.
- k_{iv} - Tune k_{iv} to remove steady state error.

PID Loop Implementation

$$u(t) = k_p e(t) + k_i \int_{-\infty}^t e(\tau) d\tau + k_d \frac{de}{dt}(t) \quad \text{PID continuous time}$$

$$e(t) = y^c(t) - y(t)$$

Taking Laplace transform...

$$U(s) = k_p E(s) + k_i \frac{E(s)}{s} + k_d s E(s)$$

Use bandwidth-limited differentiator to reduce noise

$$U(s) = k_p E(s) + k_i \frac{E(s)}{s} + k_d \frac{s}{\tau s + 1} E(s)$$

PID Loop Implementation

$$s \mapsto \frac{2}{T_s} \left(\frac{1 - z^{-1}}{1 + z^{-1}} \right) \quad \text{Tustin's rule or trapezoidal rule}$$

$$I(z) = \frac{T_s}{2} \left(\frac{1 + z^{-1}}{1 - z^{-1}} \right) E(z) \quad \text{Integrator term}$$

$$I[n] = I[n - 1] + \frac{T_s}{2} (E[n] + E[n - 1])$$

$$D(z) = \frac{\frac{2}{T_s} \left(\frac{1-z^{-1}}{1+z^{-1}} \right)}{\frac{2\tau}{T_s} \left(\frac{1-z^{-1}}{1+z^{-1}} \right) + 1} E(z) \quad \text{Differentiator term}$$

$$= \frac{\left(\frac{2}{2\tau+T_s} \right) (1 - z^{-1})}{1 - \left(\frac{2\tau-T_s}{2\tau+T_s} \right) z^{-1}} E(z)$$

$$D[n] = \left(\frac{2\tau - T_s}{2\tau + T_s} \right) D[n - 1] + \left(\frac{2}{2\tau + T_s} \right) (E[n] - E[n - 1])$$

Integrator Anti-wind-up

$$u_{\text{unsat}}^- = k_p e + k_d D + k_i I^- \quad \text{control before anti-wind-up update}$$

$$u_{\text{unsat}}^+ = k_p e + k_d D + k_i I^+ \quad \text{control after anti-wind-up update}$$

$$I^+ = I^- + \Delta I \quad \Delta I \text{ is anti-wind-up update}$$

$$u_{\text{unsat}}^+ = u_{\text{unsat}}^- + k_i \Delta I \quad \text{subtracting top two equations}$$

Let $u_{\text{unsat}}^+ = u$ value of control after saturation is applied

$$\Delta I = \frac{1}{k_i} (u - u_{\text{unsat}}^-) \quad \text{solving...}$$

$$I^+ = I^- + \frac{1}{k_i} (u - u_{\text{unsat}}^-)$$

PID Implementation (Python)

```
class PIDControl:
    def __init__(self, kp=0.0, ki=0.0, kd=0.0, Ts=0.01, sigma=0.05, limit=1.0):
        self.kp = kp
        self.ki = ki
        self.kd = kd
        self.Ts = Ts
        self.limit = limit
        self.integrator = 0.0
        self.error_delay_1 = 0.0
        self.error_dot_delay_1 = 0.0
        self.y_dot = 0.0
        self.y_delay_1 = 0.0
        self.y_dot_delay_1 = 0.0
    # gains for differentiator
        self.a1 = (2.0 * sigma - Ts) / (2.0 * sigma + Ts)
        self.a2 = 2.0 / (2.0 * sigma + Ts)

    def update(self, y_ref, y, reset_flag=False):
        if reset_flag is True:
            self.integrator = 0.0
            self.error_delay_1 = 0.0
            self.y_dot = 0.0
            self.y_delay_1 = 0.0
            self.y_dot_delay_1 = 0.0
        # compute the error
        error = y_ref - y
        # update the integrator using trapazoidal rule
        self.integrator = self.integrator \
            + (self.Ts/2) * (error + self.error_delay_1)
        # update the differentiator
        error_dot = self.a1 * self.error_dot_delay_1 \
            + self.a2 * (error - self.error_delay_1)
        # PID control
        u = self.kp * error \
            + self.ki * self.integrator \
            + self.kd * error_dot
        # saturate PID control at limit
        u_sat = self._saturate(u)
        # integral anti-windup
        # adjust integrator to keep u out of saturation
        if np.abs(self.ki) > 0.0001:
            self.integrator = self.integrator \
                + (self.Ts / self.ki) * (u_sat - u)
        # update the delayed variables
        self.error_delay_1 = error
        self.error_dot_delay_1 = error_dot
```

PID Implementation (Matlab)

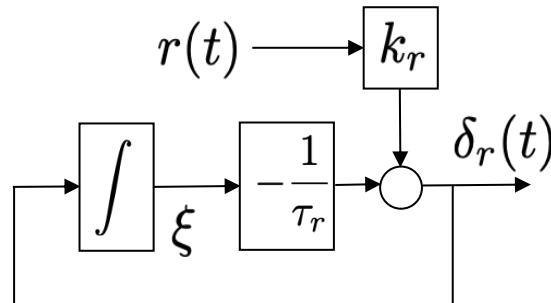
```
1 function u = pidloop(y_c, y, flag, kp, ki, kd, limit, Ts, tau)
2 persistent integrator;
3 persistent differentiator;
4 persistent error_d1;
5 if flag==1, % reset (initialize) persistent variables
6 % when flag==1
7 integrator = 0;
8 differentiator = 0;
9 error_d1 = 0; % _d1 means delayed by one time step
10 end
11 error = y_c - y; % compute the current error
12 integrator = integrator + (Ts/2)*(error + error_d1);
13 % update integrator
14 differentiator = (2*tau-Ts)/(2*tau+Ts)*differentiator...
15 + 2/(2*tau+Ts)*(error - error_d1);
16 % update differentiator
17 error_d1 = error; % update the error for next time through
18 % the loop
19 u = sat(... % implement PID control
20 kp * error +... % proportional term
21 ki * integrator +... % integral term
22 kd * differentiator,... % derivative term
23 limit... % ensure abs(u)<=limit
24 );
25 % implement integrator anti-windup
26 if ki~=0
27 u_unsat = kp*error + ki*integrator + kd*differentiator;
28 integrator = integrator + Ts/ki * (u - u_unsat);
29 end
30
31 function out = sat(in, limit)
32 if in > limit, out = limit;
33 elseif in < -limit; out = -limit;
34 else out = in;
35 end
```

Yaw Damper Implementation

Yaw damper: $\delta_r(s) = k_r \left(\frac{s}{s + \frac{1}{\tau_r}} \right) r(s)$

In time domain: $\dot{\delta}_r = -\frac{1}{\tau_r} \delta_r + k_r \dot{r}$

Integrating: $\delta_r(t) = -\frac{1}{\tau_r} \int_{-\infty}^t \delta_r(\sigma) d\sigma + k_r r(t)$



State-space form:

$$\dot{\xi} = -\frac{1}{\tau_r} \xi + k_r r$$

$$\delta_r = -\frac{1}{\tau_r} \xi + k_r r.$$

Discrete-time:

$$\xi_k = \xi_{k-1} + T_s \left(-\frac{1}{\tau_r} \xi_{k-1} + k_r r_{k-1} \right)$$

$$\delta_{rk} = -\frac{1}{\tau_r} \xi_k + k_r r_k.$$

Yaw Damper Implementation Alternative

Discrete-time:

$$\xi_k = \xi_{k-1} + T_s \left(-\frac{1}{\tau_r} \xi_{k-1} + k_r r_{k-1} \right)$$
$$\delta_{rk} = -\frac{1}{\tau_r} \xi_k + k_r r_k.$$

```
class yawDamper:  
    def __init__(self, k_r, tau_r, Ts):  
        # set initial conditions  
        self.xi = 0.  
        self.Ts = Ts  
        self.k_r = k_r  
        self.tau_r = tau_r  
  
    def update(self, u):  
        self.xi = self.xi  
            + self.Ts * (-1/self.tau_r * self.xi + self.k_r * r)  
        delta_r = -1/self.tau_r * self.xi + self.k_r * r  
        return delta_r
```

Simulation Project

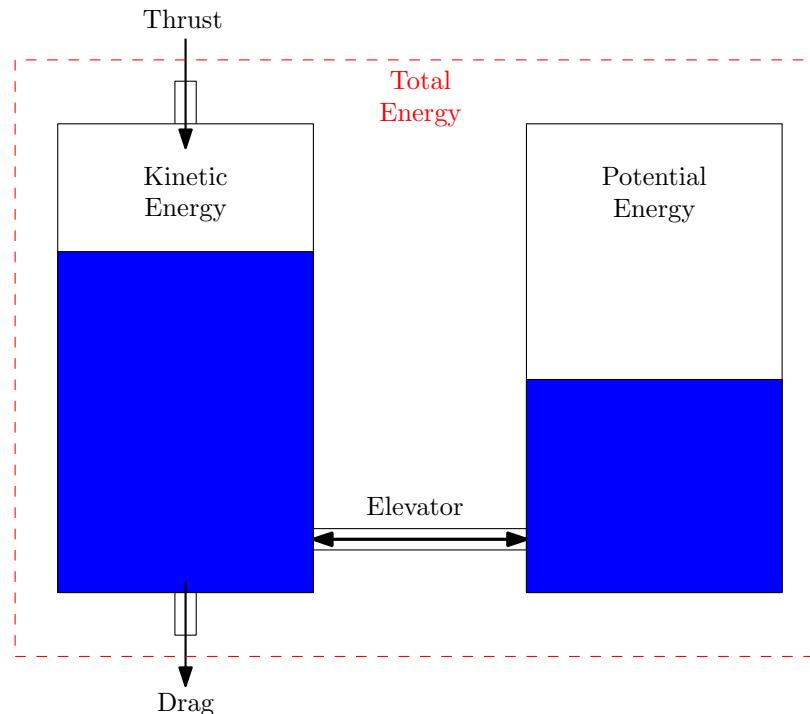
- Step 1.** Roll attitude loop. For Aerosonde model use $V_a = 25 \text{ m/s}$. Put aircraft in trim, and tune step response on roll.
- Step 2.** Tune the yaw damper to keep $\beta \approx 0$.
- Step 3.** Course attitude loop. Tune step response in χ .
- Step 4.** Pitch attitude loop. Tune step response in pitch angle.
- Step 5.** Tune step response in airspeed.
- Step 6.** Tune step response in altitude (after saturation).
- Step 7.** Stress test simulation from take-off. Command steps in altitude, airspeed, course.

Outline

- Successive Loop Closure
- Total Energy Control
- LQR Control

Total Energy Control

- Developed in the 1980's by Antonius Lambregts
- Based on energy manipulation techniques from the 1950's
- Control the energy of the system instead of the altitude and airspeed



Total Energy Control

- Kinetic Energy: $E_K \triangleq \frac{1}{2}mV_a^2$
- Potential Energy: $E_P \triangleq mgh$
- Total Energy: $E_T \triangleq E_P + E_K$
- Energy Difference: $E_D \triangleq E_P - E_K$

Total Energy Control

Original TECS proposed by Lambregts is based on energy rates:

- $T^c = T_D + k_{p,t} \dot{E}_t + k_{i,t} \int_{t_0}^t \dot{\tilde{E}}_t \delta_\tau$
 - T_D is thrust needed to counteract drag
 - PI controller based on total energy rate
- $\theta^c = k_{p,\theta} \dot{E}_d + k_{i,\theta} \int_{t_0}^t \dot{\tilde{E}}_d \delta_\tau$
 - PI controller based on energy distribution rate
- Stability shown for linear systems

We will show that the performance of this scheme is less than desirable.

Total Energy Control

If the thrust needed to counteract drag is unknown, then one possibility is to use an integrator to find T_D :

- $T^c = k_{p,t} \tilde{E}_t + k_{i,t} \int \tilde{E}_t d\tau + k_{d,t} \dot{E}_t$
 - PID controller based on total energy (not energy rate)
- $\theta^c = k_{p,\theta} \tilde{E}_d + k_{i,\theta} \int \tilde{E}_d d\tau + k_{d,\theta} \dot{E}_d$
 - PID controller based on energy distribution (not rate)

Total Energy Control

Nonlinear re-derivation:

- Error Definitions

$$\begin{aligned}\tilde{E}_K &= \frac{1}{2}m \left((V_a^d)^2 - V_a^2 \right) \\ \tilde{E}_P &= mg (h^d - h)\end{aligned}$$

- Lyapunov Function

$$V = \frac{1}{2}\tilde{E}_T^2 + \frac{1}{2}\tilde{E}_D^2$$

- Controller

$$\begin{aligned}T^c &= D + \frac{\tilde{E}_T^d}{V_a} + k_T \frac{\tilde{E}_T}{V_a} \\ \gamma^c &= \sin^{-1} \left(\frac{\dot{h}^d}{V_a} + \frac{1}{2mgV_a} \left(-k_1 \tilde{E}_K + k_2 \tilde{E}_P \right) \right)\end{aligned}$$

Total Energy Control

- Original:
$$T^c = D + k_{p,t} \frac{\dot{E}_T}{mgV_a} + k_{i,t} \frac{\tilde{E}_T}{mgV_a}$$
- Nonlinear:
$$T^c = D + \frac{\dot{E}_T^d}{V_a} + k_T \frac{\tilde{E}_T}{V_a}$$

Similar if $k_{p,T} = mg$ and $k_{i,T} = mgk_T$.

The nonlinear controller uses the desired energy rate.

Total Energy Control

- Modified Original (Ardupilot):

$$\theta^c = \frac{k_{p,\theta}}{V_a mg} \left((2 - k) \dot{E}_P - k \dot{E}_K \right) + \frac{k_{i,\theta}}{V_a mg} \tilde{E}_D$$
$$k \in [0, 2]$$

- Nonlinear:

$$\gamma^c = \sin^{-1} \left(\frac{\dot{h}^d}{V_a} + \frac{1}{2mgV_a} \left(-k_1 \tilde{E}_K + k_2 \tilde{E}_P \right) \right)$$

$$k_1 \triangleq |k_T - k_D|$$

$$k_2 \triangleq k_T + k_D$$

$$0 < k_T \leq k_D$$

Lyapunov derivation suggests potential energy error should be weighted more than kinetic energy

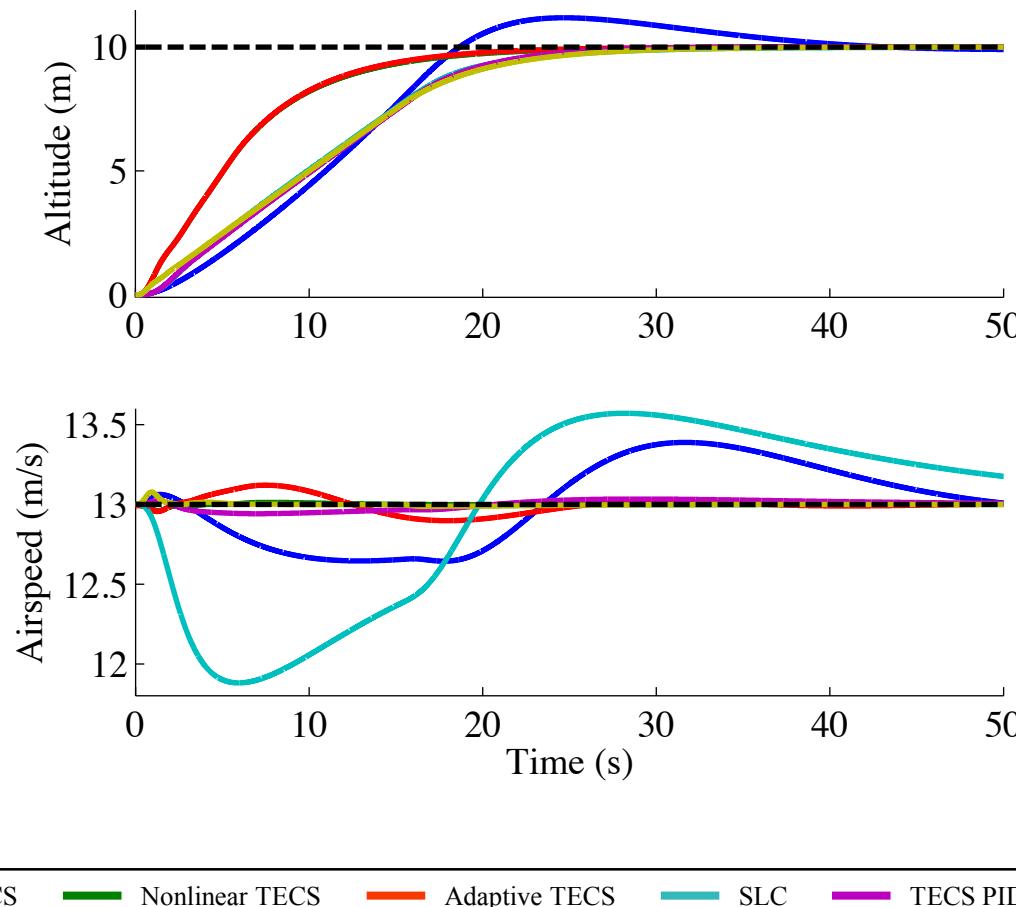
Total Energy Control

If the drag is unknown, then we can add an adaptive estimate:

$$\begin{aligned} T^c &= \hat{D} + \boldsymbol{\Phi}^\top \hat{\boldsymbol{\Psi}} + \frac{\dot{E}_T^d}{V_a} + k_T \frac{\tilde{E}_T}{V_a} \\ \gamma^c &= \sin^{-1} \left(\frac{\dot{h}^d}{V_a} + \frac{1}{2mgV_a} \left(-k_1 \tilde{E}_K + k_2 \tilde{E}_P \right) \right) \\ \dot{\hat{\boldsymbol{\Psi}}} &= \left(\Gamma_T \tilde{E}_T - \Gamma_D \tilde{E}_D \right) \boldsymbol{\Phi} V_a \end{aligned}$$

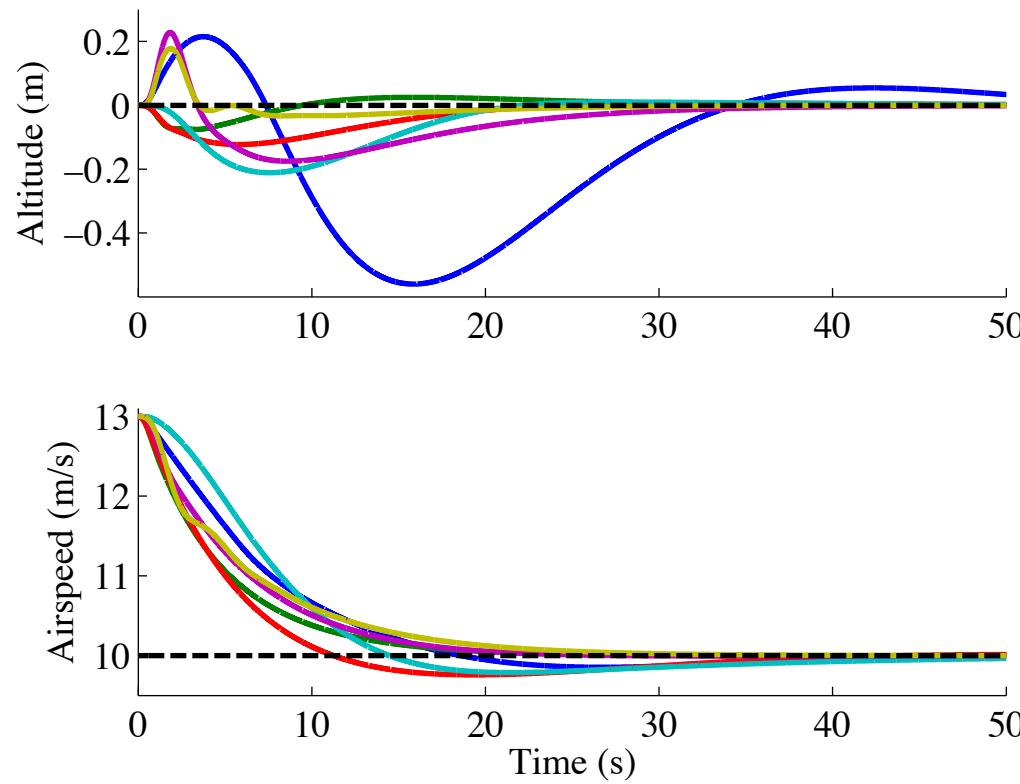
Total Energy Control

Step in Altitude, Constant Airspeed



Total Energy Control

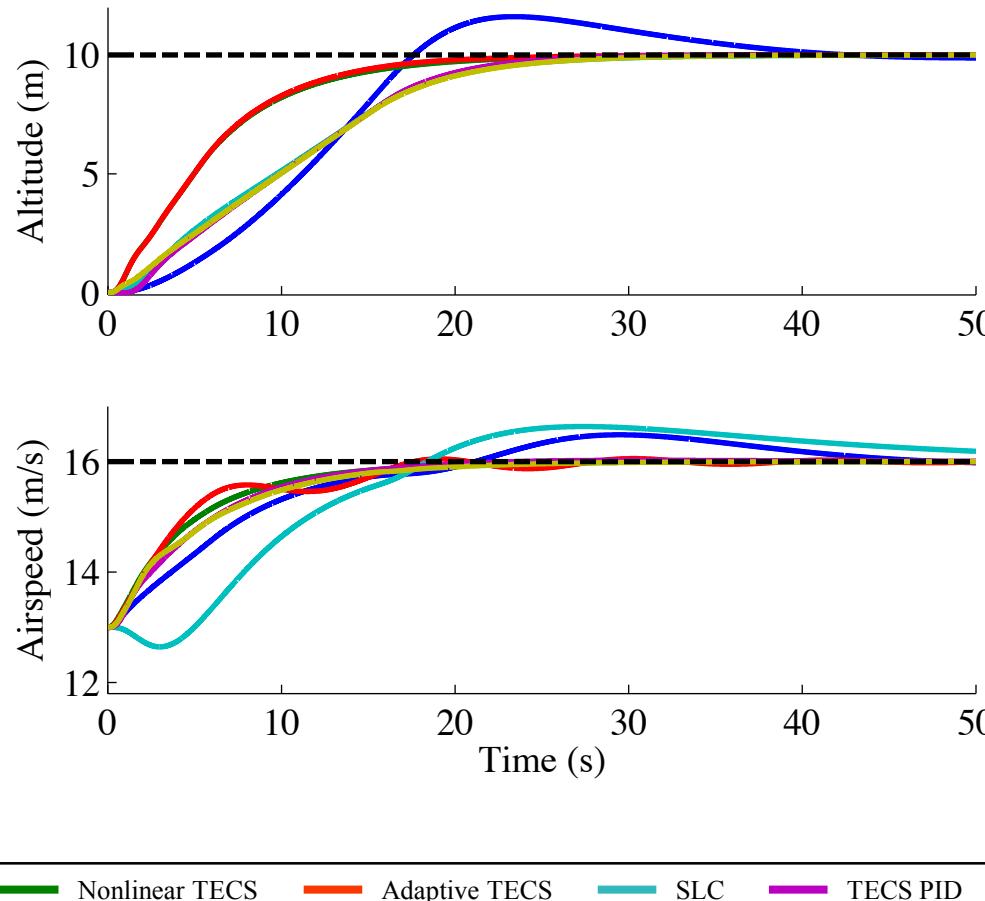
Step in Airspeed, Constant Altitude



— Original TECS — Nonlinear TECS — Adaptive TECS — SLC — TECS PID — ArduPilot PID

Total Energy Control

Step in Altitude and Airspeed



Total Energy Control

- Observations
 - TECS seems to work better than successive loop closure.
 - Removes needs for different flight modes.
 - Nonlinear TECS seems to better, but the Ardupilot controller works very well.

Outline

- Successive Loop Closure
- Total Energy Control
- LQR Control

LQR Control

Augment the States with an Integrator.

Given the state space system

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

$$z = Hx$$

where z represents the controlled output. Suppose that the objective is to drive z to a reference signal z_r . Augment the state with the integrator

$$x_I = \int_{-\infty}^t (z(\tau) - z_r) d\tau.$$

Therefore

$$\dot{x}_I = Hx - z_r = H(x - x_r)$$

LQR Control

Augment the States with an Integrator. (cont)

Defining the augmented state as $\xi = (x^\top, x_I^\top)^\top$, results in the augmented state space equations

$$\dot{\xi} = \bar{A}\xi + \bar{B}u,$$

where

$$\bar{A} = \begin{pmatrix} A & 0 \\ H & 0 \end{pmatrix} \quad \bar{B} = \begin{pmatrix} B \\ 0 \end{pmatrix}.$$

LQR Control

Linear Quadratic Regulator Theory

Given the state space equation

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

and the symmetric positive semi-definite matrix Q , and the symmetric positive definite matrix R , the LQR problem is to minimize the cost index

$$J(x_0) = \min_{u(t)} \int_0^{\infty} x^{\top}(\tau) Q x(\tau) + u^{\top}(\tau) R u(\tau) d\tau.$$

If (A, B) is controllable, and $(A, Q^{1/2})$ is observable, then a unique optimal control exists and is given in linear feedback form as

$$u^*(t) = -K_{lqr}x(t).$$

LQR Control

Linear Quadratic Regulator Theory (cont)

The LQR gain is given by

$$K_{lqr} = R^{-1}B^\top P,$$

where P is the symmetric positive definite solution of the Algebraic Riccati Equation

$$PA + A^\top P + Q - PBR^{-1}B^\top P = 0.$$

LQR Control

Linear Quadratic Regulator Theory (cont)

It should be noted that K_{lqr} is the optimal feedback gains given Q and R . The controller is tuned by changing Q and R .

Typically we choose Q and R to be diagonal matrices

$$Q = \begin{pmatrix} q_1 & 0 & \dots & 0 \\ 0 & q_2 & \dots & 0 \\ \vdots & & \ddots & \vdots \\ 0 & 0 & \dots & q_n \end{pmatrix} \quad R = \begin{pmatrix} r_1 & 0 & \dots & 0 \\ 0 & r_2 & \dots & 0 \\ \vdots & & \ddots & \vdots \\ 0 & 0 & \dots & r_m \end{pmatrix},$$

where n is the number of states, m is the number of inputs, and $q_i \geq 0$ ensures Q is positive semi-definite, and $r_i > 0$ ensure R is positive definite.

LQR Control

Lateral Autopilot

As derived in Chapter 5, the state space equations for the lateral equation of motion are given by

$$\dot{x}_{lat} = A_{lat}x_{lat} + B_{lat}u_{lat},$$

where $x_{lat} = (v, p, r, \phi, \psi)^\top$ and $u_{lat} = (\delta_a, \delta_r)^\top$.

The objective of the lateral autopilot is to drive course χ to commanded course χ_c . Therefore, we augment the state with

$$x_I = \int (\chi - \chi_c) dt.$$

Since $\chi \approx \psi$, we approximate x_I as

$$x_I = \int (H_{lat}x_{lat} - \chi_c) dt,$$

where $H_{lat} = (0, 0, 0, 0, 1)$.

LQR Control

Lateral Autopilot (cont)

The augmented lateral state equations are therefore

$$\dot{\xi}_{lat} = \bar{A}_{lat}\xi_{lat} + \bar{B}_{lat}u_{lat},$$

where

$$\xi_{lat} = (\tilde{v}, p, r, \phi, \tilde{\chi}, \int \tilde{\chi})^\top$$

$$\bar{A}_{lat} = \begin{pmatrix} A_{lat} & 0 \\ H_{lat} & 0 \end{pmatrix} \quad \bar{B}_{lat} = \begin{pmatrix} B_{lat} \\ 0 \end{pmatrix}$$

where

$$\tilde{v} = (V_a - V_a^c) * \sin \beta, \quad \tilde{\chi} = \chi - \chi^c$$

The LQR controller designed using

$$Q = \text{diag}([q_v, q_p, q_r, q_\phi, q_\chi, q_I])$$

$$R = \text{diag}([r_{\delta_a}, r_{\delta_r}]).$$

Hints: Since the goal is to drive $\tilde{\chi}$ and $\int \tilde{\chi}$ to zero, it is best to place low weights on v , p , and r relative to the other variables.

LQR Control

Longitudinal Autopilot

As derived in Chapter 5, the state space equations for the longitudinal equations of motion are given by

$$\dot{x}_{lon} = A_{lon}x_{lon} + B_{lon}u_{lon},$$

where $x_{lon} = (u, w, q, \theta, h)^\top$ and $u_{lat} = (\delta_e, \delta_t)^\top$.

The objective of the longitudinal autopilot is to drive altitude h to commanded altitude h^c , and airspeed V_a to commanded airspeed V_a^c . Therefore, we augment the state with

$$x_I = \begin{pmatrix} \int(h - h^c)dt \\ \int(V_a - V_a^c)dt \end{pmatrix} = \int \left(H_{lon}x_{lon} - \begin{pmatrix} h^c \\ V_a^c \end{pmatrix} \right) dt.$$

Therefore

$$\dot{x}_I = H_{lon}x_{lon} - \begin{pmatrix} h^c \\ V_a^c \end{pmatrix} = H_{lon}(x_{lon} - x^c)$$

where

$$H_{lon} = \begin{pmatrix} 0 & 0 & 0 & 0 & 1 \\ \frac{1}{V_a} & \frac{1}{V_a} & 0 & 0 & 0 \end{pmatrix}.$$

LQR Control

Longitudinal Autopilot (cont)

The augmented longitudinal state equations are therefore

$$\dot{\xi}_{lon} = \bar{A}_{lon}\xi_{lon} + \bar{B}_{lon}u_{lon},$$

where

$$\xi_{lon} = (\tilde{u}, \tilde{w}, q, \theta, \tilde{h}, \int \tilde{h}, \int \tilde{V}_a)^\top$$

$$\bar{A}_{lon} = \begin{pmatrix} A_{lon} & 0 \\ H_{lon} & 0 \end{pmatrix} \quad \bar{B}_{lon} = \begin{pmatrix} B_{lon} \\ 0 \end{pmatrix}$$

where

$$\tilde{u} = (V_a - V_a^c) \cos \alpha, \quad \tilde{w} = (V_a - V_a^c) \sin \alpha, \quad \tilde{h} = h - h^c, \quad \tilde{V}_a = V_a - V_a^c$$

The LQR controller designed using

$$Q = \text{diag}([q_u, q_w, q_q, q_\theta, q_h, q_{Ih}, q_{IV_a}])$$
$$R = \text{diag}([r_{\delta_e}, r_{\delta_t}]).$$

Hints: Since θ need to be allowed to be non-zero, and since pitch rate does not necessarily need to be small, the weights q_θ and q_q should be small.

Python Code

```
import sys
from numpy import array, sin, cos, radians, concatenate, zeros, diag
from scipy.linalg import solve_continuous_are, inv
sys.path.append('..')
import parameters.control_parameters as AP
#from tools.transfer_function import TransferFunction
from tools.wrap import wrap
import chap5.model_coef as M
from message_types.msg_state import MsgState
from message_types.msg_delta import MsgDelta

class Autopilot:
    def __init__(self, ts_control):
        self.Ts = ts_control
        # initialize integrators and delay variables
        self.integratorCourse = 0
        self.integratorAltitude = 0
        self.integratorAirspeed = 0
        self.errorCourseD1 = 0
        self.errorAltitudeD1 = 0
        self.errorAirspeedD1 = 0
        # compute LQR gains
        CrLat = array([[0, 0, 0, 0, 1.0]])
        AAlat = concatenate((
            concatenate((M.A_lat, zeros((5,1))), axis=1),
            concatenate((CrLat, zeros((1,1))), axis=1)),
            axis=0)
        BBlat = concatenate((M.B.lat, zeros((1,2))), axis=0)
        Qlat = diag([.001, .01, .1, 100, 1, 100]) # v, p, r, phi, chi, intChi
        Rlat = diag([1, 1]) # a, r
        Plat = solve_continuous_are(AAlat, BBlat, Qlat, Rlat)
        self.Klat = inv(Rlat) @ BBlat.T @ Plat
        #CrLon = array([[0, 0, 0, 0, 1.0], [1.0, 0, 0, 0, 0]])
        CrLon = array([[0, 0, 0, 0, 1.0], [1/AP.Va0, 1/AP.Va0, 0, 0, 0]])
        AAlon = concatenate((
            concatenate((M.A_lon, zeros((5,2))), axis=1),
            concatenate((CrLon, zeros((2,2))), axis=1)),
            axis=0)
        BBlon = concatenate((M.B.lon, zeros((2,2))), axis=0)
        Qlon = diag([10, 10, .001, .01, 10, 100, 100]) # u, w, q, theta, h, intH
        Rlon = diag([1, 1]) # e, t
        Plon = solve_continuous_are(AAlon, BBlon, Qlon, Rlon)
        self.Klon = inv(Rlon) @ BBlon.T @ Plon
        self.commanded_state = MsgState()
```

Python Code

```
def update(self, cmd, state):
    # lateral autopilot
    errorAirspeed = state.Va - cmd.airspeed_command
    chi_c = wrap(cmd.course_command, state.chi)
    errorCourse = saturate(state.chi - chi_c, -radians(15), radians(15))
    self.integratorCourse = self.integratorCourse + (self.Ts/2) * (errorCourse +
    self.errorCourseD1 = errorCourse
    xLat = array([[errorAirspeed * sin(state.beta)], # v
                  [state.p],
                  [state.r],
                  [state.phi],
                  [errorCourse],
                  [self.integratorCourse]]))
    tmp = -self.Klat @ xLat
    delta_a = saturate(tmp.item(0), -radians(30), radians(30))
    delta_r = saturate(tmp.item(1), -radians(30), radians(30))

    # longitudinal autopilot
    altitude_c = saturate(cmd.altitude_command,
                           state.altitude - 0.2*AP.altitude_zone,
                           state.altitude + 0.2*AP.altitude_zone)
    errorAltitude = state.altitude - altitude_c
    self.integratorAltitude = self.integratorAltitude \
        + (self.Ts/2) * (errorAltitude + self.errorAltitudeD)
    self.errorAltitudeD1 = errorAltitude
    self.integratorAirspeed = self.integratorAirspeed \
        + (self.Ts/2) * (errorAirspeed + self.errorAirspeedD)
    self.errorAirspeedD1 = errorAirspeed
    xLon = array([[errorAirspeed * cos(state.alpha)], # u
                  [errorAirspeed * sin(state.alpha)], # w
                  [state.q],
                  [state.theta],
                  [errorAltitude],
                  [self.integratorAltitude],
                  [self.integratorAirspeed]]])
    tmp = -self.Klon @ xLon
    delta_e = saturate(tmp.item(0), -radians(30), radians(30))
    delta_t = saturate(tmp.item(1), 0.0, 1.0)

    # construct control outputs and commanded states
    delta = MsgDelta(elevator=delta_e,
                     aileron=delta_a,
                     rudder=delta_r,
                     throttle=delta_t)
    self.commanded_state.altitude = cmd.altitude_command
    self.commanded_state.Va = cmd.airspeed_command
```