Final Quadsim Project

525.661 Unmanned Aerial Vehicles System and Controls

Alec Portelli

1)

Fixed-wing and multi-rotor UAVs share similarities in experiencing forces and moments that affect their flight dynamics. Both feel lift, opposing gravity, and drag. However, the force and moment generation with aerodynamics are a lot different. Fixed-wing UAVs rely on aerodynamic lift produced by the shape of their wings, which creates differential air pressure, while multi-rotor UAVs generate lift through the vertical thrust of multiple rotors. Control over pitch, roll, and yaw for fixed-wing UAVs is achieved via aerodynamic control surfaces like elevators and ailerons, whereas multi-rotor UAVs adjust rotor speeds to induce these moments. In terms of aerodynamics, fixed-wing UAVs experience drag from air resistance, whereas multi-rotor UAVs generate lift purely through rotor thrust induces drag primarily from structural resistance.

2)

2)

$$F_{p_i} = (\rho \cdot C_{prop} \cdot S_{prop} \cdot k^2_{prop})\, \delta^2_{p_i}$$

Total gravity $= m \cdot g$

$$m \cdot g = \sum_{i=1}^{4} F_{p_i} = \sum_{i=1}^{4} (\rho \cdot C_{prop} \cdot S_{prop} \cdot k^2_{prop})\, \delta^2_{p_i}$$

Assuming $\delta_{p_i} = \delta_{p,2} = \delta_{p,3} = \delta_{p,4} = \delta^*_t$

We need to solve for throttle:

$$m \cdot g = 4 \cdot (\rho \cdot C_{prop} \cdot S_{prop} \cdot k^2_{prop}) \cdot \delta^2_{p_i}$$

$$m \cdot g / (4 \cdot \rho \cdot C_{prop} \cdot S_{prop} \cdot k^2_{prop}) = \delta^2_{p_i}$$

$$\sqrt{m \cdot g / (4 \cdot \rho \cdot C_{prop} \cdot S_{prop} \cdot k^2_{prop})} = \delta_{p_i}$$

$m = 1.023$ $\quad C_{prop} = 1$

$g = 9.8067$ $\quad S_{prop} = 0.0387$

$\rho = 1.2682$ $\quad k_{prop} = 17.4556$

$$\sqrt{(1.023) \cdot (9.8067) / (4 \cdot 1.2682 \cdot 1 \cdot 0.0387 \cdot 17.4556^2)}$$

$$\delta_{p_i} = \sqrt{10.0322541 / (51.817750)}$$

$$\boxed{\delta_{p_i} = 0.409529}$$

3)

A: Derivation Of Throttle To Altitude

Derivation of throttle to altitude transfer function

We have the equation $F_{net} = \sum F_{p_i} - mg$

We can rewrite this into: $m \cdot \ddot{h} = 4 \cdot k_{Fp} k^2 \Omega_p \cdot \delta_t^2 - mg$

Divide by m: $\ddot{h} = \frac{4}{m} \cdot k_{Fp} \cdot k^2 \Omega_p \cdot \delta_t^2 - g$

Since we are deriving from trim, we can split $\ddot{h}$ into $\ddot{h}^*$ and $\bar{\ddot{h}}$

Rewrite as $\ddot{h}^* + \bar{\ddot{h}} = k_{Fp} \cdot k^2 \Omega_p \cdot \delta_t^2 - g \cdot \frac{4}{m}$

$\delta_t = \delta_t^* + \bar{\delta}_t$, which we can substitute into the equation

$\ddot{h}^* + \bar{\ddot{h}} = k_{Fp} \cdot k^2 \Omega_p \cdot (\delta_t^* + \bar{\delta}_t)^2 - g \cdot \frac{4}{m}$

Expand: $(\delta_t^* + \bar{\delta}_t)(\delta_t^* + \bar{\delta}_t) = (\delta_t^{*2} + 2\delta_t^* \bar{\delta}_t + \bar{\delta}_t^2) \cdot \frac{4}{m}$

$[k_{Fp} \cdot k^2 \Omega_p][\delta_t^{*2} + 2\delta_t^* \bar{\delta}_t + \bar{\delta}_t^2)$ can be rewritten as:

$\frac{4}{m}[k_{Fp} \cdot k^2 \Omega_p][\delta_t^{*2}] + \frac{4}{m}[k_{Fp} \cdot k^2 \Omega \cdot 2\delta_t^* \bar{\delta}_t] + \frac{4}{m}[k_{Fp} k^2 \Omega_p \cdot \bar{\delta}_t^2] - g$

$(\frac{4}{m} k_{Fp} k^2 \Omega_p \delta_t^{*2}) + (\frac{4}{m} k_{Fp} \cdot k^2 \Omega \cdot 2 \cdot \delta_t^* \bar{\delta}_t) + (\frac{4}{m} k_{Fp} \cdot k^2 \Omega_p \cdot \bar{\delta}_t^2) - g$

$\frac{4}{m} k_{Fp} k^2 \Omega_p \delta_t^{*2} + \frac{8}{m} k_{Fp} \cdot k^2 \Omega_p \cdot \delta_t^* \cdot \bar{\delta}_t + \frac{4}{m} k_{Fp} \cdot k^2 \Omega_p \cdot \bar{\delta}_t^2 - g$

Since we are at trim, $\ddot{h}^*$ goes to ϕ

$\frac{4}{m} k_{Fp} k^2 \Omega_p \delta_t^{*2} = g$ and $\bar{\delta}_t^2 \cdot \frac{4}{m} k_{Fp} \cdot k^2 \Omega_p \approx 0$, so simplifies to:

$\bar{\ddot{h}} = \frac{8}{m} k_{Fp} \cdot k^2 \Omega_p \cdot \delta_t^* \cdot \bar{\delta}_t$ = Linearized equation

We can take the linear itself equation and take the Laplace to find the transfer function

$$G_{dt2h} = \frac{\bar{h}(s)}{\delta_t(s)} = \frac{\frac{g}{m} \cdot K_{FP} \cdot K_{SP}^2 \cdot \delta_t^*}{s^2}$$

$$K_{FP} = \frac{1.26 \cdot 1 - (0.37707) \cdot 17.26^2}{946.1011^2} = 0.00001575$$

$$\frac{g}{m} = 7.8201 \qquad K_{SP} = 946.1011$$

$$\delta_t^* = .4095$$

$$7.8201 - 0.00001575 \cdot 946.1011^2 \cdot 0.4095 = \boxed{47.8984 \text{ rad/s}^2}$$

Derivation Of Elevator To Pitch

Derivation of elevator to pitch transfer function

$$M_y = \Delta x \left[(F_{p1} + F_{p3}) - (F_{p2} + F_{p4}) \right]$$

We know: $F_{pi} = K_{Fp} \Omega_{pi}^2$ So we can rewrite M_y as:

$$J_y \ddot{\theta} = \Delta x \cdot K_{Fp} \cdot k^2 \Omega_p \left[(\delta_{p1}^2 + \delta_{p3}^2) - (\delta_{p2}^2 + \delta_{p4}^2) \right]$$

We know to deviate from level trim is:

$$\left.\begin{aligned} \delta_{p1} = \delta_{p3} &= \delta_t^* + \delta_e^* \\ \delta_{p2} = \delta_{p4} &= \delta_t^* - \delta_e^* \end{aligned}\right] \rightarrow \underline{\text{No}} \text{ star}$$

Trim in this case is 0

We can write that $\ddot{\theta} = \ddot{\theta}^* + \bar{\ddot{\theta}}$

The equation is now $J_y \cdot [\ddot{\theta}^* + \bar{\ddot{\theta}}] =$ everything before

We can substitute for the throttles:

$$J_y \cdot [\ddot{\theta}^* + \bar{\ddot{\theta}}] = \Delta x \cdot K_{Fp} \cdot k^2 \Omega_p \cdot \left[\left[(\delta_t^* + \delta_e)^2 + (\delta_t^* + \delta_e)^2 \right] - \left[(\delta_t^* - \delta_e)^2 + (\delta_t^* - \delta_e)^2 \right] \right]$$

Expanding gives us:

$$(\delta_t^* + \delta_e)^2 = (\delta_t^* + \delta_e)(\delta_t^* + \delta_e) = \left[\delta_t^{*2} + 2\delta_e \delta_t^* + \delta_e^2 \right] \cdot 2$$

$$(\delta_t^* - \delta_e)^2 = (\delta_t^* - \delta_e)(\delta_t^* - \delta_e) = 2\left[\delta_t^{*2} - 2\delta_e \delta_t^* + \delta_e^2 \right]$$

$\rightarrow$

Factor out the 2

$$2\left[\left(\delta_t^{*2} + 2\delta_e\delta_t^* + \delta_e^2\right) - \left(\delta_t^{*2} - 2\delta_e\delta_t^* + \delta_e^2\right)\right]$$

$$2\left[\left(\delta_t^{*2} + 2\delta_e\delta_t^* + \delta_e^2 - \delta_t^* + 2\delta_e\delta_t^* - \delta_e^2\right)\right]$$

$$2\left[\left(\delta_t^{*2} + 4\delta_e\delta_t^* - \delta_t^{*2}\right)\right]$$

$$2\left[4\delta_e\delta_t^*\right] = 8\delta_e\delta_t^*$$

Linearized equation becomes $J_y \cdot [\ddot{\theta}^* + \ddot{\bar{\theta}}] = \Delta x \cdot k_{pF} \cdot k^2 \Omega p \cdot 8\delta_e\delta_t^*$

Since we are at trim, $\ddot{\theta}^*$ is 0

$$J_y \cdot \ddot{\bar{\theta}} = \Delta x \cdot k_{pF} \cdot k^2 \Omega p \cdot 8\delta_e\delta_t^*$$

$$\boxed{\ddot{\bar{\theta}} = \frac{\Delta x \cdot k_{pF} \cdot k^2 \Omega p \cdot 8\delta_e\delta_t^*}{J_y}}$$ → Linearized equation

Transfer function: $G_{d_{e2pitch}} = \frac{8 \cdot \Delta x}{J_y} \cdot \frac{k^2 \Omega p \cdot \delta_t^* \cdot k_{Fp}}{s^2}$

$$\frac{8 \cdot (0.1579)}{0.0095} = 13.2968$$

$(0.4095) = \delta_t^*$ $\quad k_{Fp} = \frac{1.26 \cdot 1 \cdot 0.38707 \cdot 17^2}{946^2} = 0.00001575$

$$\frac{946^2 \cdot (0.4095) \cdot (0.00001575) \cdot 8 \cdot (0.1579)}{0.0095} = \frac{7.71}{0.0095} = 810.5 \text{ rad/s}^2$$

Derivation Of Aileron To Roll

Derivation of elevator to roll

$M_x = \Delta y\left[(F_1 + F_3) - (F_2 + F_4)\right]$ which can be rewritten as

$J_x \ddot{\phi} = \Delta y \cdot K_{FP} \cdot k^2 \Omega_P \left[(\delta_{P1}^2 + \delta_{P3}^2) - (\delta_2^2 + \delta_{P4}^2)\right]$

Deflections from trim are: $\delta_1 = \delta_3 = \delta_t^* + \delta_a$

$\delta_2 = \delta_4 = \delta_t^* - \delta_a$

Based off the derivation from elevator to pitch, it is the same linearized equation, but swap J_x and J_y, Δx with Δy

$J_x\left[\ddot{\phi}^* + \ddot{\phi}\right] = \Delta y \cdot K_{PF} \cdot k^2 \Omega_P \cdot \gamma\, \delta_a\, \delta_t^*$

Since we are deviating from trim, $\ddot{\phi}^* = 0$, which simplifies to

$J_x\left[\ddot{\phi}\right] = \Delta y \cdot K_{PF} \cdot k^2 \Omega_P \cdot \gamma\, \delta_a\, \delta_t^*$

Divide by J_x:

$\ddot{\phi} = \dfrac{\Delta y \cdot K_{PF} \cdot k^2 \Omega_P \cdot \gamma\, \delta_a \cdot \delta_t^*}{J_x}$ → Linearized function

The Laplace form for the transfer function would be:

$G_{da\,roll}\ \dfrac{\phi(s)}{\delta_a(s)} = \dfrac{\Delta y \cdot K_{PF} \cdot k^2 \Omega_P \cdot \gamma \cdot \delta_t^*}{J_x}$

plugging in values yields: $\dfrac{(.1579)(0.00001575)\cdot(996.1011)^2 \cdot \gamma \cdot (.4095)}{.0093} =$ $\boxed{810\ \text{rad/s}^2}$

Derivation Of Rudder To Yaw

Derivation of rudder to yaw transfer function

$$M_z = [(T_{p_1} + T_{p2}) - (T_{p3} + T_{p4})]$$

We can rewrite this as: $J_z \ddot{\psi} = k_{Tp} \cdot k^2_{\Omega p} [(\delta_{p_1}^2 + \delta_{p_2}^2) - (\delta_{p_3} + \delta_{p_4})]$

To derivate from trim we can assume that:

$$\delta_{p_1} = \delta_{p2} = \delta_t^* + \delta_r$$

$$\delta_{p_3} = \delta_{p_4} = \delta_t^* - \delta_r$$ And the trim is zero

We can rewrite that $\ddot{\psi}^* + \bar{\ddot{\psi}} = \ddot{\psi}$, so the new equation becomes

$$J_z [\ddot{\psi}^* + \bar{\ddot{\psi}}] = [(\delta_{p_1}^2 + \delta_{p2}^2) - (\delta_{p_3}^2 + \delta_{p_4}^2)] \cdot k_{Tp} \cdot k^2_{\Omega p}$$

We can substitute for the throttles:

$$[(\delta_t^* + \delta_r)^2 + (\delta_t^* + \delta_r)^2] = (\delta_t^* + \delta_r)(\delta_t^* + \delta_r) \rightarrow$$

$$2 \cdot (\delta^{*2} + 2\delta_r \delta_t^* + \delta_r^2)$$

$$2 \cdot (\delta_t^* - \delta_r)^2 = (\delta_t^* - \delta_r)(\delta_t^* - \delta_r) = \delta_t^{*2} - 2\delta_r \delta_t^* + \delta_r^2$$

$$2(\delta_t^{*2} - 2\delta_r \delta_t^* + \delta_r^2)$$

Factor the 2: $2[\delta_t^{*2} + 2\delta_r \delta_t^* + \delta_r^2 - (\delta_t^{*2} - 2\delta_r \delta_t^* + \delta_r^2)]$

$$2[\delta_t^{*2} + 2\delta_r \delta_t^* + \delta_r^2 - \delta_t^{*2} + 2\delta_r \delta_t^* - \delta_r^2]$$

Simplify: $2[4\delta_r \delta_t^*] \rightarrow 8\delta_r \delta_t^*$

$\rightarrow$

Linearized equation becomes $J_z[\bar{\ddot{\psi}} \cdot \ddot{\psi}^*] = 8\,\delta_r\,\delta_t^* \cdot K_{Tp} \cdot k^2 \Omega_p$

Since we are at trim, $\ddot{\psi}^* = 0$ and we can divide by J_z

$$\boxed{\bar{\ddot{\psi}} = \frac{8 \cdot \delta_r \cdot \delta_t^* \cdot K_{Tp} \cdot k^2 \Omega_p}{J_z}}$$ → Linearized equation

Transfer function: $G_{\delta_r 2 yaw} = \dfrac{8 \cdot \delta_t^* \cdot K_{Tp} \cdot k^2 \Omega_p}{J_z \cdot s^2}$

plugging in values:

$\delta_t^* = 0.4095 \quad J_z = 0.0186$

$K_{\Omega p} = 946.1011$

$K_{Tp} = 2.6673e^{-7}$

$$\frac{8 \cdot (0.4095) \cdot (2.6673e^{-7}) \cdot (946.1011)^2}{0.0186} = 42.0512 \text{ rad}/s^2$$

4)

Fixed-wing and quadcopter autopilot control structures share similarities in their overall function of achieving desired flight conditions. However, they differ in their approach due to the difference in flight dynamics of fixed-wing aircraft and quadcopters. In the inner loop, fixed-wing autopilots focus on attitude and airspeed control, which use PID controllers to adjust control surfaces and throttle for pitch, roll, and airspeed. On the other hand, quadcopter inner loops prioritize orientation and angular rate control, using PID controllers to manage rotor speeds for stable roll, pitch, and yaw. In the outer loops, both systems use different approaches for altitude and navigation control. Fixed-wing autopilots adjust pitch and throttle for to maintain altitude and use waypoint following algorithms for navigation. Quadcopters use autopilots to control rotor thrust for altitude and align with waypoint coordinates using GPS and navigation algorithms.

5)

Fixed-wing aircraft and quadcopters rely on different sensors and state estimation techniques because of their unique flight dynamics. Both use inertial measurement units (IMUs) which have accelerometers and gyroscopes for attitude stabilization and navigation. Both vehicles also use GPS and receivers are commonly used for precise position and velocity data. However, fixed-wing aircraft typically incorporate airspeed sensors such as Pitot tubes for measuring airflow. Airflow measurements are key for managing lift which is generated by the control surfaces. However, quadctopers, which generate lift only through rotor thrust, do not usually require airspeed sensors due to their different flight dynamics. The quadcopter isn't using control surfaces like a fixed wing, but generates lift through the motors instead. While both types of vehicles make use of altitude sensors, fixed-wing aircraft may utilize barometric altimeters, whereas quadcopters may use barometers or ultrasonic sensors. Implementing a Pitot sensor on a quadcopter is not practical due to the lack of control surfaces. Quadcopters do not wings and rely entirely on rotor thrust, making the measurement of airspeed less important and adding complexity and cost to the system.

6)

For all info, reference the question_6.m. Resulting matrices:

```
A_lon =

   -0.5083         0         0   -9.8066         0
         0    0.2484         0         0         0
         0         0         0         0         0
         0         0    1.0000         0         0
         0    1.0000         0         0         0
```

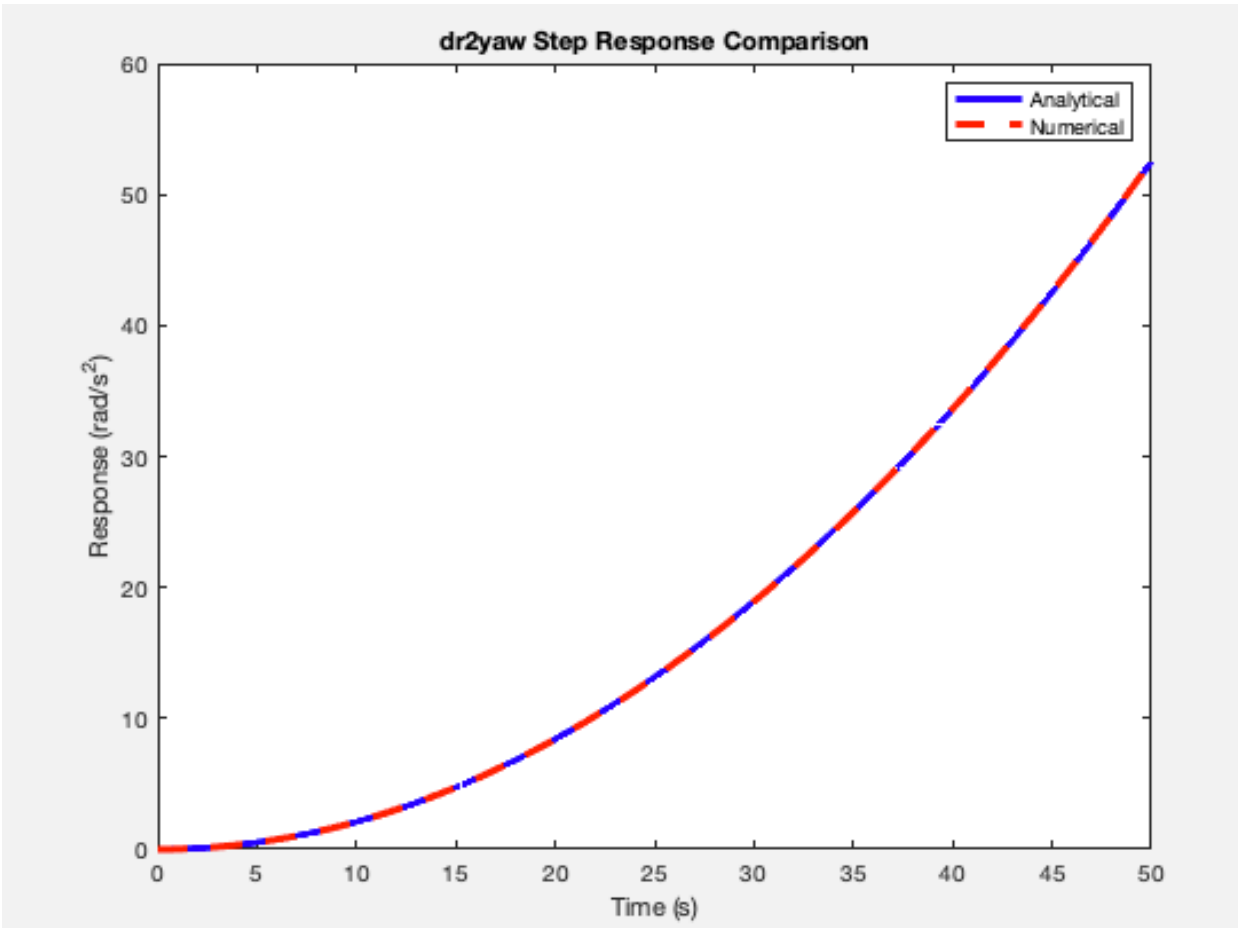
```
B_lon =

         0         0
         0  -47.8971
  810.7474         0
         0         0
         0         0
```

```
A_lat =

   -0.5083         0         0    9.8066         0
         0         0         0         0         0
         0         0         0         0         0
         0    1.0000         0         0         0
         0         0    1.0000         0         0
```

```
B_lat =

         0         0
  810.7474         0
         0   42.0500
         0         0
         0         0
```

As seen in the lat/lon matrices, the value 810.7474 appears in both B matrices. This is equivalent to the values we found for both the aileron to roll and elevator to pitch derivations. We can also see the 42.05 which is equal to the rudder to yaw transfer function, and for throttle to altitude the 47.8971 but here its negative because of the way altitude is described.

Rudder To Yaw

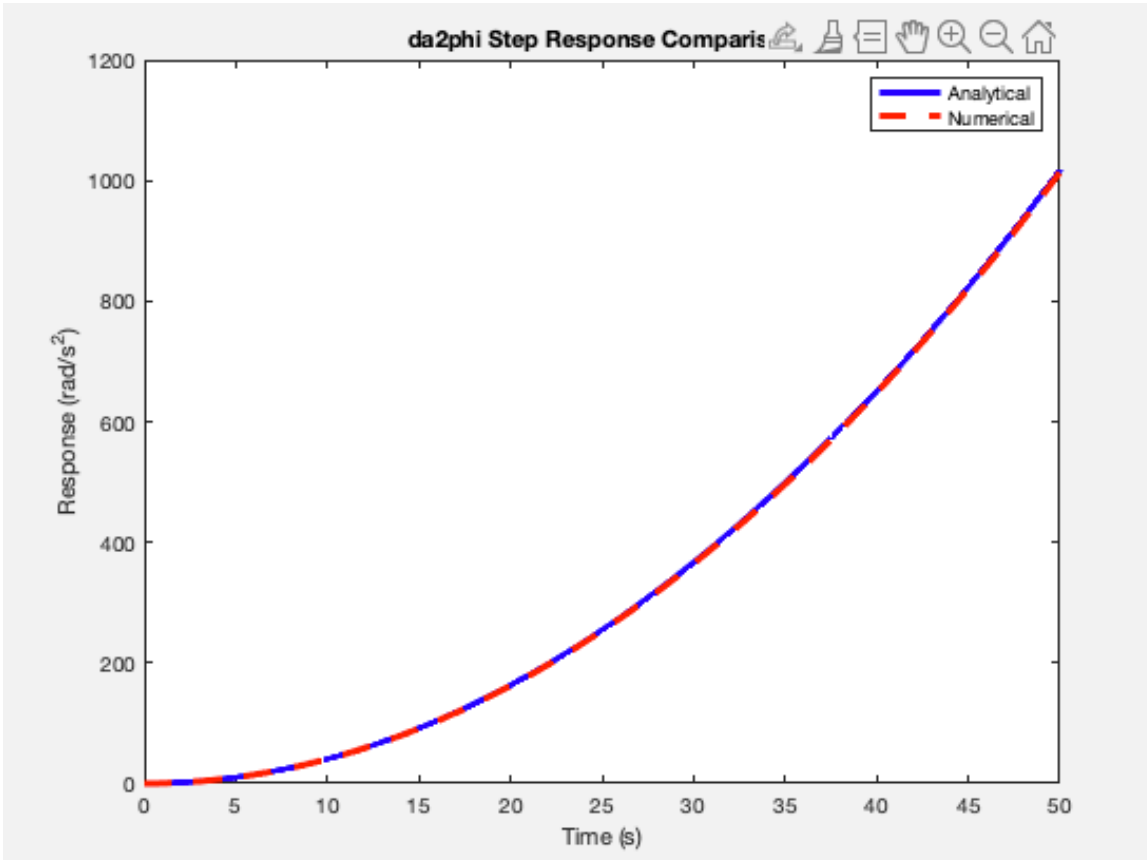


Resulting Transfer Function

```
dr2yaw_numerical =

  42.05
  -----
   s^2
```

Analytical and numerical match up perfectly.

Aileron To Roll

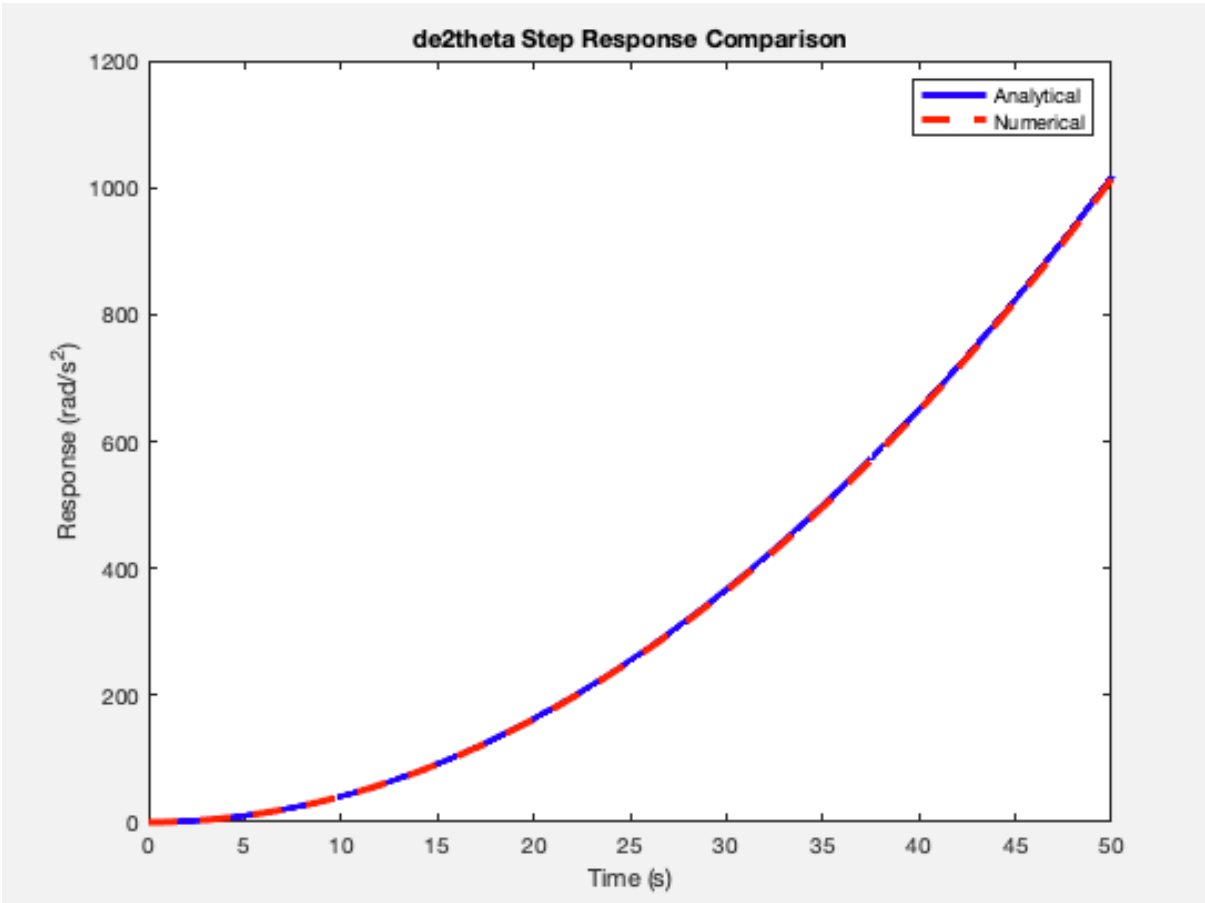


Resulting Transfer Function

```
da2phi_numerical =

  810.75
  ------
   s^2
```

Analytical and numerical match up perfectly.

Elevator to Pitch

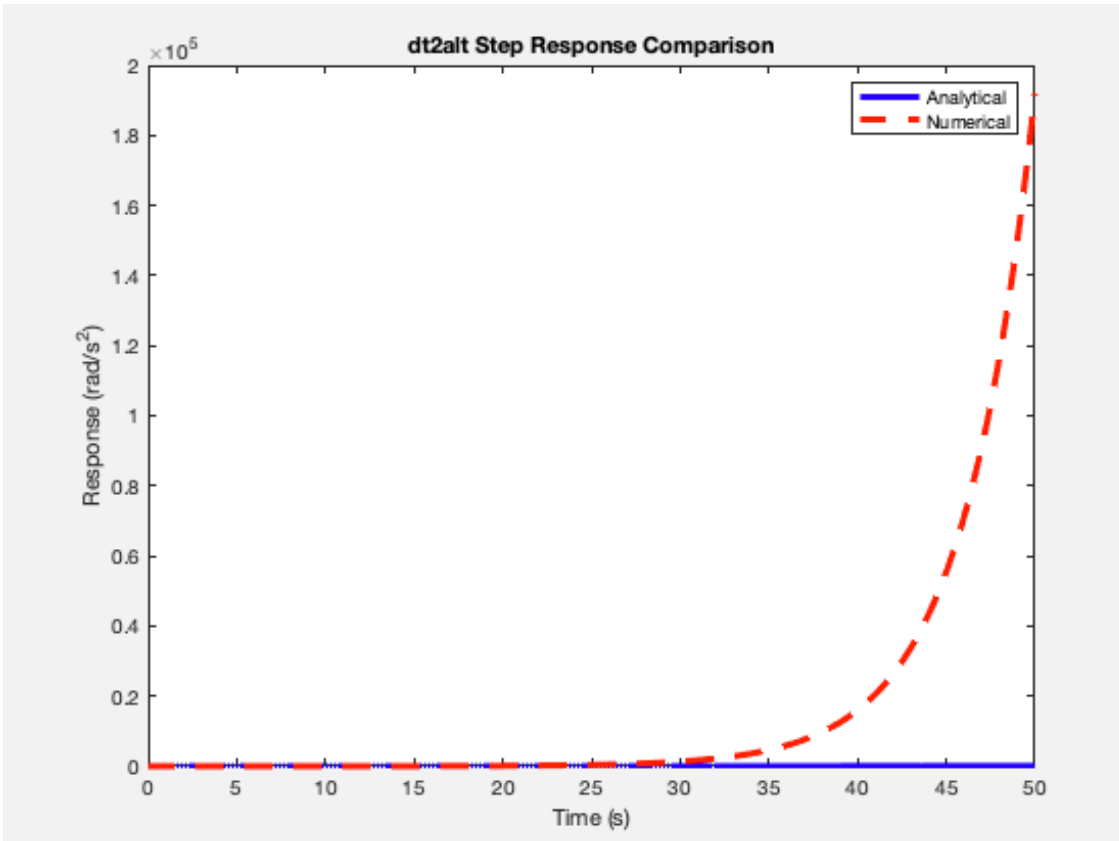


Resulting Transfer Function

```
de2theta_numerical =

  810.75
  ------
   s^2
```

Analytical and numerical match up perfectly.

Throttle to Altitude



Resulting Transfer Function

```
dt2alt_numerical =

     47.897
  ------------
  s (s-0.2484)
```

Analytical and numerical match up perfectly for a bit. However, it was mentioned in the notes that there is an unstable pole for the numerical derivation which we can clearly see towards the end.

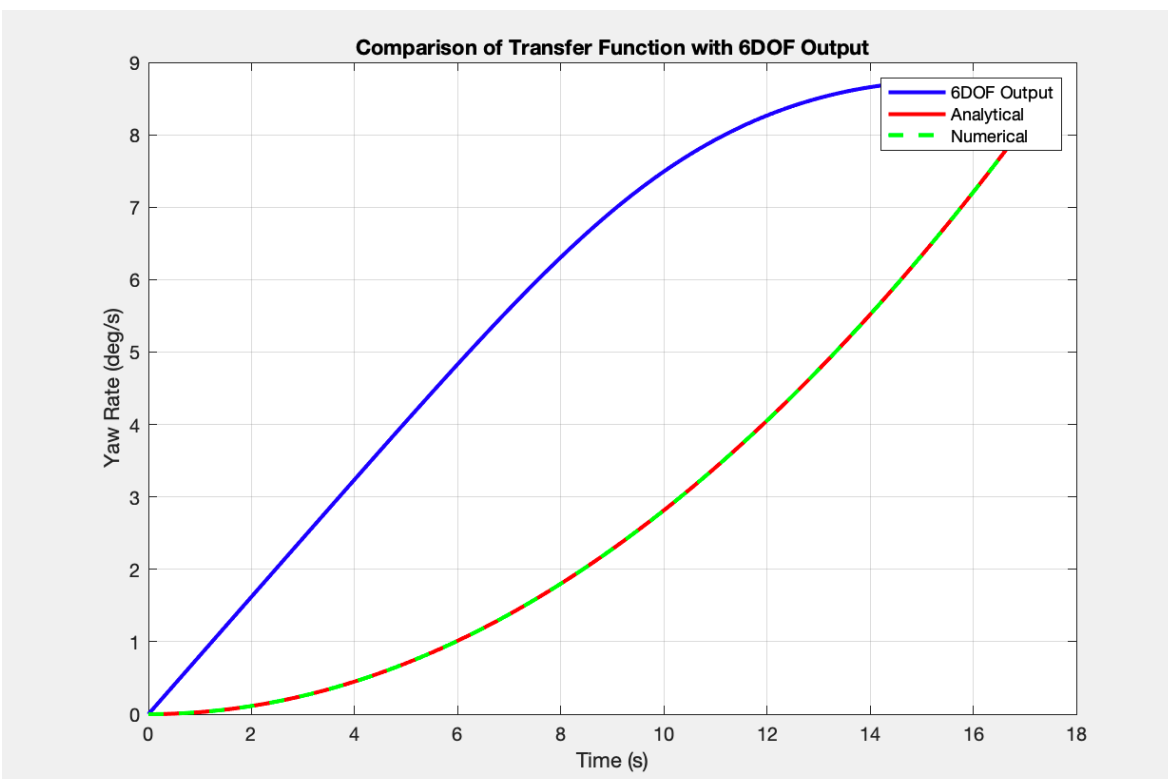
Question 7)

For each control surface, I added a deviation of 0.001 * (PI/180) in the quadsim_controls.m file. Each deviation was added independently, and no more than 1 control surface had a deviation during a simulation.

Transfer Functions

```
%% Transfer functions ----------------------------------------------------

s = tf('s');
% dr2yaw
dr2yaw_numerical = H(kpsi,kdr);
dr2yaw_analytical = (8 * P.delta_t0 * P.k_Tp * P.k_omega^2) / (P.Jz * s^2);

% da2phi
da2phi_numerical = H(kphi, kda);
da2phi_analytical = (8 * P.delta_t0 * P.delta_y * P.k_Pf * P.k_omega * P.k_omega) ...
     / (P.Jx * s^2);

% de2theta
de2theta_numerical = H(ktheta,kde);
de2theta_analytical = (8 * P.delta_x * P.k_Pf * P.k_omega * P.k_omega * P.delta_t0) ...
     / (P.Jy * s^2);

% dt2alt
dt2alt_numerical = -1*H(kpd,kdt);
dt2alt_analytical = ((8/P.mass) * P.k_Pf * P.k_omega * P.k_omega ...
    * P.delta_t0) / (s^2);
```

de2theta:

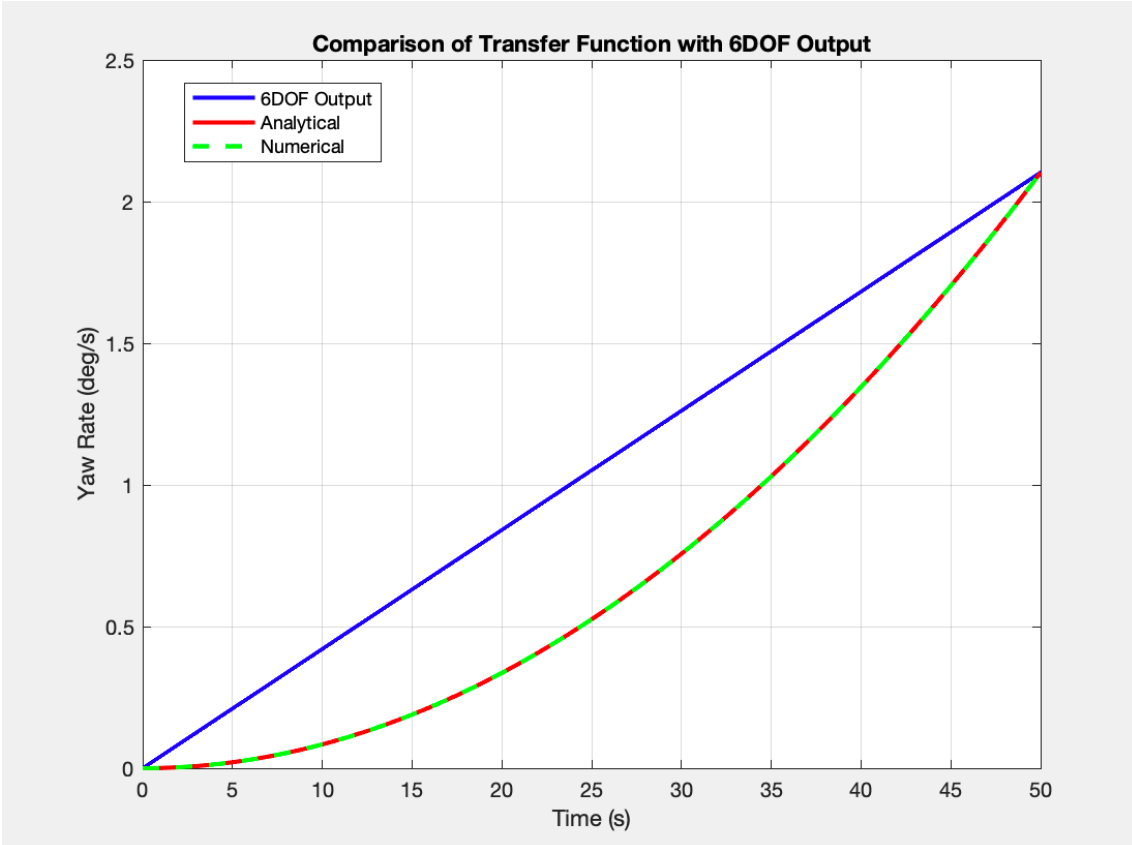


Plotting function:

```
s = tf('s');
de2theta_analytical = (8 * P.delta_x * P.k_Pf * P.k_omega * P.k_omega * P.delta_t0) / (P.Jy * s^2);
t_end = max(out.time_s); % End time for the simulation
[y_tf, t_tf] = step(de2theta_analytical, t_end);

% Step 2: Scale and shift the transfer function response
scaling_factor = max(abs(out.q_dps)) / max(abs(y_tf)); % Scaling factor
shift_factor = 0; % Shift factor (adjust as needed)
y_tf_scaled = scaling_factor * y_tf + shift_factor; % Scaled and shifted transfer function response
```

In this plot, we observe that the simulation rate is in a similar ballpark to the numerical/analytical transfer functions. While the transfer functions are increasing in rate, the rate of the simulation begins to plateau a bit. At approximately 17 seconds the quadcopter crashes, but we see the exact same rate at that time from the transfer functions and the simulation, demonstrating a the synchronization between the two.

dr2yaw

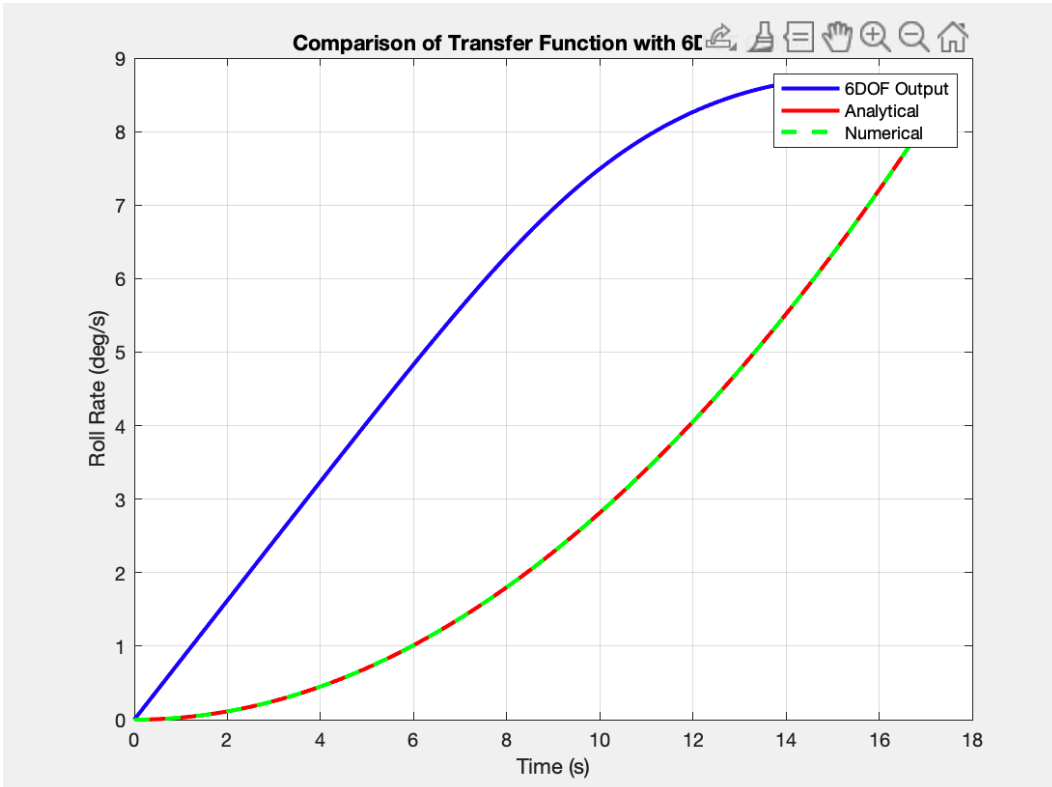


Plotting Function

```
s = tf('s');
dr2yaw_analytical = (8 * P.delta_t0 * P.k_Tp * P.k_omega^2) / (P.Jz * s^2);
t_end = max(out.time_s); % End time for the simulation
[y_tf, t_tf] = step(dr2yaw_analytical, t_end);

% Step 2: Scale and shift the transfer function response
scaling_factor = max(abs(out.r_dps)) / max(abs(y_tf)); % Scaling factor
shift_factor = 0; % Shift factor (adjust as needed)
y_tf_scaled = scaling_factor * y_tf + shift_factor; % Scaled and shifted transfer function response
```

In this plot, we see a linear rate with the yaw from the simulation model, unlike the prior plot. However, we see a similar behavior here from the transfer functions where they converge to the correct final yaw rate at the end of the simulation. The transfer functions is never more than a degree off from the actual simulation rare, which implies a realistic response. Overall, a good performance from the transfer functions.

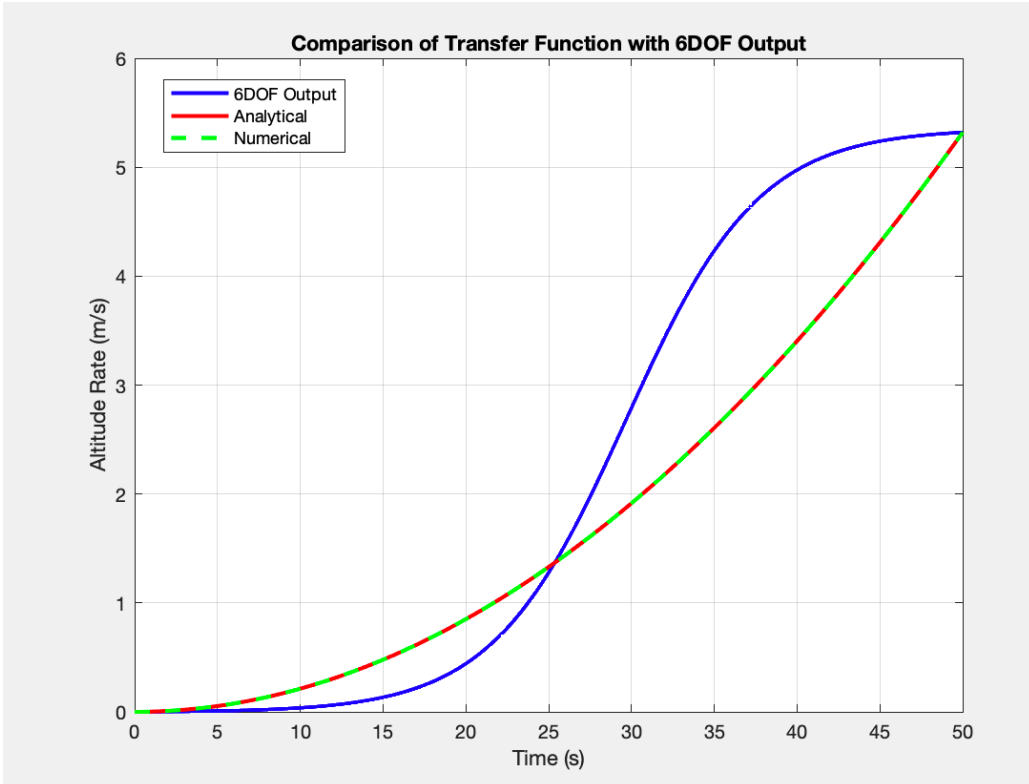
da2phi




```
% Step 1: Simulate the step response of the transfer function
s = tf('s');
da2phi_analytical = (8 * P.delta_t0 * P.delta_y * P.k_Pf * P.k_omega * P.k_omega) ...
     / (P.Jx * s^2);
t_end = max(out.time_s); % End time for the simulation
[y_tf, t_tf] = step(da2phi_analytical, t_end);

% Step 2: Scale and shift the transfer function response
scaling_factor = max(abs(out.p_dps)) / max(abs(y_tf)); % Scaling factor
shift_factor = 0; % Shift factor (adjust as needed)
y_tf_scaled = scaling_factor * y_tf + shift_factor; % Scaled and shifted transfer function response
```

Another similar output to the other previous plots. We can see the converge right before the quadcopter crashes at around 17 seconds. The roll rate was linear until it began to decrease as the quadcopter began to flip out of control and was not able to continue to increase the rate of roll. We observe that the transfer function was not as accurate for roll than it was for yaw, but still was able to approximate the rate from the simulation.

dt2alt



Plotting Function:

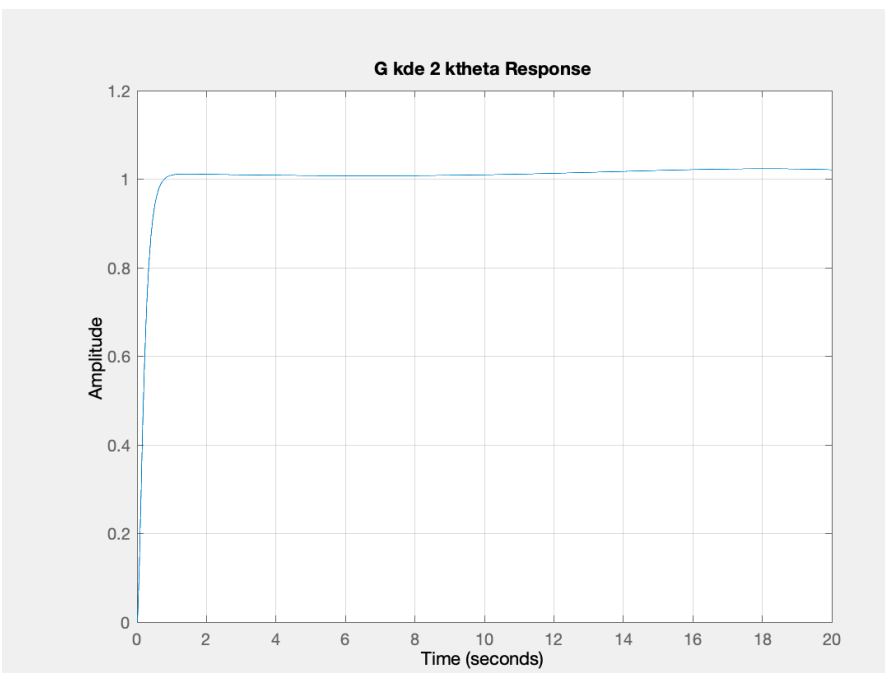
```
s = tf('s');
dt2alt_analytical = ((8/P.mass) * P.k_Pf * P.k_omega * P.k_omega ...
    * P.delta_t0) / (s^2);
t_end = max(out.time_s); % End time for the simulation
[y_tf, t_tf] = step(dt2alt_analytical, t_end);

% Step 2: Scale and shift the transfer function response
scaling_factor = max(abs(out.alt_rate_mps)) / max(abs(y_tf)); % Scaling fac
shift_factor = 0; % Shift factor (adjust as needed)
y_tf_scaled = scaling_factor * y_tf + shift_factor; % Scaled and shifted tr
```

This plot is a little more interesting than before. The altitude rate of the quadcopter ramps up quickly and then plateaus. The transfer function once again converges at the end of the simulation to where the quadcopter altitude rate finishes. The transfer function does not match the altitude rate exactly but it follows a similar trend to the simulation output.

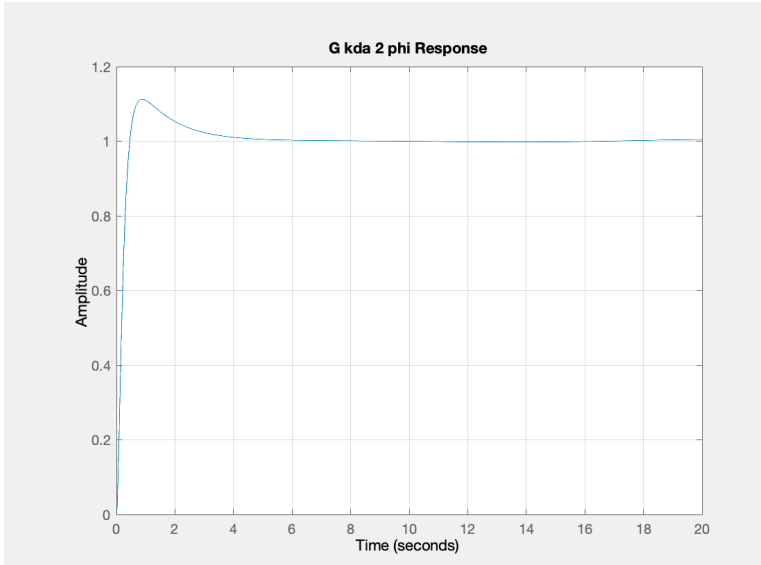
8)

In the interest of time, I manually tuned the PID controller gains instead of using advanced methods. I tuned the PID gains by iteratively adjusting the P, I, and D based on observations of the system's response. Starting with the proportional gain, I gradually increased it until the system exhibited oscillations, determining the critical gain and oscillation period. Next, I fine-tuned the integral gain to reduce steady-state error without causing significant oscillations or instability. Finally, I adjusted the derivative gain to dampen oscillations and enhance system stability. This process continued until the controller's performance aligned with desired criteria such as stability, response time, and steady-state error. While manual tuning was effective for this project, more complex systems or precise control requirements may benefit from advanced tuning methods or optimization algorithms.

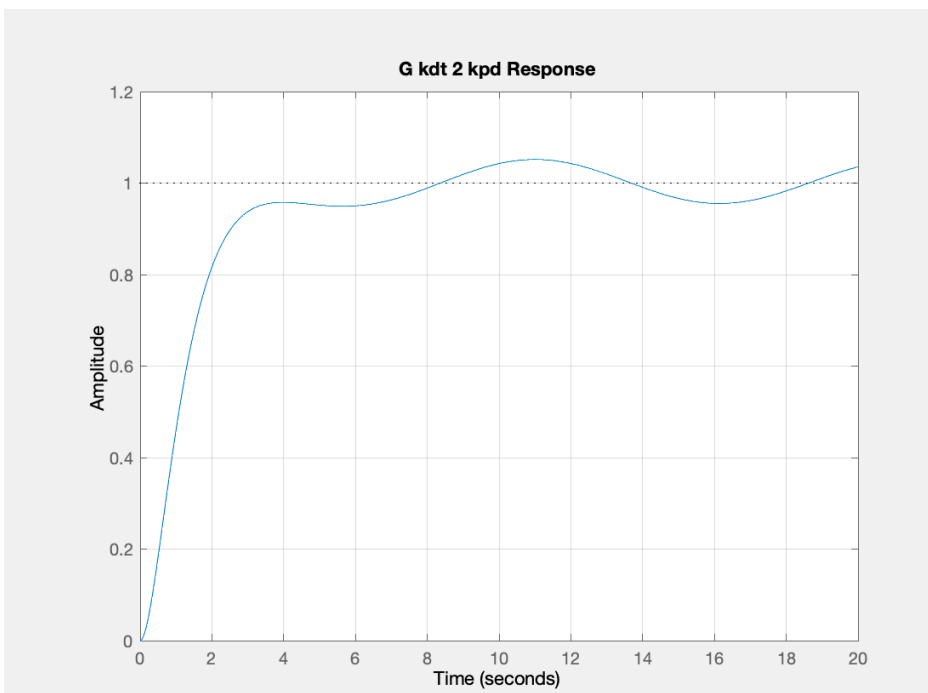
Before deploying to the simulation, I made sure to try the controller by itself and plot the results. Once I was happy with the response, I then used it on the quadcopter. Here are the first round results before using the controller on the quadcopter:



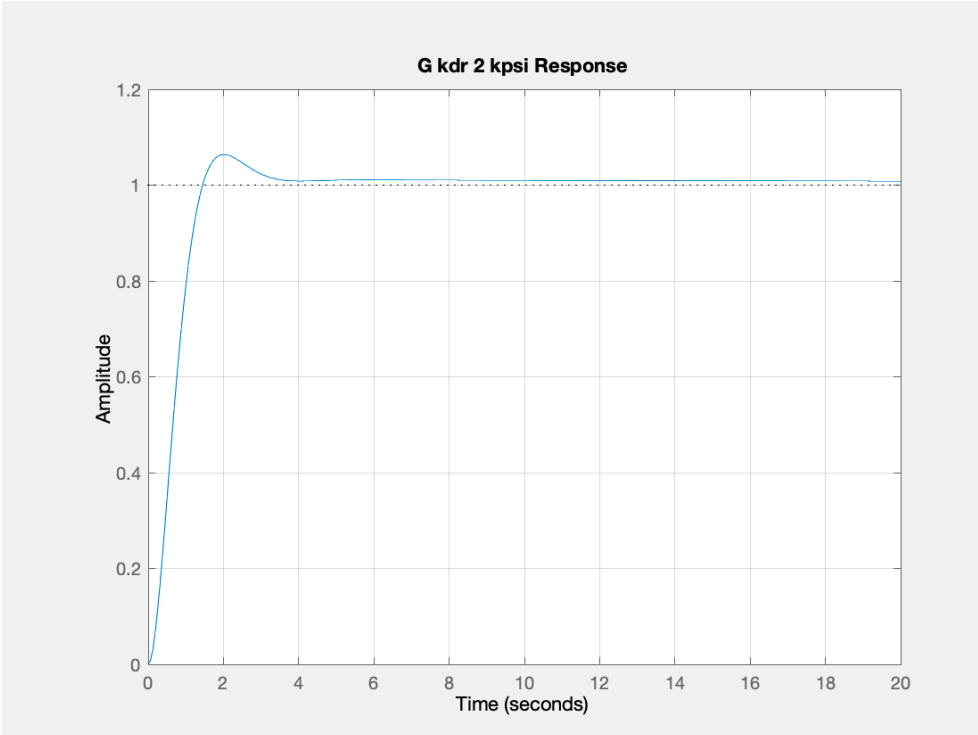
Gains: `(H(ktheta,kde), 0.11, 0.0075, 0.025);`



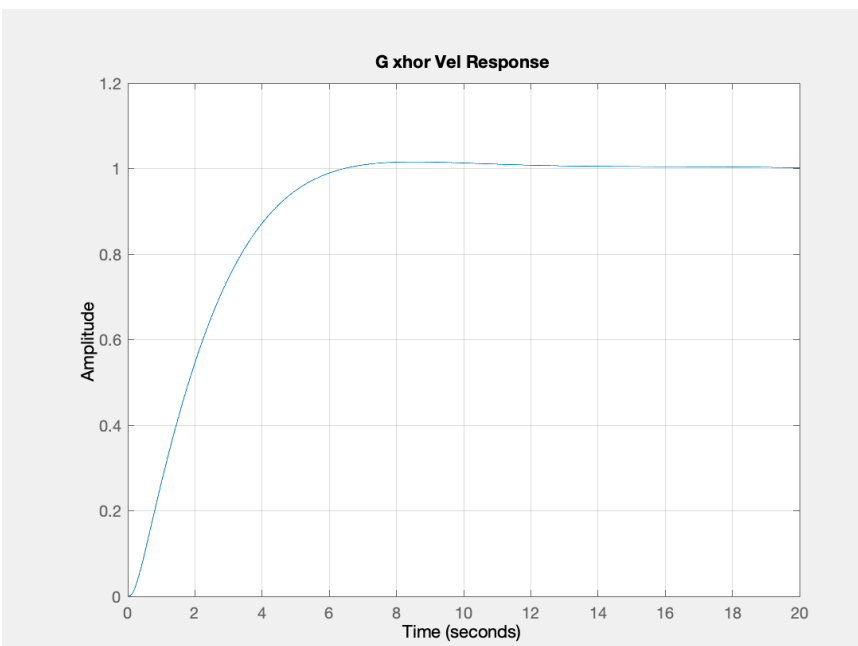
Gains: `(H(kphi,kda), 0.11, 0.075, 0.025);`



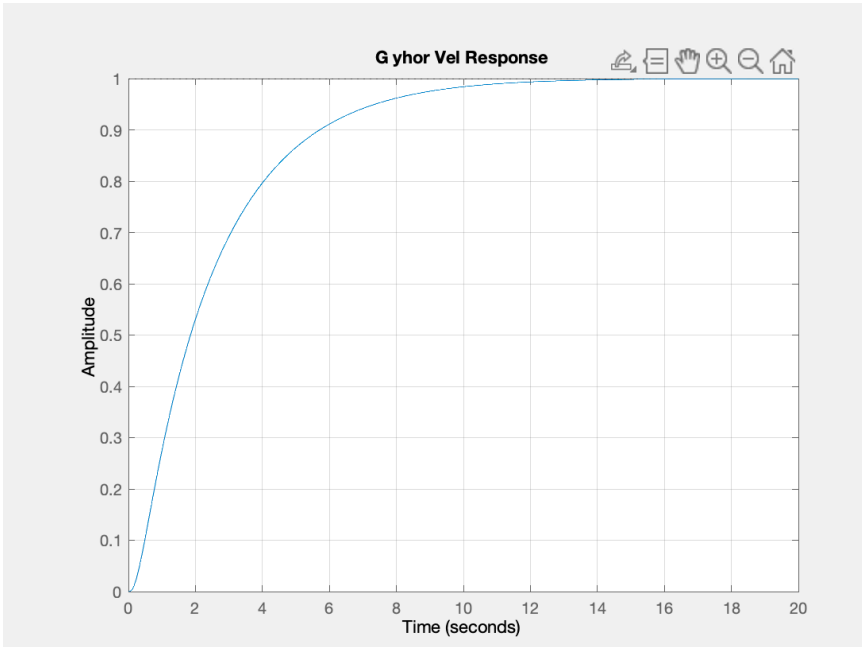
Gains: `(-H(kpd,kdt), 0.05, 0.0001, 0.07);`



Gains: `(H(kpsi,kdr), 0.11, 0.002, 0.07);`



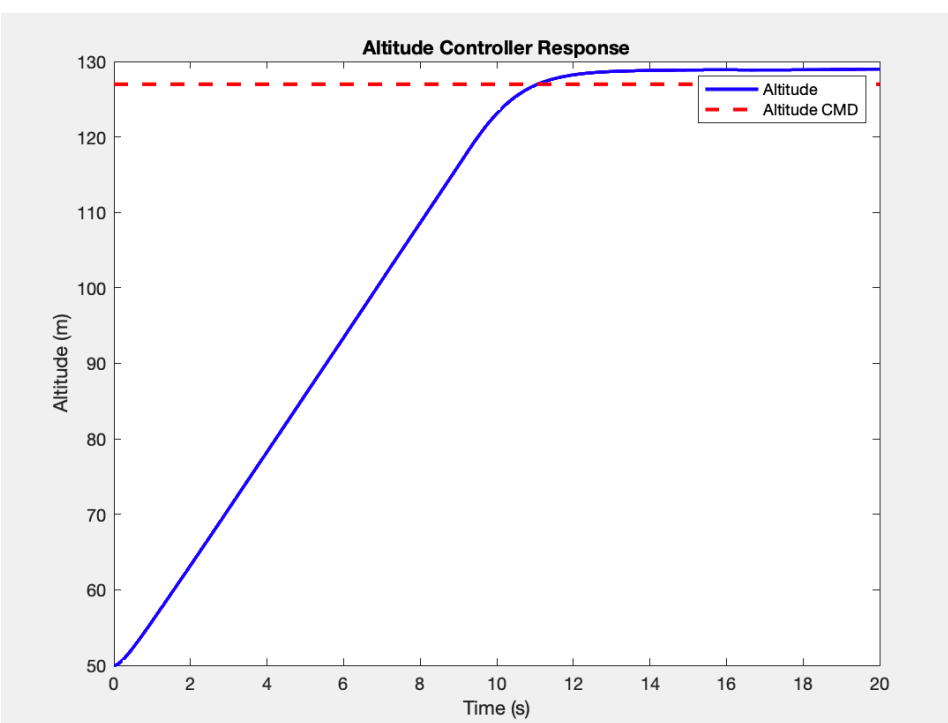
Gains: `-0.035, -0.025, 0);`



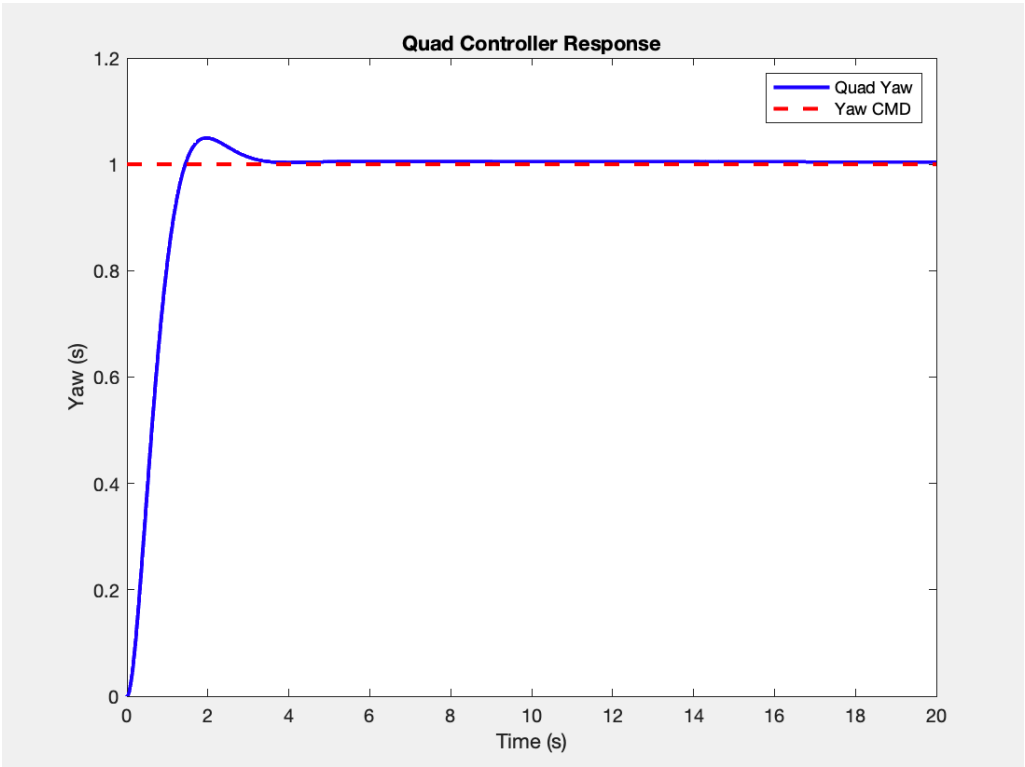
Gains: `0.05, 0.025, 0);`

Once I was happy with the responses, I then went and tried the controllers on the quadcopter using steps and commands.

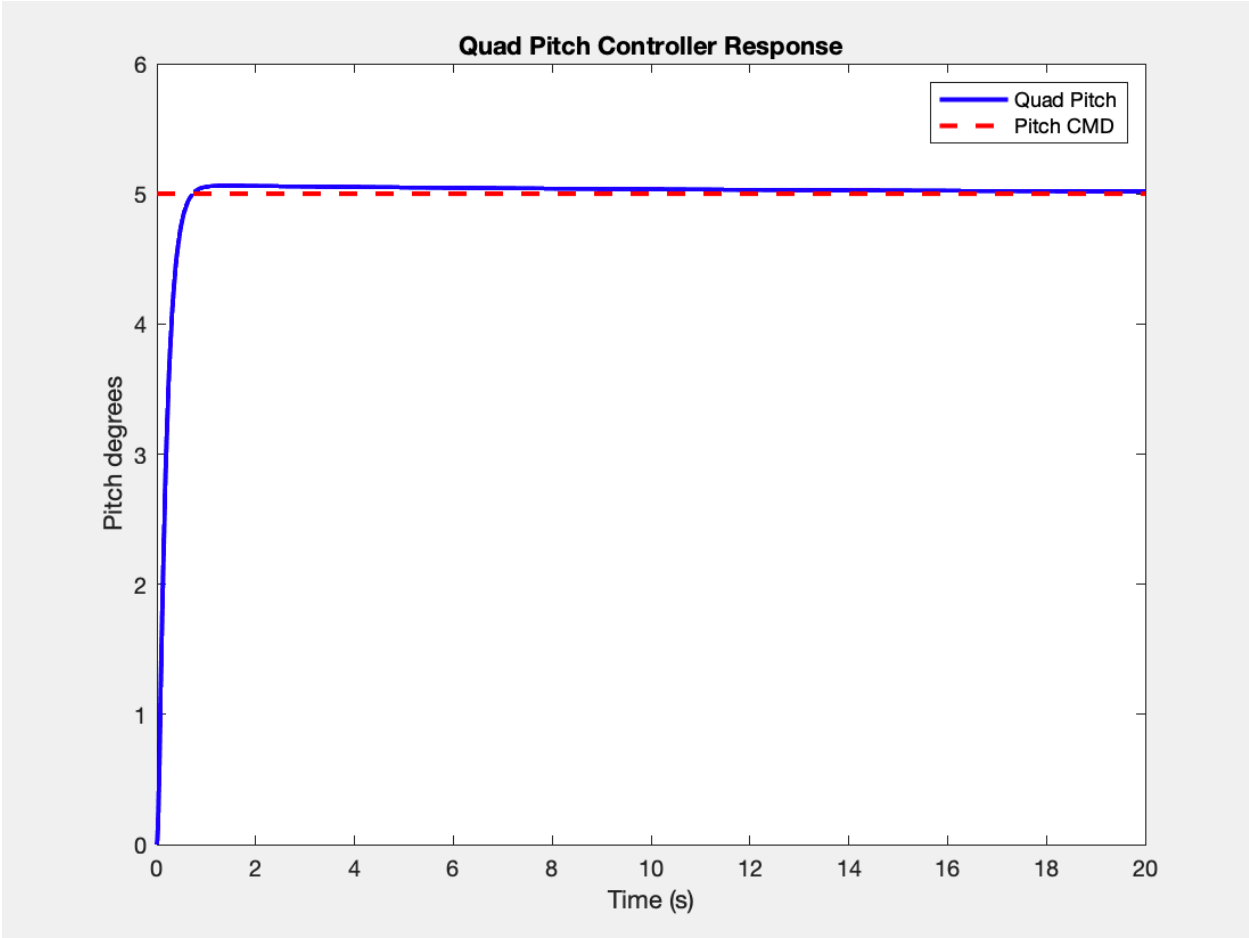
For altitude test, I set the command to 127 meters:



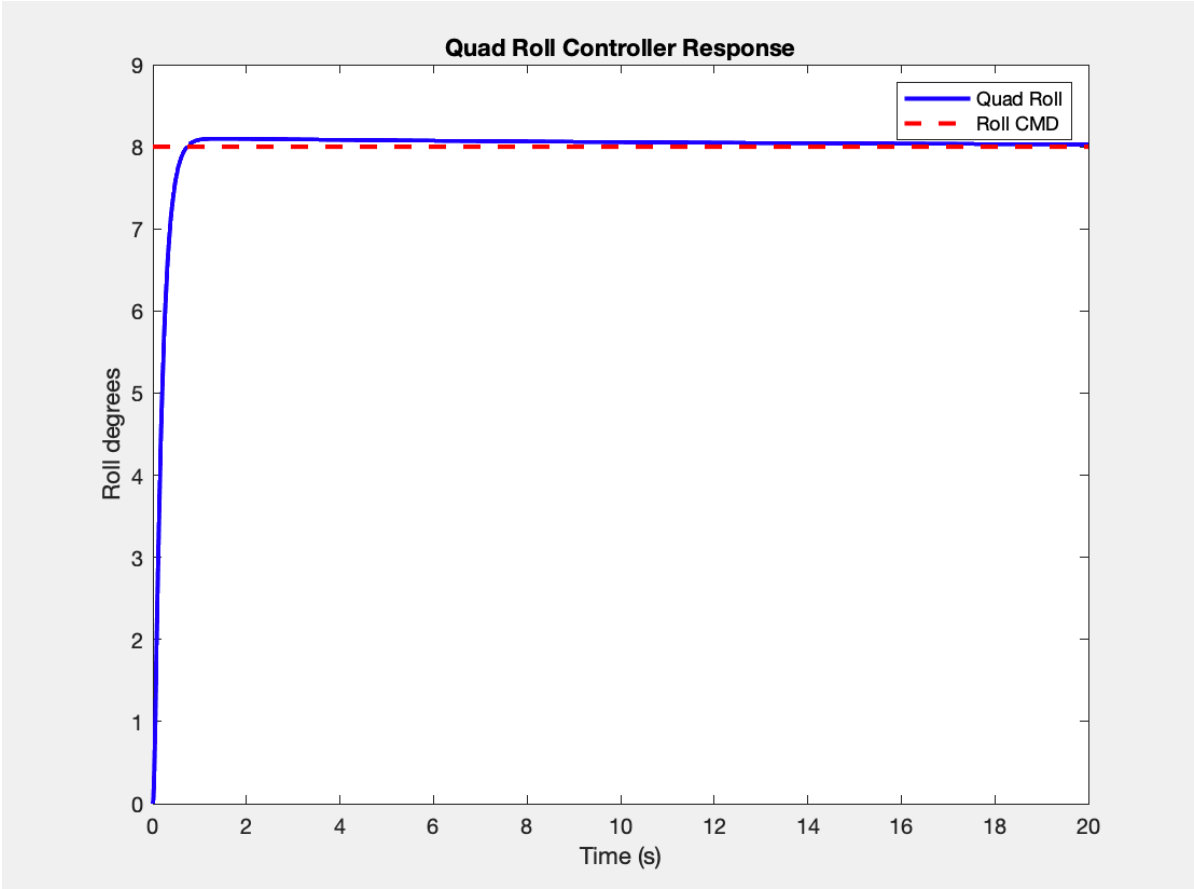
For yaw I set command to 1 degree:



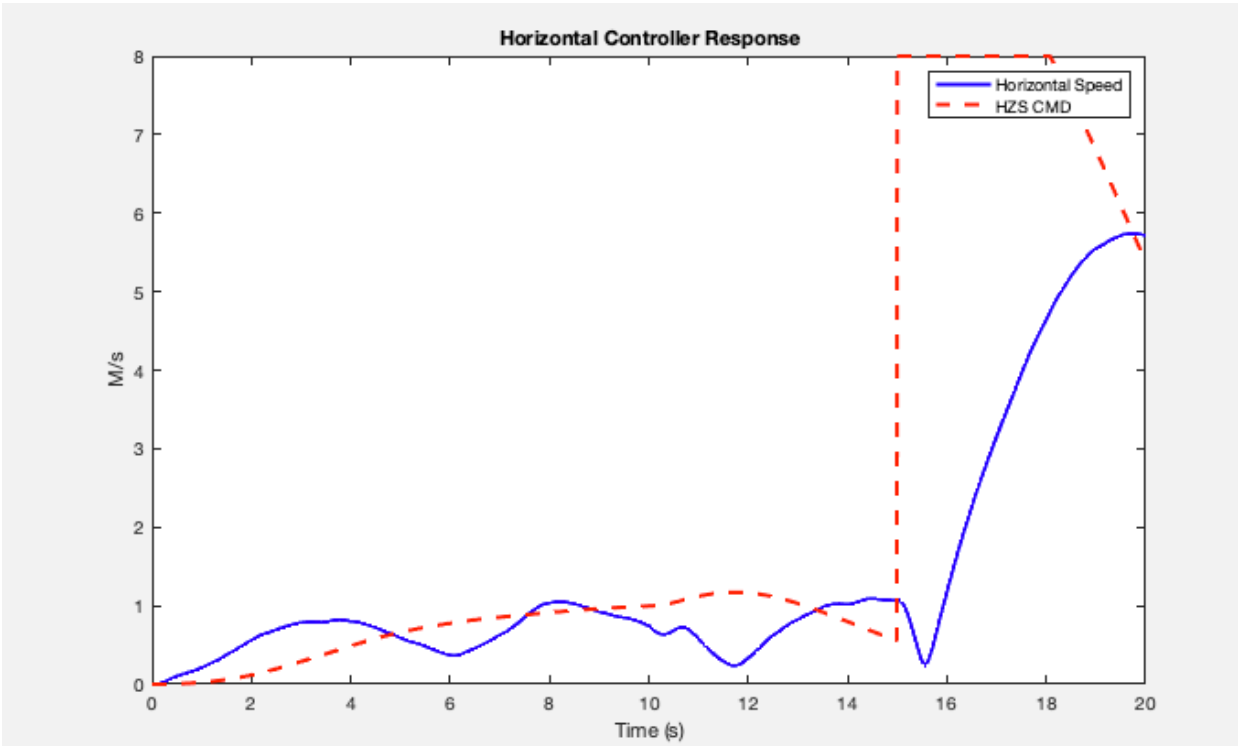
For pitch I set the command to 5 degrees:



For roll I set command to 8 degrees:



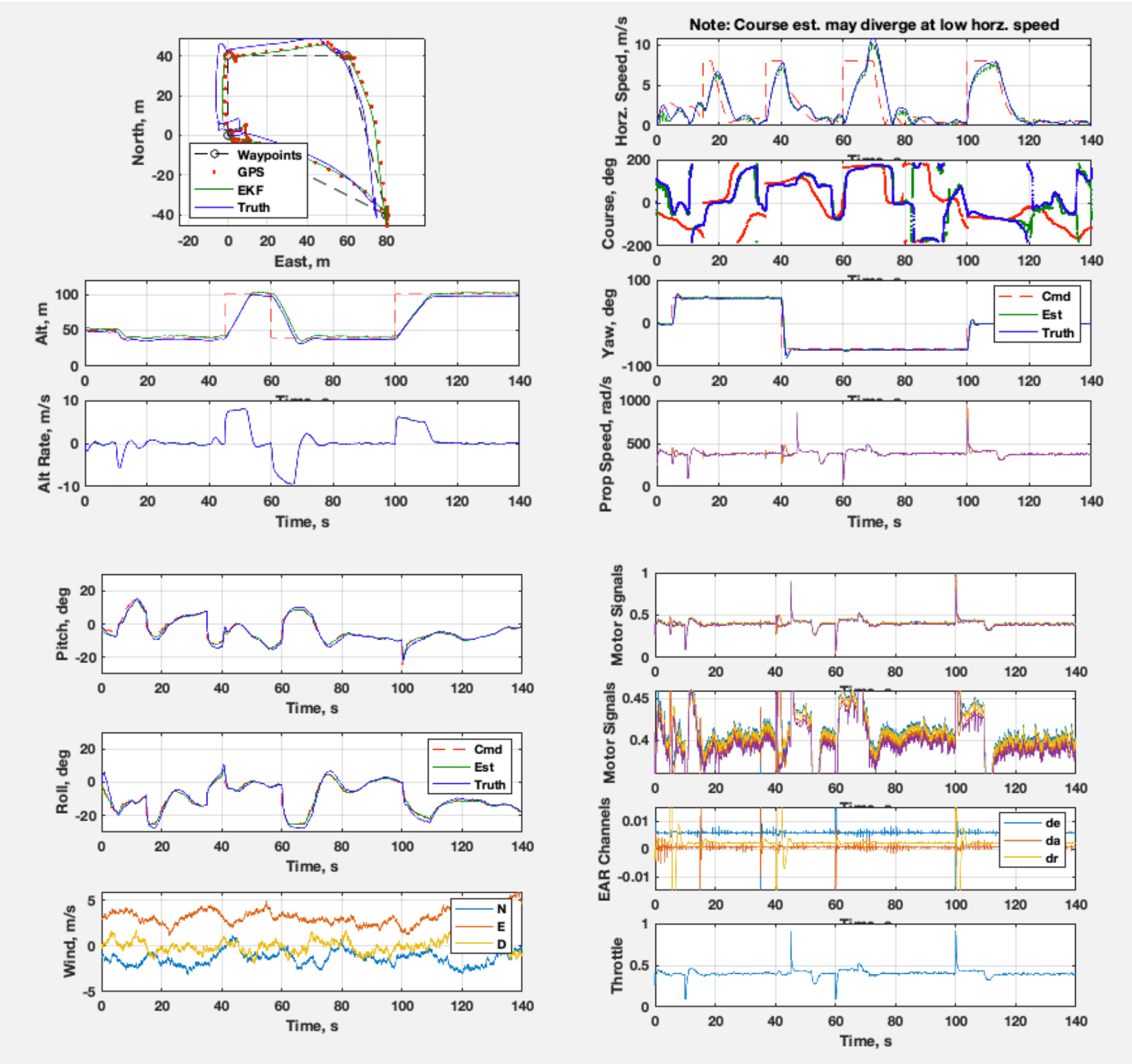
Used default commands to run simulation for horizontal speed controller Utilizes both X and Y controllers (plots of testing each individual X and Y controller were provided earlier in this question



9)

Using linearized models in tuning and verifying a controller has several benefits, including simplicity, analytical insights, efficient controller design, and overall verification of model accuracy. In the context of the simulation, both analytically derived models from transfer functions and numerically derived models from A and B matrices are used for a better understanding of the system's behavior. Analytical models provide fundamental and accurate insights into the system's dynamics, while numerical models help find nonlinear effects and complex dynamics. Comparing the open-loop step responses from both types of models allows engineers to assess model consistency and accuracy, aiding in controller design and verification. Additionally, looking at the system's response to small deviations from trim conditions helps evaluate controller robustness. Leveraging both analytical and numerical models ensures a thorough understanding of the system's behavior and increases the performance and robustness of controller designs in quadcopter simulation.

10)


Note: Course est. may diverge at low horz. speed
North, m
East, m
Waypoints
GPS
EKF
Truth
Horz. Speed, m/s
Course, deg
Alt, m
Yaw, deg
Cmd
Est
Truth
Alt Rate, m/s
Prop Speed, rad/s
Time, s
Pitch, deg
Motor Signals
Roll, deg
Wind, m/s
N
E
D
EAR Channels
de
da
dr
Throttle

Appendix:

mapChannelsToMotors.m

```
% load_quadsim.m
%
% Initializer for quadsim.mdl.
%
% Developed for JHU EP 525.461, UAV Systems & Control
% Adapted from design project in "Small Unmanned Aircraft: Theory and
% Practice", RWBeard & TWMcClain, Princeton Univ. Press, 2012

function [delta_1, delta_2, delta_3, delta_4] =
 mapChannelsToMotors(delta_e,delta_a,delta_r,delta_t)
% Map quadcopter channels to motors
%
% Inputs:
%    delta_e: Elevator
%    delta_a: Aileron
%    delta_r: Rudder
%    delta_t: Throttle
%
% Outputs:
%    delta_1: front right motor
%    delta_2: back left motor
%    delta_3: front left motor
%    delta_4: back right motor

    % Map channels to motors
    %     3   1
    %       X
    %     2   4

    M_channel_motors = [1/4, -1/4, 1/4, -1/4;
                        -1/4, 1/4, 1/4, -1/4;
                        1/4,  1/4, -1/4, -1/4;
                        1/4,  1/4, 1/4,  1/4;];

    motor_singal_gain = inv(M_channel_motors) ...
        * [delta_e; delta_a; delta_r; delta_t;];

    delta_1 = motor_singal_gain(1); % front right
    delta_2 = motor_singal_gain(2); % back left
    delta_3 = motor_singal_gain(3); % front left
    delta_4 = motor_singal_gain(4); % back right
end

Not enough input arguments.

Error in mapChannelsToMotors (line 35)
        * [delta_e; delta_a; delta_r; delta_t;];
```

Published with MATLAB® R2023a

quadsim_forces_moments.m

```
% quadsim_forces_moments.m
%
% Developed for JHU EP 525.461, UAV Systems & Control
% Adapted from design project in "Small Unmanned Aircraft: Theory and
% Practice", RWBeard & TWMcClain, Princeton Univ. Press, 2012
%
function out = quadsim_forces_moments(uu, P)

    % Extract variables from input vector uu
    %   uu = [wind_ned(1:3); deltas(1:4); x(1:12); time(1)];
    k=(1:3);          wind_ned=uu(k);   % Total wind vector, ned, m/s
    k=k(end)+(1:4);   deltas=uu(k);     % Control surface commands: [delta_e
 delta_a delta_r delta_t]
    k=k(end)+(1:12);  x=uu(k);          % states
    k=k(end)+(1);     time=uu(k);       % Simulation time, s

    % Extract state variables from x
    pn    = x(1);   % North position, m
    pe    = x(2);   % East position, m
    pd    = x(3);   % Down position, m
    u     = x(4);   % body-x groundspeed component, m/s
    v     = x(5);   % body-y groundspeed component, m/s
    w     = x(6);   % body-z groundspeed component, m/s
    phi   = x(7);   % EulerAngle: roll, rad
    theta = x(8);   % EulerAngle: pitch, rad
    psi   = x(9);   % EulerAngle: yaw, rad
    p     = x(10);  % body rate about x, rad/s
    q     = x(11);  % body rate about y, rad/s
    r     = x(12);  % body rate about z, rad/s

    % Combine states to vector form for convenience
    P_ned = [pn; pe; pd];   % NED position, m
    vg_b  = [u; v; w];      % Groundspeed vector, body frame, m/s
    w_b   = [p; q; r];      % body rates about x,y,z, rad/s

    % Extract control commands from deltas
    delta_e = deltas(1); % Elevator, +/-
    delta_a = deltas(2); % Aileron, +/-
    delta_r = deltas(3); % Rudder, +/-
    delta_t = deltas(4); % Throttle, 0 - 1
    [delta_1, delta_2, delta_3, delta_4] =
 mapChannelsToMotors(delta_e,delta_a,delta_r,delta_t);

    % Your code goes below...

    % Prop rotation rates
    omega_1 = P.k_omega*delta_1 + P.prop_1_omega_bias;
    omega_2 = P.k_omega*delta_2 + P.prop_2_omega_bias;
    omega_3 = P.k_omega*delta_3 + P.prop_3_omega_bias;
    omega_4 = P.k_omega*delta_4 + P.prop_4_omega_bias;

    % Torques
```

```
    Tp_1 = P.k_Tp*(omega_1^2);
    Tp_2 = P.k_Tp*(omega_2^2);
    Tp_3 = P.k_Tp*(omega_3^2);
    Tp_4 = P.k_Tp*(omega_4^2);

    % Air in calculatins
    R_ned2b = eulerToRotationMatrix(phi,theta,psi);
    wind_b = R_ned2b*wind_ned;
    va_b = vg_b-wind_b;
    Vair_in = -(va_b(3));

    % Forces
    Fp_1 = P.rho*P.C_prop*P.S_prop ...
        *(Vair_in + (omega_1/P.k_omega)*(P.k_motor - Vair_in)) ...
        *((omega_1/P.k_omega)*(P.k_motor - Vair_in));

    Fp_2 = P.rho*P.C_prop*P.S_prop ...
        *(Vair_in + (omega_2/P.k_omega)*(P.k_motor - Vair_in)) ...
        *((omega_2/P.k_omega)*(P.k_motor - Vair_in));

    Fp_3 = P.rho*P.C_prop*P.S_prop ...
        *(Vair_in + (omega_3/P.k_omega)*(P.k_motor - Vair_in)) ...
        *((omega_3/P.k_omega)*(P.k_motor - Vair_in));

    Fp_4 = P.rho*P.C_prop*P.S_prop ...
        *(Vair_in + (omega_4/P.k_omega)*(P.k_motor - Vair_in)) ...
        *((omega_4/P.k_omega)*(P.k_motor - Vair_in));

    % Set moments
    m_x = (P.delta_y*(Fp_3 + Fp_2) - dy*(Fp_1 + Fp_4));
    m_y = (P.delta_x*(Fp_1 + Fp_3) - dx*(Fp_2 + Fp_4));
    m_z = ((Tp_1 + Tp_2) - (Tp_3 + Tp_4));

    % Add matrices
    f_grav_b = P.mass*R_ned2b*[0; 0; P.gravity];
    f_props_b = [0;0;-(Fp_1+Fp_2+Fp_3+Fp_4)];
    f_b_rotorDrag = -P.mu_rotorDrag*[vg_b(1)-wind_b(1);vg_b(2)-wind_b(2);0];

    % Combine
    f_b=f_grav_b+f_props_b+f_b_rotorDrag;
    m_b=[m_x;m_y;m_z];

    % Compile function output
    out = [f_b; m_b]; % Length 3+3=6

end
```

```
Not enough input arguments.

Error in quadsim_forces_moments (line 11)
    k=(1:3);          wind_ned=uu(k);   % Total wind vector, ned, m/s
```

Published with MATLAB® R2023a

quadsim_sensors.m

```
% quadsim_sensors.m
%
% Developed for JHU EP 525.461, UAV Systems & Control
% Adapted from design project in "Small Unmanned Aircraft: Theory and
% Practice", RWBeard & TWMcClain, Princeton Univ. Press, 2012
%
function out = quadsim_sensors(uu, P)

    % Extract variables from input vector uu
    %   uu = [f_and_m(1:6); x(1:12); wind_ned(1:3); time(1)];
    k=(1:6);            f_and_m=uu(k);   % Forces and Moments, body
    k=k(end)+(1:12);    x=uu(k);         % states
    k=k(end)+(1:3);     wind_ned=uu(k);  % wind vector, ned, m/s
    k=k(end)+(1);       time=uu(k);      % Simulation time, s

    % Extract forces and moments from f_and_m
    fb_x = f_and_m(1); % Total force along body x, N
    fb_y = f_and_m(2); % Total force along body y, N
    fb_z = f_and_m(3); % Total force along body z, N
    mb_x = f_and_m(4); % Total moment about body x, N-m
    mb_y = f_and_m(5); % Total moment about body y, N-m
    mb_z = f_and_m(6); % Total moment about body z, N-m

    % Extract state variables from x
    pn    = x(1);   % North position, m
    pe    = x(2);   % East position, m
    pd    = x(3);   % Down position, m
    u     = x(4);   % body-x groundspeed component, m/s
    v     = x(5);   % body-y groundspeed component, m/s
    w     = x(6);   % body-z groundspeed component, m/s
    phi   = x(7);   % EulerAngle: roll, rad
    theta = x(8);   % EulerAngle: pitch, rad
    psi   = x(9);   % EulerAngle: yaw, rad
    p     = x(10);  % body rate about x, rad/s
    q     = x(11);  % body rate about y, rad/s
    r     = x(12);  % body rate about z, rad/s

    % Gyro
    p_gyro = p + P.sigma_noise_gyro*randn; % rad/s
    q_gyro = q + P.sigma_noise_gyro*randn; % rad/s
    r_gyro = r + P.sigma_noise_gyro*randn; % rad/s

    % Accelerometer
    ax = ((1/P.mass)*fb_x) + P.gravity*sin(theta);
    ay = ((1/P.mass)*fb_y) - P.gravity*cos(theta)*sin(phi);
    az = ((1/P.mass)*fb_z) - P.gravity*cos(theta)*cos(phi);

    ax_accel= ax + P.sigma_noise_accel*randn;
    ay_accel= ay + P.sigma_noise_accel*randn;
    az_accel= az + P.sigma_noise_accel*randn;

    % Barometric Pressure Altimeter
```

```
    P0 = 101325;
    R = 8.31432;
    M = 0.0289644;%
    T = 5/9*(P.air_temp_F-32)+273.15;

    persistent bias_static_press
    if(time==0)
        bias_static_press = P.sigma_bias_static_press*randn;
    end
    true_static_press = P0*exp(((-M*P.gravity)*(P.h0_ASL-pd))/(R*T));
    static_press = true_static_press + bias_static_press +
P.sigma_noise_static_press*randn;

    persistent bias_diff_press
    if(time==0)
        bias_diff_press = P.sigma_bias_diff_press*randn;
    end
    R_ned2b = eulerToRotationMatrix(phi,theta,psi);
    v_wb = R_ned2b*wind_ned;
    v_ab = [u;v;w]-v_wb;
    Va = sqrt(v_ab(1)^2+v_ab(2)^2+v_ab(3)^2);
    true_diff_press = 0.5*P.rho*Va^2;
    diff_press = true_diff_press + bias_diff_press +
P.sigma_noise_diff_press*randn;

    persistent bias_mag
    if(time==0)
        bias_mag = P.sigma_bias_mag*randn;
    end
    psi_mag = psi + bias_mag + P.sigma_noise_mag*randn; % Magnetometer
measurement, rad

    % GPS Position and Velocity Measurements
    persistent time_gps_prev gps_north_error gps_east_error gps_alt_error ...
               pn_gps pe_gps alt_gps Vn_gps Ve_gps Vd_gps
    if(time==0)
        gps_north_error = P.sigma_bias_gps_north*randn;
        gps_east_error = P.sigma_bias_gps_east*randn;
        gps_alt_error = P.sigma_bias_gps_alt*randn;
        time_gps_prev = -inf;
    end

    if(time>time_gps_prev+P.Ts_gps)

        % Gauss-Markov growth of GPS position errors
        gps_north_error = exp(-P.Ts_gps/P.tau_gps)*gps_north_error +
P.sigma_eta_gps_north*randn*sqrt(P.Ts_gps);
        gps_east_error  = exp(-P.Ts_gps/P.tau_gps)*gps_east_error +
P.sigma_eta_gps_east*randn*sqrt(P.Ts_gps);
        gps_alt_error   = exp(-P.Ts_gps/P.tau_gps)*gps_alt_error +
P.sigma_eta_gps_alt*randn*sqrt(P.Ts_gps);

        % GPS Position Measurements
        pn_gps = pn + gps_north_error;
```

```
        pe_gps = pe + gps_east_error;
        alt_gps= -pd + gps_alt_error;

        % GPS Velocity Measurements
            % Compute Rotation Matrices
        R_ned2b = eulerToRotationMatrix(phi,theta,psi);
        R_b2ned = R_ned2b';
            % Manipulate states
        vg_ned = R_b2ned*[u; v; w];
        Vn_gps = vg_ned(1) + P.sigma_noise_gps_speed*randn;
        Ve_gps = vg_ned(2) + P.sigma_noise_gps_speed*randn;
        Vd_gps = vg_ned(3) + P.sigma_noise_gps_speed*randn;

        time_gps_prev = time;
    end

    % Compile output vector
    out = [ ...
            pn_gps; ...
            pe_gps; ...
            alt_gps;  ...
            Vn_gps; ...
            Ve_gps; ...
            Vd_gps; ...
            p_gyro; ...
            q_gyro; ...
            r_gyro; ...
            ax_accel;...
            ay_accel;...
            az_accel;...
            static_press; ...
            diff_press; ...
            psi_mag;...
            0; % future use
            0; % future use
            0; % future use
          ]; % Length: 18

end
```

```
Not enough input arguments.

Error in quadsim_sensors (line 11)
    k=(1:6);            f_and_m=uu(k);    % Forces and Moments, body
```

Published with MATLAB® R2023a

quadsim_estimates.m

```
% quadsim_estimates.m
%
% Developed for JHU EP 525.461, UAV Systems & Control
% Adapted from design project in "Small Unmanned Aircraft: Theory and
% Practice", RWBeard & TWMcClain, Princeton Univ. Press, 2012
%
function out = quadsim_estimates(uu,P)

    % Extract variables from input vector uu
    %   uu = [meas(1:18); time(1)];
    k=(1:18);                meas=uu(k);   % Sensor Measurements
    k=k(end)+(1);            time=uu(k);   % Simulation time, s

    % Extract mesurements
    k=1;
    pn_gps = meas(k); k=k+1; % GPS North Measurement, m
    pe_gps = meas(k); k=k+1; % GPS East Measurement, m
    alt_gps= meas(k); k=k+1; % GPS Altitude Measurement, m
    Vn_gps = meas(k); k=k+1; % GPS North Speed Measurement, m/s
    Ve_gps = meas(k); k=k+1; % GPS East Speed Measurement, m/s
    Vd_gps = meas(k); k=k+1; % GPS Downward Speed Measurement, m/s
    p_gyro = meas(k); k=k+1; % Gyro Body Rate Meas. about x, rad/s
    q_gyro = meas(k); k=k+1; % Gyro Body Rate Meas. about y, rad/s
    r_gyro = meas(k); k=k+1; % Gyro Body Rate Meas. about z, rad/s
    ax_accel = meas(k); k=k+1; % Accelerometer Meas along x, m/s/s
    ay_accel = meas(k); k=k+1; % Accelerometer Meas along y, m/s/s
    az_accel = meas(k); k=k+1; % Accelerometer Meas along z, m/s/s
    static_press = meas(k); k=k+1; % Static Pressure Meas., N/m^2
    diff_press = meas(k); k=k+1; % Differential Pressure Meas., N/m^2
    psi_mag = meas(k); k=k+1; % Yaw Meas. from Magnetometer, rad
    future_use = meas(k); k=k+1;
    future_use = meas(k); k=k+1;
    future_use = meas(k); k=k+1;

    diff_press = 0;

    % Filter raw measurements
    persistent lpf_static_press ...
               lpf_diff_press ...
               lpf_p_gyro ...
               lpf_q_gyro ...
               lpf_r_gyro ...
               lpf_psi_mag
    if(time==0)
        % Filter initializations
        lpf_static_press = static_press;
        lpf_diff_press = diff_press;
        lpf_p_gyro = p_gyro;
        lpf_q_gyro = q_gyro;
        lpf_r_gyro = r_gyro;
        lpf_psi_mag = psi_mag;
    end
```

```
    lpf_static_press =
 LPF(static_press,lpf_static_press,P.tau_static_press,P.Ts);
    lpf_diff_press   = LPF(diff_press,lpf_diff_press,P.tau_diff_press,P.Ts);
    lpf_p_gyro = LPF(p_gyro,lpf_p_gyro,P.tau_gyro,P.Ts);
    lpf_q_gyro = LPF(q_gyro,lpf_q_gyro,P.tau_gyro,P.Ts);
    lpf_r_gyro = LPF(r_gyro,lpf_r_gyro,P.tau_gyro,P.Ts);
    lpf_psi_mag = LPF(psi_mag,lpf_psi_mag,P.tau_mag,P.Ts);

    P0 = 101325;
    R = 8.31432;
    M = 0.0289644;
    T = 5/9*(P.air_temp_F-32)+273.15;
    P_launch = P0*exp(((-M*P.gravity)/(R*T))*P.h0_ASL);
    h_baro = ((-R*T)/(M*P.gravity))*log(lpf_static_press/P_launch);

    Va_pitot = 0;

    Q_att = diag([P.sigma_noise_gyro; P.sigma_noise_gyro].^2);
    R_att = (10^4)*diag([P.sigma_noise_accel; P.sigma_noise_accel;
 P.sigma_noise_accel].^2);
    sigma_ekfInitUncertainty = [5*pi/180; 5*pi/180];

    persistent xhat_att P_att
    if(time==0)
        xhat_att=[0; 0]; % States: [phi; theta]
        P_att=diag(sigma_ekfInitUncertainty.^2);
    end

    phi = xhat_att(1);
    theta = xhat_att(2);

    N=10;
    for i=1:N
        f_att = [1 sin(phi)*tan(theta) cos(phi)*tan(theta); ...
            0 cos(phi) -sin(phi)]*[p_gyro; q_gyro; r_gyro];

        A_att = [q_gyro*cos(phi)*tan(theta)-r_gyro*sin(phi)*tan(theta)
 (q_gyro*sin(phi)+r_gyro*cos(phi))*(1+tan(theta)^2); ...
            -q_gyro*sin(phi)-r_gyro*cos(phi) 0];

        xhat_att = xhat_att+(P.Ts/N)*f_att;

        P_att = P_att + (P.Ts/N)*(A_att*P_att + P_att*A_att' + Q_att);

        P_att = real(.5*P_att + .5*P_att');
    end

    y_att = [ax_accel; ay_accel; az_accel]; % Vector of actual measurements

    h_att = [q_gyro*Va_pitot*sin(theta)+P.gravity*sin(theta); ...
        r_gyro*Va_pitot*cos(theta)-p_gyro*Va_pitot*sin(theta)-
P.gravity*cos(theta)*sin(phi); ...
```

```
        -q_gyro*Va_pitot*cos(theta)-P.gravity*cos(theta)*cos(phi)]; %
 Mathematical model of measurements based on xhat

    C_att = [0 q_gyro*Va_pitot*cos(theta)+P.gravity*cos(theta); ...
        -P.gravity*cos(phi)*cos(theta) -r_gyro*Va_pitot*sin(theta)-
p_gyro*Va_pitot*cos(theta)+P.gravity*sin(phi)*sin(theta); ...
        P.gravity*sin(phi)*cos(theta)
 q_gyro*Va_pitot*sin(theta)+P.gravity*cos(phi)*sin(theta)]; % Linearization
 (Jacobian) of h(x,...) wrt x

    L_att = (P_att*C_att')/(C_att*P_att*C_att' + R_att); % Kalman Gain matrix

    I_att = eye(length(xhat_att));
    P_att = (I_att - L_att*C_att)*P_att;
    xhat_att = xhat_att + L_att*(y_att - h_att);
    xhat_att = mod(xhat_att+pi,2*pi)-pi;
    phi_hat_unc   = sqrt(P_att(1,1));
    theta_hat_unc = sqrt(P_att(2,2));

    Q_gps = diag([.1 .1 .1 .1 .1 .1].^2);
    R_gps = diag([2 2 2 .1 .1 .1].^2);
    sigma_ekfInitUncertainty = [P.sigma_eta_gps_north P.sigma_eta_gps_east -
P.sigma_eta_gps_alt ...
        P.sigma_noise_gps_speed P.sigma_noise_gps_speed
 P.sigma_noise_gps_speed];

    persistent xhat_gps P_gps prev_pn_gps prev_pe_gps
    if(time==0)
        xhat_gps=[pn_gps; pe_gps; -alt_gps; Vn_gps; Ve_gps; Vd_gps];
        P_gps=diag(sigma_ekfInitUncertainty.^2);
        prev_pn_gps = pn_gps;
        prev_pe_gps = pe_gps;
    end

    p_n = xhat_gps(1);
    p_e = xhat_gps(2);
    p_d = xhat_gps(3);
    v_n = xhat_gps(4);
    v_e = xhat_gps(5);
    v_d = xhat_gps(6);

    N=10;
    for i=1:N
        R_ned2b = eulerToRotationMatrix(phi,theta,psi_mag);
        f_gps = [v_n; v_e; v_d; ((R_ned2b'*[ax_accel; ay_accel;
 az_accel])+[0;0;P.gravity])];
        A_gps = [0 0 0 1 0 0; 0 0 0 0 1 0; 0 0 0 0 0 1; zeros(3, 6)];
        xhat_gps = xhat_gps+(P.Ts/N)*f_gps;
        P_gps = P_gps + (P.Ts/N)*(A_gps*P_gps + P_gps*A_gps' + Q_gps);
        P_gps = real(.5*P_gps + .5*P_gps');
    end

    if prev_pn_gps ~= pn_gps || prev_pe_gps ~= pe_gps
        meas_available = 1;
```

```
    else
        meas_available = 0;
    end

    if meas_available
        y_gps = [pn_gps; pe_gps; -alt_gps; Vn_gps; Ve_gps; Vd_gps]; % Vector
of actual measurements
        h_gps = [p_n; p_e; p_d; v_n; v_e; v_d]; % Mathematical model of
measurements based on xhat
        C_gps = eye(length(xhat_gps)); % Linearization (Jacobian) of h(x,...)
wrt x
        L_gps = (P_gps*C_gps')/(C_gps*P_gps*C_gps' + R_gps); % Kalman Gain
matrix
        I_gps = eye(length(xhat_gps));
        P_gps = (I_gps - L_gps*C_gps)*P_gps; % Covariance matrix updated with
measurement information
        xhat_gps = xhat_gps + L_gps*(y_gps - h_gps); % States updated with
measurement information
        pn_hat_unc   = sqrt(P_gps(1,1)); % EKF-predicted uncertainty in pn
estimate, rad
        pe_hat_unc = sqrt(P_gps(2,2)); % EKF-predicted uncertainty in pe
estimate, rad
        pd_hat_unc = sqrt(P_gps(3,3)); % EKF-predicted uncertainty in pd
estimate, rad
        vn_hat_unc = sqrt(P_gps(4,4)); % EKF-predicted uncertainty in vn
estimate, m/s
        ve_hat_unc = sqrt(P_gps(5,5)); % EKF-predicted uncertainty in ve
estimate, m/s
        vd_hat_unc = sqrt(P_gps(6,6)); % EKF-predicted uncertainty in vd
estimate, m/s
    end

    prev_pn_gps = pn_gps;
    prev_pe_gps = pe_gps;

    pn_hat=xhat_gps(1);
    pe_hat=xhat_gps(2);
    h_hat=h_baro;
    Va_hat=Va_pitot;
    phi_hat=xhat_att(1);
    theta_hat=xhat_att(2);
    psi_hat=lpf_psi_mag;
    p_hat=lpf_p_gyro;
    q_hat=lpf_q_gyro;
    r_hat=lpf_r_gyro;
    Vn_hat=xhat_gps(4);
    Ve_hat=xhat_gps(5);
    Vd_hat=xhat_gps(6);
    wn_hat=0;
    we_hat=0;

    % Compile output vector
    out = [...
            pn_hat;...    % 1
```

```
            pe_hat;...     % 2
            h_hat;...      % 3
            Va_hat;...     % 4
            phi_hat;...    % 5
            theta_hat;... % 6
            psi_hat;...    % 7
            p_hat;...      % 8
            q_hat;...      % 9
            r_hat;...      % 10
            Vn_hat;...     % 11
            Ve_hat;...     % 12
            Vd_hat;...     % 13
            wn_hat;...     % 14
            we_hat;...     % 15
            phi_hat_unc;...    % 16
            theta_hat_unc;... % 17
            0; % future use
            0; % future use
            0; % future use
            0; % future use
            0; % future use
            0; % future use
        ];

end

function y = LPF(u,yPrev,tau,Ts)
%
%  Y(s)       a            1
% ------ = ------- = -----------,  tau: Filter time contsant, s
%  U(s)     s + a     tau*s + 1          ( tau = 1/a )
%

alpha_LPF = exp(-Ts/tau);
y = alpha_LPF*yPrev + (1 - alpha_LPF)*u;

end
```

```
Not enough input arguments.

Error in quadsim_estimates (line 11)
    k=(1:18);                meas=uu(k);    % Sensor Measurements
```

Published with MATLAB® R2023a

quadsim_control.m

```
% quadsim_control.m
%
% Flight control logic for quadsim
%
% Developed for JHU EP 525.461, UAV Systems & Control
% Adapted from design project in "Small Unmanned Aircraft: Theory and
% Practice", RWBeard & TWMcClain, Princeton Univ. Press, 2012
%
function out = quadsim_control(uu,P)
    persistent call_count;

    % Check if call_count is empty (first call)
    if isempty(call_count)
        call_count = 1;
    end

    % Extract variables from input vector uu
    %   uu = [traj_cmds(1:4); estimates(1:23); time(1)];
    k=(1:4);         traj_cmds=uu(k); % Trajectory Commands
    k=k(end)+(1:23); estimates=uu(k); % Feedback state estimates
    k=k(end)+(1);    time=uu(k);      % Simulation time, s

    % Extract variables from traj_cmds
    h_c      = traj_cmds(1);  % commanded altitude (m)
    Vhorz_c  = traj_cmds(2);  % commanded horizontal speed (m/s) (change from uavsim)
    chi_c    = traj_cmds(3);  % commanded course (rad)
    psi_c    = traj_cmds(4);  % yaw course (rad) (change from uavsim)

    % Extract variables from estimates
    pn_hat       = estimates(1);  % inertial North position, m
    pe_hat       = estimates(2);  % inertial East position, m
    h_hat        = estimates(3);  % altitude, m
    Va_hat       = estimates(4);  % airspeed, m/s
    phi_hat      = estimates(5);  % roll angle, rad
    theta_hat    = estimates(6);  % pitch angle, rad
    psi_hat      = estimates(7);  % yaw angle, rad
    p_hat        = estimates(8);  % body frame roll rate, rad/s
    q_hat        = estimates(9);  % body frame pitch rate, rad/s
    r_hat        = estimates(10); % body frame yaw rate, rad/s
    Vn_hat       = estimates(11); % north speed, m/s
    Ve_hat       = estimates(12); % east speed, m/s
    Vd_hat       = estimates(13); % downward speed, m/s
    wn_hat       = estimates(14); % wind North, m/s
    we_hat       = estimates(15); % wind East, m/s
    future_use   = estimates(16:23);

    % Initialize controls to trim (to be with PID logic)
    delta_e=P.delta_e0;
    delta_a=P.delta_a0;
    delta_r=P.delta_r0;
    delta_t=P.delta_t0;

    % Initialize autopilot commands (may be overwritten with PID logic)
    phi_c = 0;
    theta_c = 0;
```

```
Vhx_c_list = zeros(1, 2001);
Vhy_c_list = zeros(1, 2001);

Vhx_hat_list = zeros(1, 2001);
Vhy_hat_list = zeros(1, 2001);

R_ned2b = eulerToRotationMatrix(phi_hat,theta_hat,psi_hat);
vgb_hat = R_ned2b*[Vn_hat; Ve_hat; Vd_hat];
hdot_hat = -vgb_hat(3);

% Set "first-time" flag, which is used to initialize PID integrators
firstTime=(time==0);

% Flight control logic
%   <code goes here>
% e.g.
%    delta_a = PID_roll_hold(phi_c, phi_hat, p_hat, firstTime, P);
%
% Note: For logging purposes, use variables:
%         Vhorz_c,  chi_c, h_c, phi_c, theta_c, psi_c

% Getting trajectory commands
[WP_n, WP_e, h_c, psi_c] = get_quadsim_trajectory_commands(time);
chi_c = atan2(WP_e-pe_hat, WP_n - pn_hat);
k_pos = 0.25;
Vhorz_c = k_pos*sqrt((WP_e-pe_hat)^2 + (WP_n - pn_hat)^2);
if (Vhorz_c > 8)
     Vhorz_c = 8;
end
%
Vn_cmd = Vhorz_c*cos(chi_c);
Ve_cmd = Vhorz_c*sin(chi_c);

Vh_c = [cos(psi_hat) sin(psi_hat); -sin(psi_hat) cos(psi_hat)]*[Vn_cmd; Ve_cmd];
Vhx_c = Vh_c(1); Vhy_c = Vh_c(2);

Vh_hat = [cos(psi_hat) sin(psi_hat); -sin(psi_hat) cos(psi_hat)]*[Vn_hat; Ve_hat];
Vhx_hat = Vh_hat(1); Vhy_hat = Vh_hat(2);

Vhy_c_list(call_count) = Vhy_c;
Vhx_c_list(call_count) = Vhx_c;
Vhy_hat_list(call_count) = Vhy_hat;
Vhy_hat_list(call_count) = Vhy_hat;

if(firstTime)
     % Initialize integrators
     PIR_vhorz_hold_x(0,0,0,firstTime,P);
     PIR_vhorz_hold_y(0,0,0,firstTime,P);
     PIR_roll_hold(0,0,0,firstTime,P);
     PIR_pitch_hold(0,0,0,firstTime,P);
     PIR_alt_hold(0,0,0,firstTime,P);
     PIR_yaw_hold(0,0,0,firstTime,P);
end

% Controls
theta_c = PIR_vhorz_hold_x(Vhx_c, Vhx_hat, 0, firstTime, P);
phi_c = PIR_vhorz_hold_y(Vhy_c, Vhy_hat, 0, firstTime, P);
delta_t = PIR_alt_hold(h_c, h_hat, hdot_hat, firstTime, P);
```

```
    delta_a = PIR_roll_hold(phi_c, phi_hat, p_hat, firstTime, P);
    delta_e = PIR_pitch_hold(theta_c, theta_hat, q_hat, firstTime, P);
    delta_r = PIR_yaw_hold(psi_c, psi_hat, r_hat, firstTime, P);

    % Compile vector of control surface deflections
    delta = [ ...
            delta_e; ...
            delta_a; ...
            delta_r; ...
            delta_t; ...
        ];

    % Override control delta with manual flight delta
    if P.manual_flight_flag
        error('Manual flight not supported in quadsim')
    end

    % Compile autopilot commands for logging/vis
    ap_command = [ ...
            Vhorz_c; ...
            h_c; ...
            chi_c; ...
            phi_c; ...
            theta_c;
            psi_c; ... % change from uavsim
            0; ... % future use
            0; ... % future use
            0; ... % future use
        ];

    call_count = call_count + 1;

    % Compile output vector
    out=[delta;ap_command]; % 4+9=13

end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% roll_hold
%   - regulate roll using aileron
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function u = PIR_roll_hold(phi_c, phi_hat, p_hat, init_flag, P)

    % Set up PI with rate feedback
    y_c = phi_c;
    y = phi_hat;
    y_dot = p_hat;

    kp = 0.11;
    ki = 0.0075;
    kd = 0.025;

    u_lower_limit = -0.1;
    u_upper_limit = +0.1;

    % Initialize integrator (e.g. when t==0)
    persistent error_int;
    if( init_flag )
```

```
    % Output saturation & integrator clamping
    %   - Limit u to u_upper_limit & u_lower_limit
    %   - Clamp if error is driving u past limit
    if u > u_upper_limit
        u = u_upper_limit;
        if ki*error>0
            error_int = error_int - P.Ts*error;
        end
    elseif u < u_lower_limit
        u = u_lower_limit;
        if ki*error<0
            error_int = error_int - P.Ts*error;
        end
    end

end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% pitch_hold
%   - regulate pitch using elevator
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function u = PIR_pitch_hold(theta_c, theta_hat, q_hat, init_flag, P)

    % Set up PI with rate feedback
    y_c = theta_c;
    y = theta_hat;
    y_dot = q_hat;

    kp = 0.11;
    ki = 0.0075;
    kd = 0.025;

    u_lower_limit = -0.1;
    u_upper_limit = 0.1;

    % Initialize integrator (e.g. when t==0)
    persistent error_int;
    if( init_flag )
        error_int = 0;
    end

    % Perform "PI with rate feedback"
    error = y_c - y;  % Error between command and response
    error_int = error_int + P.Ts*error; % Update integrator
    u = kp*error + ki*error_int - kd*y_dot;

    % Output saturation & integrator clamping
    %   - Limit u to u_upper_limit & u_lower_limit
    %   - Clamp if error is driving u past limit
    if u > u_upper_limit
        u = u_upper_limit;
        if ki*error>0
            error_int = error_int - P.Ts*error;
        end
    elseif u < u_lower_limit
        u = u_lower_limit;
        if ki*error<0
```

```
            error_int = error_int - P.Ts*error;
        end
    end

end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% yaw_hold
%   - regulate yaw using rudder
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function u = PIR_yaw_hold(psi_c, psi_hat, r_hat, init_flag, P)

    % Set up PI with rate feedback
    y_c = psi_c;
    y = psi_hat;
    y_dot = r_hat;

    kp = 0.11;
    ki = 0.002;
    kd = 0.07;

    u_lower_limit = -0.1;
    u_upper_limit = 0.1;

    % Initialize integrator (e.g. when t==0)
    persistent error_int;
    if( init_flag )
        error_int = 0;
    end

    % Perform "PI with rate feedback"
    % error = y_c - y;  % Error between command and response
    error = mod(y_c - y + pi, 2*pi)-pi;
    error_int = error_int + P.Ts*error; % Update integrator
    u = kp*error + ki*error_int - kd*y_dot;

    % Output saturation & integrator clamping
    %   - Limit u to u_upper_limit & u_lower_limit
    %   - Clamp if error is driving u past limit
    if u > u_upper_limit
        u = u_upper_limit;
        if ki*error>0
            error_int = error_int - P.Ts*error;
        end
    elseif u < u_lower_limit
        u = u_lower_limit;
        if ki*error<0
            error_int = error_int - P.Ts*error;
        end
    end

end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% horizontal velocity hold - x axis
%   - regulate horizontal velocity hold through pitch - x axis
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function u = PIR_vhorz_hold_x(Vhx_c, Vhx_hat, not_used, init_flag, P)
```

```
    % Set up PI with rate feedback
    y_c = Vhx_c;
    y = Vhx_hat;
    y_dot = 0;

    kp = -0.035;
    ki = -0.025;
    kd = -0.00001;

    u_lower_limit = -P.theta_max;
    u_upper_limit = +P.theta_max;

    % Initialize integrator (e.g. when t==0)
    persistent error_int;
    if( init_flag )
        error_int = 0;
    end

    % Perform "PI with rate feedback"
    error = y_c - y;  % Error between command and response
    error_int = error_int + P.Ts*error; % Update integrator
    u = kp*error + ki*error_int - kd*y_dot;

    % Output saturation & integrator clamping
    %   - Limit u to u_upper_limit & u_lower_limit
    %   - Clamp if error is driving u past limit
    if u > u_upper_limit
        u = u_upper_limit;
        if ki*error>0
            error_int = error_int - P.Ts*error;
        end
    elseif u < u_lower_limit
        u = u_lower_limit;
        if ki*error<0
            error_int = error_int - P.Ts*error;
        end
    end

end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% horizontal velocity hold - y axis
%   - regulate horizontal velocity hold through roll - y axis
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function u = PIR_vhorz_hold_y(Vhy_c, Vhy_hat, not_used, init_flag, P)

    % Set up PI with rate feedback
    y_c = Vhy_c;
    y = Vhy_hat;
    y_dot = 0;

    kp = 0.0349;
    ki = 0.02;
    kd = 0.00001;

    u_lower_limit = -P.phi_max;
    u_upper_limit = +P.phi_max;
```

```
    % Initialize integrator (e.g. when t==0)
    persistent error_int;
    if( init_flag )
        error_int = 0;
    end

    % Perform "PI with rate feedback"
    error = y_c - y;  % Error between command and response
    error_int = error_int + P.Ts*error; % Update integrator
    u = kp*error + ki*error_int - kd*y_dot;

    % Output saturation & integrator clamping
    %   - Limit u to u_upper_limit & u_lower_limit
    %   - Clamp if error is driving u past limit
    if u > u_upper_limit
        u = u_upper_limit;
        if ki*error>0
            error_int = error_int - P.Ts*error;
        end
    elseif u < u_lower_limit
        u = u_lower_limit;
        if ki*error<0
            error_int = error_int - P.Ts*error;
        end
    end

end
```

```
Not enough input arguments.

Error in quadsim_control (line 19)
    k=(1:4);         traj_cmds=uu(k); % Trajectory Commands
```

Published with MATLAB® R2023a