Chapter 6

Successive Loop Closure
Architecture

Path planner

Path manager

Path following

Autopilot

Unmanned Vehicle

Path Definition

Waypoints

Destination, obstacles

Airspeed, Altitude, Heading, commands

Servo commands

Wind

Map

Status

Tracking error

Position error

State estimator

On-board sensors

\( \hat{x}(t) \)
Outline

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

Open-loop system

\[ u \rightarrow P_1(s) \rightarrow y_1 = u_2 \rightarrow P_2(s) \rightarrow y_2 = u_3 \rightarrow P_3(s) \rightarrow y_3 \]

Closed-loop system

\[ r_3 + \rightarrow C_3(s) \rightarrow C_2(s) \rightarrow C_1(s) \rightarrow u_1 \rightarrow u_2 \rightarrow u_3 \rightarrow y_3 \]

At frequencies below inner-loop bandwidth, approximate CLTF as 1, then design middle loop.
SLC: Two Loops Closed

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

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

Course Control

Roll Control

\[ \chi^c + k_{iX} \frac{s}{s} + \phi^c \]

\[ k_{pX} \]

\[ e_\chi, k_{p_\phi} \]

\[ e_\phi \]

\[ \delta_a \]

\[ \frac{a_{\phi_2}}{s + a_{\phi_1}} \]

\[ \frac{1}{s} \]

\[ \phi \]

\[ \frac{g/V_g}{s} \]

\[ r^c = 0 + -1 \]

\[ k_r \]

\[ \frac{\tau_r s}{\tau_r s + 1} \]

Yaw Damper

Roll Autopilot

\[ H_{\phi/\phi^c}(s) = \frac{k_{p\phi}a_{\phi_2}}{s^2 + (a_{\phi_1} + a_{\phi_2} k_{d\phi}) s + k_{p\phi} a_{\phi_2}} = \frac{\omega_{n\phi}^2}{s^2 + 2\zeta_{\phi} \omega_{n\phi} s + \omega_{n\phi}^2} \]

Closed Loop TF

Canonical 2\textsuperscript{nd}-order TF

Design parameters are \( \omega_{n\phi} \) and \( \zeta_{\phi} \)

Gains are given by

\[
\begin{align*}
    k_{p\phi} &= \frac{\omega_{n\phi}^2}{a_{\phi_2}} \\
    k_{d\phi} &= \frac{2\zeta_{\phi} \omega_{n\phi} - a_{\phi_1}}{a_{\phi_2}}
\end{align*}
\]

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
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

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

Note:

- There is a zero in the response to the course command $\chi^c$
- The presence of the zero at the origin ensures rejection of low-frequency disturbances
With a zero, the canonical second-order TF is given by

\[ H(s) = \frac{2\zeta\omega_n s + \omega_n^2}{s^2 + 2\zeta\omega_n s + \omega_n^2} = \left( \frac{2\zeta}{\omega_n} \right) \frac{\omega_n^2 s}{s^2 + 2\zeta\omega_n s + \omega_n^2} + \frac{\omega_n^2}{s^2 + 2\zeta\omega_n s + \omega_n^2} \]

Note that $\zeta$ has a different effect when the zero is present.
Course Hold Loop

\[ \chi = \frac{(k_{px} g/V_g) s + (k_{ix} g/V_g)}{s^2 + (k_{px} g/V_g) s + (k_{ix} g/V_g)} \chi^c + \frac{(g/V_g) s}{s^2 + (k_{px} g/V_g) s + (k_{ix} g/V_g)} d_\chi \]

Equating coefficients to canonical TF gives:

\[ \omega_{n_\chi}^2 = k_{ix} g/V_g \quad \text{and} \quad 2 \zeta_\chi \omega_{n_\chi} = k_{px} g/V_g \]

or

\[ \omega_{n_\chi} = \frac{1}{W_\chi} \omega_{n_\phi} \quad \quad k_{px} = 2 \zeta_\chi \omega_{n_\chi} V_g/g \quad \quad k_{ix} = \omega_{n_\chi}^2 V_g/g \]

Design parameters are bandwidth separation \( W_\chi \) and damping ratio \( \zeta_\chi \)
The rudder is used to counteract 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 $\tau_r$. 

\[ r^c = 0 + \delta_r \]

\[ \frac{-C_r(s^2 + 2\zeta\omega_n s + \omega_n^2)}{(s + \rho_p)(s^2 + 2\zeta_d\omega_d s + \omega_d^2)} \]

\[ k_r \]

\[ \frac{\tau_r s}{\tau_r s + 1} \]
Lateral Autopilot - Summary

If model is known, the the design parameters are

**Inner Loop (roll attitude hold)**

- $\omega_{n\phi}$ - Error in roll when aileron just saturates
- $\zeta_\phi$ - Damping ratio for roll attitude loop

**Outer Loop (course hold)**

- $W_\chi > 1$ - Bandwidth separation between roll and course loops
- $\zeta_\chi$ - Damping ratio for course hold loop

**Yaw damper (if rudder is available)**

- $\tau_r$ - 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)**

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

**Outer Loop (course hold)**

- $k_{p\chi}$ - Tune $k_{p\chi}$ to get acceptable step response
- $k_{i\chi}$ - Tune $k_{i\chi}$ to remove steady state error

**Sideslip hold (if rudder is available)**

- $\tau_r$ - Tune $\tau_r$ to get acceptable step response
- $k_r$ - Tune $k_r$ to remove steady state error
Longitudinal Flight Regimes

\[
h^c = \text{sat}(h^c, h \pm h_{\text{hold}})
\]

- Regulate altitude by commanding pitch
- Regulate airspeed by commanding throttle

Take-off zone
- Regulate pitch to a fixed $\theta^c$
- Full throttle
Altitude Hold Using Commanded Pitch

\[ h^c + \frac{k_{ih}}{s} \rightarrow k_{pn} \rightarrow \theta^c \rightarrow e_\theta \rightarrow k_{p_\theta} \rightarrow \delta_e' \rightarrow \delta_e \rightarrow \frac{a_{\theta_3}s}{s^2 + a_{\theta_1}s + a_{\theta_2}} \rightarrow q \rightarrow \frac{1}{s} \rightarrow \theta \rightarrow \frac{V_a}{s} \rightarrow h \]
Pitch Attitude Hold

\[ H_{\theta/\theta^c}(s) = \frac{k_{p_{\theta}}a_{\theta_3}}{s^2 + (a_{\theta_1} + k_{d_{\phi}}a_{\theta_3})s + (a_{\theta_2} + k_{p_{\theta}}a_{\theta_3})} = \frac{K_{\theta_{DC}}\omega_{n_{\theta}}^2}{s^2 + 2\zeta_{\theta}\omega_{n_{\theta}}s + \omega_{n_{\theta}}^2} \]

Closed Loop TF

Note: Non-unity DC Gain

Equating coefficients, the gains are given by

\[ k_{p_{\theta}} = \frac{\omega_{n_{\theta}}^2 - a_{\theta_2}}{a_{\theta_3}} \quad k_{d_{\phi}} = \frac{2\zeta_{\theta}\omega_{n_{\theta}} - a_{\theta_1}}{a_{\theta_3}} \]

Design parameters are \( \omega_{n_{\theta}} \) and \( \zeta_{\theta} \)

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 $\frac{\theta^c}{\theta} \approx K_{\theta_{DC}}$
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

\[ H_{h/h^c}(s) = \frac{K_{\theta DC} V_a k_{ph}}{s^2 + K_{\theta DC} V_a k_{ph} s + K_{\theta DC} V_a k_{ih}} \quad \text{and} \quad H_c(s) = \frac{2\zeta \omega_n s + \omega_n^2}{s^2 + 2\zeta \omega_n s + \omega_n^2} \]

gives the coefficients

\[
\begin{align*}
k_{ih} &= \frac{\omega_{n_{ih}}^2}{K_{\theta DC} V_a} \\
k_{ph} &= \frac{2\zeta \omega_{n_{ih}}}{K_{\theta DC} V_a}
\end{align*}
\]

where bandwidth separation is achieved by selecting

\[ \omega_{n_{ih}} = \frac{1}{W_h} \omega_{n_{\theta}} \]

Design parameters are bandwidth separation \( W_h \) and damping ratio \( \zeta_{\theta} \).
Airspeed Hold Using Throttle

\[
V_a = \left( \frac{aV_2 (k_{pv} s + k_{iv})}{s^2 + (aV_1 + aV_2 k_{pv}) s + aV_2 k_{iv}} \right) V_a^c + \left( \frac{s}{s^2 + (aV_1 + aV_2 k_{pv}) s + aV_2 k_{iv}} \right) dV
\]

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

\[ H_{V_a/V_a^c}(s) = \left( \frac{aV_2 k_{pV} s + aV_2 k_{iV}}{s^2 + (aV_1 + aV_2 k_{pV})s + aV_2 k_{iV}} \right) = \frac{2\zeta \omega_n s + \omega_n^2}{s^2 + 2\zeta \omega_n s + \omega_n^2} \]

gives the gains

\[ k_{iV} = \frac{\omega_n^2}{aV_2} \quad \quad k_{pV} = \frac{2\zeta \omega_n - aV_1}{aV_2} \]

Design parameters are natural frequency \( \omega_n \) and damping ratio \( \zeta \)

The control signal is

\[ \delta_t = \delta_t^* + \bar{\delta}_t \]

\[ = \delta_t^* + k_{pV} (V_{a}^c - V_a) + \frac{k_{iV}}{s} (V_{a}^c - V_a) \]
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
Longitudinal Autopilot - Summary

If model is known, the the design parameters are

**Inner Loop (pitch attitude hold)**

- $e_{\theta}^{\text{max}}$ - Error in pitch when elevator just saturates.
- $\zeta_{\theta}$ - Damping ratio for pitch attitude loop.

**Altitude Hold Outer Loop**

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

**Airspeed Hold Outer Loop**

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

**Throttle hold (inner loop)**

- $\omega_{n_V}$ - Natural frequency for throttle loop.
- $\zeta_V$ - 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**

- $k_{p_h}$ - Tune $k_{p_h}$ to get acceptable step response.
- $k_{i_h}$ - Tune $k_{i_h}$ to remove steady state error.

**Airspeed Hold Outer Loop**

- $k_{p_v^2}$ - Tune $k_{p_v^2}$ to get acceptable step response.
- $k_{i_v^2}$ - Tune $k_{i_v^2}$ to remove steady state error.

**Throttle hold (inner loop)**

- $k_{p_v}$ - Tune $k_{p_v}$ to get acceptable step response.
- $k_{i_v}$ - Tune $k_{i_v}$ to remove steady state error.
PID Loop Implementation

\[ u(t) = k_p e(t) + k_i \int_{-\infty}^{t} e(\tau) d\tau + k_d \frac{de}{dt}(t) \]

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

\[ s \mapsto \frac{2}{T_s} \left( \frac{1 - z^{-1}}{1 + z^{-1}} \right) \]  
Tustin’s rule or trapezoidal rule

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

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

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

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

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

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

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

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

\[ \Delta I = \frac{1}{k_i} (u - u_{\text{unsat}}^-) \quad \text{solving…} \]

\[ I^+ = I^- + \frac{1}{k_i} (u - u_{\text{unsat}}^-) \]
PID Implementation

```
function u = pidloop(y_c, y, flag, kp, ki, kd, limit, Ts, tau)
persistent integrator;
persistent differentiator;
persistent error_d1;
if flag==1, % reset (initialize) persistent variables
    integrator = 0;
    differentiator = 0;
    error_d1 = 0; % _d1 means delayed by one time step
end
error = y_c - y; % compute the current error
integrator = integrator + (Ts/2)*error + error_d1; % update integrator
differentiator = (2*tau-Ts)/(2*tau+Ts)*differentiator + 2/(2*tau+Ts)*(error - error_d1); % update differentiator
error_d1 = error; % update the error for next time through % the loop
u = sat(... % implement PID control
    kp * error +...
    ki * integrator +...
    kd * differentiator,... % derivative term
    limit... % ensure abs(u)<=limit
);
% implement integrator anti-windup
if ki~=0
    u_unsat = kp*error + ki*integrator + kd*differentiator;
    integrator = integrator + Ts/ki * (u - u_unsat);
end
function out = sat(in, limit)
if in > limit, out = limit;
elseif in < -limit; out = -limit;
else out = in;
end
```

update with Python
Yaw Damper Implementation

For a general, first-order continuous-time transfer function of the form

\[ H(s) = \frac{U(s)}{Y(s)} = k \frac{n_1 s + n_0}{d_1 s + d_0} \]

we want to be able to calculated the equivalent discrete-time transfer function

\[ H(z) = \frac{U(z)}{Y(z)} = \frac{b_0 + b_1 z^{-1}}{1 + a_1 z^{-1}} \]

for a specific sample rate \( T_s \). Using Tustin’s method (trapezoidal rule) by substituting \( s \rightarrow \frac{2}{T_s} \left( \frac{1 - z^{-1}}{1+z^{-1}} \right) \), we can calculate the coefficients of \( H(z) \) as

\[
\begin{align*}
b_0 &= \frac{k(2n_1 + T_sn_0)}{2d_1 + T_sd_0} \\
b_1 &= -\frac{k(2n_1 - T_sn_0)}{2d_1 + T_sd_0} \\
a_1 &= -\frac{k(2d_1 - T_sd_0)}{2d_1 + T_sd_0}
\end{align*}
\]

By taking the inverse \( z \) transform, we can convert our discrete transfer function to an discrete-time difference equation:

\[ u_k = -a_1 u_{k-1} + b_0 y_k + b_1 y_{k-1} \]
For our yaw damper,

\[ H(s) = \frac{\delta_r(s)}{r(s)} = k_r \left( \frac{\tau_r s}{\tau_r s + 1} \right) = k_r \left( \frac{s}{s + p_w o} \right), \]

where \( k_r = 0.2, p_w o = 0.45 \text{ rad/s, and } T_s = 0.01 \text{ s} \). This gives a discrete-time transfer function of

\[ H(z) = \frac{\delta_r(z)}{r(z)} = \frac{0.1996 - 0.1996 z^{-1}}{1 - 0.9955 z^{-1}} \]

with the corresponding difference equation

\[ \delta_{r,k} = 0.9955 \delta_{r,k-1} + 0.1996 r_k - 0.1996 r_{k-1} \]

The coefficients of \( H(z) \) can be calculated manually as we have done here or they can be computed in MATLAB using the c2d command with the 'tustin' option or in Python using the control.matlab.c2d function. This is especially convenient for higher-order transfer functions.
Simulation Project

- Roll attitude loop. For Aerosonde model use $V_a = 17 \text{ m/s}$. Note that $\phi_{\text{max}}$ is a design parameter. Put aircraft in trim, and command steps on roll.

- Course attitude loop. Command steps in $\chi$. Can by-pass simplified simulink files.

- For sideslip, assume no rudder, i.e., set $\delta_r = 0$.

- Pitch attitude loop. Note that $e_{\theta_{\text{max}}}$ is a design parameter. Don’t use simplified simulink file. Command steps in pitch angle.

- Altitude using pitch, airspeed using pitch, and airspeed using throttle: implement directly on full Simulink model.

- Implement full autopilot using state machine for longitudinal control. Simulation should be from take-off to altitude hold.
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: \( E_K \triangleq \frac{1}{2} m V_a^2 \)
- 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:

- \( T^c = T_D + k_{p,t} \dot{E}_t + k_{i,t} \int_{t_0}^{t} \dot{E}_t \delta_\tau \)
  - \( T_D \) is thrust needed to counteract drag
  - PI controller based on total energy rate

- \( \theta^c = k_{p,\theta} \dot{E}_d + k_{i,\theta} \int_{t_0}^{t} \dot{E}_d \delta_\tau \)
  - 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 trust needed to counteract drag is unknown, then one possibility is to use an integrator to find $T_D$:

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

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

Nonlinear re-derivation:

- Error Definitions

\[ \tilde{E}_K = \frac{1}{2} m \left( (V_a^d)^2 - V_a^2 \right) \]
\[ \tilde{E}_P = mg (h^d - h) \]

- Lyapunov Function

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

- Controller

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

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: \[ T^c = D + \frac{\dot{E}^d_T}{V_a} + k_T \frac{\tilde{E}_T}{V_a} \]

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):

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

\[ k \in [0, 2] \]

- Nonlinear:

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

\[ k_1 \triangleq |k_T - k_D| \]

\[ k_2 \triangleq k_T + k_D \]

\[ 0 < k_T \leq k_D \]

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:

\[
T^c = \hat{D} + \Phi^\top \hat{\Psi} + \frac{\dot{E}_T^d}{V_a} + k_T \frac{\tilde{E}_T}{V_a}
\]

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

\[
\dot{\Psi} = \left( \Gamma_T \tilde{E}_T - \Gamma_D \tilde{E}_D \right) \Phi V_a
\]
Total Energy Control

Step in Altitude, Constant Airspeed

![Graph showing the response of different control systems to a step change in altitude with constant airspeed.](image)

- **Original TECS**
- **Nonlinear TECS**
- **Adaptive TECS**
- **SLC**
- **TECS PID**
- **ArduPilot PID**

Total Energy Control

Step in Airspeed, Constant Altitude

![Graph showing step in airspeed with constant altitude for different control systems, including Original TECS, Nonlinear TECS, Adaptive TECS, SLC, TECS PID, and ArduPilot PID.](image)

Total Energy Control

Step in Altitude and Airspeed

![Graph showing step in altitude and airspeed over time. The graph includes lines for different control systems: Original TECS, Nonlinear TECS, Adaptive TECS, SLC, TECS PID, and ArduPilot PID. The y-axes represent altitude (m) and airspeed (m/s) with time (s) on the x-axis.]

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 \( z_r \) and further suppose that \( z_r \) is a step, i.e., \( \dot{z}_r = 0 \). The first step is to augment the state with the integrator

\[ x_I = \int_{-\infty}^{t} (z(\tau) - z_r) \, d\tau. \]
LQR Control

Augment the States with an Integrator. (cont)

Defining the augmented state as $\xi = (x^T, x_I^T)^T$, 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

\[ J(x_0) = \min_{u(t), t \geq 0} \int_0^\infty x^\top(\tau) Q x(\tau) + u^\top(\tau) R u(\tau) d\tau. \]

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

$$Q = \begin{pmatrix}
  q_1 & 0 & \cdots & 0 \\
  0 & q_2 & \cdots & 0 \\
  \vdots & \vdots & \ddots & \vdots \\
  0 & 0 & \cdots & q_n
\end{pmatrix},$$

$$R = \begin{pmatrix}
  r_1 & 0 & \cdots & 0 \\
  0 & r_2 & \cdots & 0 \\
  \vdots & \vdots & \ddots & \vdots \\
  0 & 0 & \cdots & r_m
\end{pmatrix},$$

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 $\chi$ to commanded course $\chi_c$. 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

\[
\dot{\xi}_{lat} = \tilde{A}_{lat}\xi_{lat} + \tilde{B}_{lat}u_{lat},
\]

where

\[
\tilde{A}_{lat} = \begin{pmatrix} A_{lat} & 0 \\ H_{lat} & 0 \end{pmatrix}, \quad \tilde{B}_{lat} = \begin{pmatrix} B_{lat} \\ 0 \end{pmatrix}
\]

The LQR controller designed using

\[
Q = \text{diag} \left( [q_v, q_p, q_r, q_\phi, q_\chi, q_I] \right) \\
R = \text{diag} \left( [r_{\delta_a}, r_{\delta_r}] \right).
\]
LQR Control

Longitudinal Autopilot

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

\[ \dot{x}_{\text{lon}} = A_{\text{lon}} x_{\text{lon}} + B_{\text{lon}} u_{\text{lon}}, \]

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

The objective of the longitudinal autopilot is to drive altitude \( h \) to commanded altitude \( h_c \), and airspeed \( V_a \) to commanded airspeed \( V_{ac} \). Therefore, we augment the state with

\[
x_I = \begin{pmatrix} \int (h - h_c) dt \\ \int (V_a - V_{ac}) dt \end{pmatrix} = \int \left( H_{\text{lon}} x_{\text{lon}} - \begin{pmatrix} h_c \\ V_{ac} \end{pmatrix} \right) dt,
\]

where

\[
H_{\text{lon}} = \begin{pmatrix} 0 & 0 & 0 & 0 & 1 \\ \frac{1}{V_a} & \frac{1}{V_a} & 0 & 0 & 0 \end{pmatrix}.
\]
LQR Control

Longitudinal Autopilot (cont)

The augmented longitudinal state equations are therefore

\[
\dot{\xi}_{\text{lon}} = \bar{A}_{\text{lon}} \xi_{\text{lon}} + \bar{B}_{\text{lon}} u_{\text{lon}},
\]

where

\[
\bar{A}_{\text{lon}} = \begin{pmatrix} A_{\text{lon}} & 0 \\ H_{\text{lon}} & 0 \end{pmatrix}, \quad \bar{B}_{\text{lon}} = \begin{pmatrix} B_{\text{lon}} \\ 0 \end{pmatrix}
\]

The LQR controller designed using

\[
Q = \text{diag} ( [q_u, q_w, q_q, q_\theta, q_h, q_I])
\]

\[
R = \text{diag} ( [r_{\delta_e}, r_{\delta_t}])
\]