

Drone Control Final Update

Adin Sacho-Tanzer and Gabriel Kret

Introduction

- Aircraft dynamics simulators are helpful to test controllers without needing to run an expensive real-world test
- Our goal: simulate an Aerosonde UAV accurately, including PID control, sensors and a Kalman filter
- Then, implement the same simulation for a Cessna 172

Implementation

- Used code structure from Beard
 - Rigid body class implements rigid body EOMs
 - Aircraft class implements aerodynamics, thrust, gravity, and wind
 - Trim conditions computed using Jacobians and minimize function
 - Added PID control
 - Tuned PID
 - **Added sensors**
 - **Added Kalman filter**
 - **Changed all parameters to Cessna 172**
 - **Retuned PID for new plane**
 - **Retuned Kalman filter for new plane**
 - **Added automated takeoff, cruise, turn, descend**

EOMs

$$\begin{pmatrix} \dot{p}_n \\ \dot{p}_e \\ \dot{p}_d \end{pmatrix} = \begin{pmatrix} e_1^2 + e_0^2 - e_2^2 - e_3^2 & 2(e_1e_2 - e_3e_0) & 2(e_1e_3 + e_2e_0) \\ 2(e_1e_2 + e_3e_0) & e_2^2 + e_0^2 - e_1^2 - e_3^2 & 2(e_2e_3 - e_1e_0) \\ 2(e_1e_3 - e_2e_0) & 2(e_2e_3 + e_1e_0) & e_3^2 + e_0^2 - e_1^2 - e_2^2 \end{pmatrix} \begin{pmatrix} u \\ v \\ w \end{pmatrix}$$

$$\begin{pmatrix} \dot{u} \\ \dot{v} \\ \dot{w} \end{pmatrix} = \begin{pmatrix} rv - qw \\ pw - ru \\ qu - pv \end{pmatrix} + \frac{1}{m} \begin{pmatrix} f_x \\ f_y \\ f_z \end{pmatrix}$$

$$\begin{pmatrix} \dot{e}_0 \\ \dot{e}_1 \\ \dot{e}_2 \\ \dot{e}_3 \end{pmatrix} = \frac{1}{2} \begin{pmatrix} 0 & -p & -q & -r \\ p & 0 & r & -q \\ q & -r & 0 & p \\ r & q & -p & 0 \end{pmatrix} \begin{pmatrix} e_0 \\ e_1 \\ e_2 \\ e_3 \end{pmatrix}$$

$$\begin{pmatrix} \dot{p} \\ \dot{q} \\ \dot{r} \end{pmatrix} = \begin{pmatrix} \Gamma_1 pq - \Gamma_2 qr \\ \Gamma_5 pr - \Gamma_6(p^2 - r^2) \\ \Gamma_7 pq - \Gamma_1 qr \end{pmatrix} + \begin{pmatrix} \Gamma_3 l + \Gamma_4 n \\ \frac{1}{J_y} m \\ \Gamma_4 l + \Gamma_8 n \end{pmatrix}$$

Linear Aerodynamic Model

$$F_{\text{lift}} = \frac{1}{2} \rho V_a^2 S \left[C_{L_0} + C_{L_\alpha} \alpha + C_{L_q} \frac{c}{2V_a} q + C_{L_{\delta_e}} \delta_e \right]$$

$$F_{\text{drag}} = \frac{1}{2} \rho V_a^2 S \left[C_{D_0} + C_{D_\alpha} \alpha + C_{D_q} \frac{c}{2V_a} q + C_{D_{\delta_e}} \delta_e \right]$$

$$m = \frac{1}{2} \rho V_a^2 S c \left[C_{m_0} + C_{m_\alpha} \alpha + C_{m_q} \frac{c}{2V_a} q + C_{m_{\delta_e}} \delta_e \right]$$

$$f_y \approx \frac{1}{2} \rho V_a^2 S \left[C_{Y_0} + C_{Y_\beta} \beta + C_{Y_p} \frac{b}{2V_a} p + C_{Y_r} \frac{b}{2V_a} r + C_{Y_{\delta_a}} \delta_a + C_{Y_{\delta_r}} \delta_r \right]$$

$$l \approx \frac{1}{2} \rho V_a^2 S b \left[C_{l_0} + C_{l_\beta} \beta + C_{l_p} \frac{b}{2V_a} p + C_{l_r} \frac{b}{2V_a} r + C_{l_{\delta_a}} \delta_a + C_{l_{\delta_r}} \delta_r \right]$$

$$n \approx \frac{1}{2} \rho V_a^2 S b \left[C_{n_0} + C_{n_\beta} \beta + C_{n_p} \frac{b}{2V_a} p + C_{n_r} \frac{b}{2V_a} r + C_{n_{\delta_a}} \delta_a + C_{n_{\delta_r}} \delta_r \right]$$

Wind

$$\mathbf{V}_w^b = \begin{pmatrix} u_w \\ v_w \\ w_w \end{pmatrix} = \mathcal{R}_v^b(\phi, \theta, \psi) \begin{pmatrix} w_{n_s} \\ w_{e_s} \\ w_{d_s} \end{pmatrix} + \begin{pmatrix} u_{w_g} \\ v_{w_g} \\ w_{w_g} \end{pmatrix}$$

$$\mathbf{V}_a^b = \begin{pmatrix} u_r \\ v_r \\ w_r \end{pmatrix} = \begin{pmatrix} u - u_w \\ v - v_w \\ w - w_w \end{pmatrix}$$

$$V_a = \sqrt{u_r^2 + v_r^2 + w_r^2}$$

$$\alpha = \tan^{-1} \left(\frac{w_r}{u_r} \right)$$


$$\beta = \sin^{-1} \left(\frac{v_r}{\sqrt{u_r^2 + v_r^2 + w_r^2}} \right)$$

Trim Conditions

Objective is to compute trim states and inputs when aircraft simultaneously satisfies three conditions:

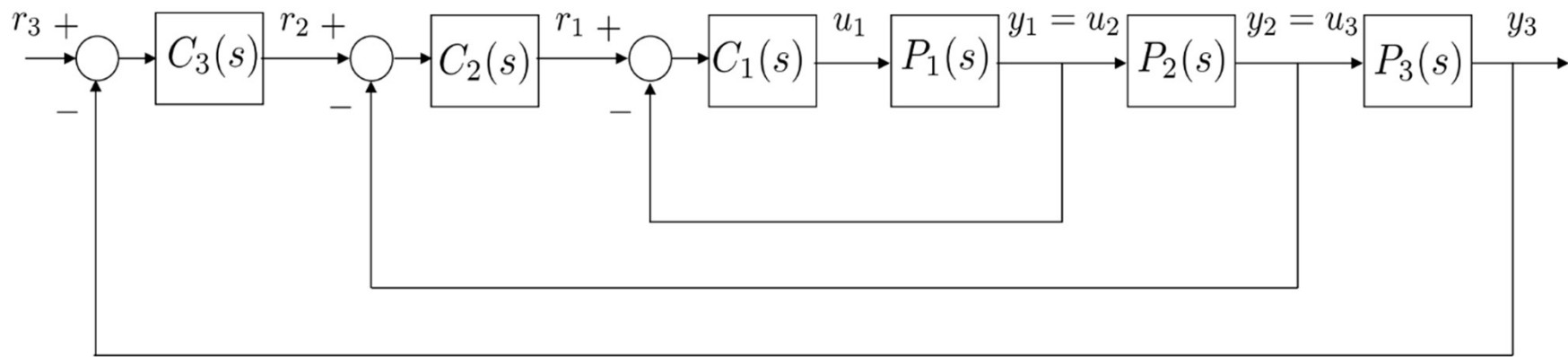
- Traveling at constant speed V_a^*
- Climbing at constant flight path angle γ^*
- In constant orbit of radius R^*

Goal: Converge to the trim objective function via minimization and enforcement of the trim constraints.

$$\dot{x}^* = \begin{pmatrix} \dot{p}_n^* \\ \dot{p}_e^* \\ \dot{h}^* \\ \dot{u}^* \\ \dot{v}^* \\ \dot{w}^* \\ \dot{\phi}^* \\ \dot{\theta}^* \\ \dot{\psi}^* \\ \dot{p}^* \\ \dot{q}^* \\ \dot{r}^* \end{pmatrix} = \begin{pmatrix} [\text{don't care}] \\ [\text{don't care}] \\ V_a^* \sin \gamma^* \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ \frac{V_a^*}{R^*} \cos \gamma^* \\ 0 \\ 0 \\ 0 \end{pmatrix} = f(x^*, u^*)$$


NOTE: when evaluating our state derivatives at trim conditions we expect all relevant results to be zero (see next slide). Our implementation achieved this except for q_{dot} which was non-zero.

Implementing PID w/ Successive Loop Closure

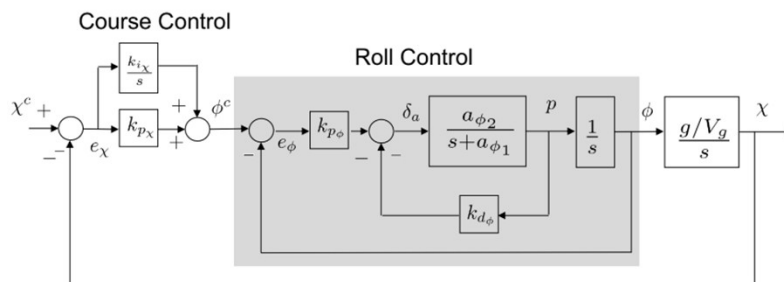


- SLC is a step-by-step method where you tune and close one control loop at a time, starting from the innermost loop, then moving outward, to keep the system stable and easier to control.
- The inner loops then act as a DC gain on the outer loops.
- The above loop is generic. We implement the above controller in the Autopilot class for both lateral and longitudinal control.
- We can then tune the gains appropriately to achieve satisfactory flight dynamics.

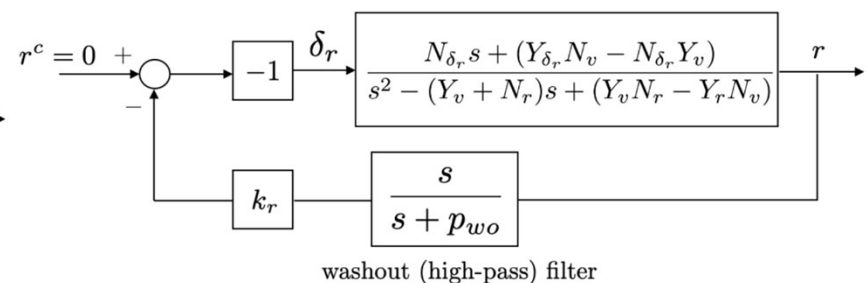
Implementing PID

- Implemented the autopilot class which holds the “components” capable of control (Aileron, Yaw Damper, Throttle, etc.)
- The autopilot class applies the appropriate controller to the component.
- For instance, Throttle is controlled with PI while Roll is controlled with PD (+rate control).
- Each of the controllers (PD, PI, PID, TF) is implemented with their unique control law called to each parameter in the Autopilot class.

Lateral-directional Autopilot



Yaw Damper

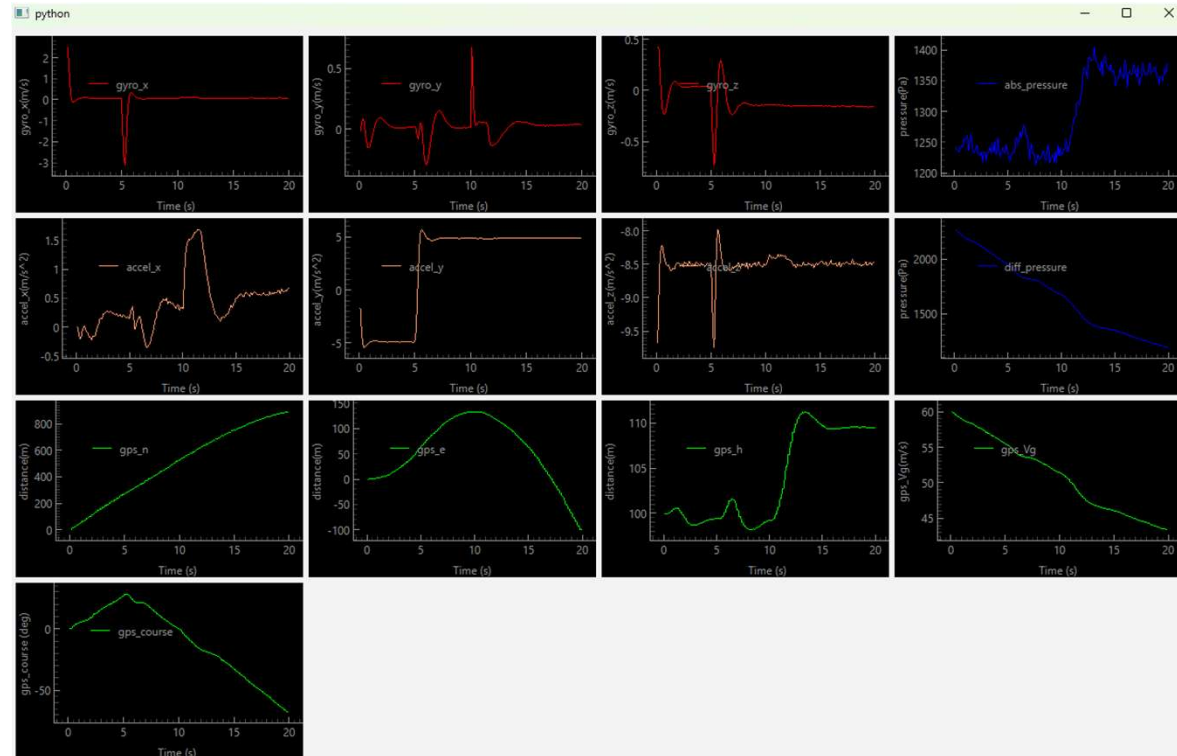


washout (high-pass) filter

```
autopilot.py  
pd_control_with_rate.py  
pi_control.py  
pid_control.py  
tf_control.py
```

Sensors

- We simulated:
 - Gyroscopes
 - Accelerometers
 - Magnetometers
 - Pressure sensors
 - GPS
- Given the state, we simulate what those sensors would output and add noise
- Sensor outputs are read by Kalman filter to provide estimate of state



Kalman Filter

- Kalman filter estimates state based on sensor measurements, model propagation
 - Model differential equations for how state changes over time
 - Model equations for what sensors should output given aircraft state
- When a new sensor measurement comes in, update the estimated state given the measurement, taking into account the expected variance of the sensors.

Final Project: Cessna 172

Changes necessary for Cessna

- Change stability derivatives
- Change engine model
- Retune PID
- Retune Kalman filter



Image source:

https://upload.wikimedia.org/wikipedia/commons/thumb/a/ae/Cessna_172S_Skyhawk_SP%2C_Private_JP6817606.jpg/960px-Cessna_172S_Skyhawk_SP%2C_Private_JP6817606.jpg

Stability derivatives + physical parameters

- Found source:
 - Kasnakoğlu, Coşku. (2016). Investigation of Multi-Input Multi-Output Robust Control Methods to Handle Parametric Uncertainties in Autopilot Design. PLOS ONE. 11. e0165017. 10.1371/journal.pone.0165017.
- Just requires changing all aircraft parameters in `aerosonde_parameters.py`

Engine model

- Engine was previously modeled as DC electric motor with formulas to calculate force and torque from RPM, which depends on windspeed, depends on voltage etc...
 - Complicated analysis specific to DC motor
- We found a source for a basic engine model: Lutze, F. H. (n.d.). *Thrust models* [Supplemental course notes]. Virginia Tech Department of Aerospace and Ocean Engineering.
<https://archive.aoe.vt.edu/lutze/AOE3104/thrustmodels.pdf>

Engine Model (cont.)

- Model:

$$T = SP \left(A_p \frac{\rho}{\rho_{SL}} - B_p \right) \frac{\eta_p}{V_\infty}$$

- Where:

- T is thrust
- SP is shaft power
- ρ is air density at altitude
- ρ_{SL} is air density at sea level
- η_p is propellor efficiency
- V_∞ is airspeed
- A_p is a constant equal to 1.132
- B_p is a constant equal to 0.132

Engine Model (cont.)

- We took:
 - $SP = \max(\delta_t * 134 \text{ kW}, 0.05 * 134 \text{ kW})$
 - 134 kW is max engine power for this model of Cessna (https://en.wikipedia.org/wiki/Cessna_172)
 - 0.05 is to ensure we always have some engine power
 - $\rho_{SL} = \rho$ because our simulation generally assumes constant density
 - $\eta_p = 0.8$, because it's apparently a typical value (<https://web.mit.edu/16.unified/www/FALL/thermodynamics/notes/node86.html>)
- We also assumed torque from propeller = 0
 - This is only a basic model
 - Primary purpose of engine is thrust, not torque, so this is probably minor

Engine Model (cont.)

- In addition to T , we also need $\frac{\partial T}{\partial V_a}$ and $\frac{\partial T}{\partial \delta_t}$ for calculating the transfer functions necessary to tune the PID control
- These are easy to calculate using our new model:
 - $\frac{\partial T}{\partial V_a} = (-134 \text{ kW})\delta_t(A_p - B_p)\frac{\eta_p}{V_a^2}$
 - $\frac{\partial T}{\partial \delta_t} = (134 \text{ kW})(A_p - B_p)\frac{\eta_p}{V_a}$

Trim Condition

- Started at $V_a = 62.8$ m/s based on research on Cessna 172 cruising speed
- Started at $\gamma = 0$
- Successfully computed a valid trim state

```
elevator= -0.004330320590381889  
aileron= -9.943101234202169e-09  
rudder= -1.970739043085918e-08  
throttle= 0.6953171466096343
```

Trim State:

```
[[ 2.21483670e-16]  
 [ 1.38603097e-15]  
 [-4.55482139e-15]  
 [ 6.27964546e+01]  
 [ 0.00000000e+00]  
 [-6.67304267e-01]  
 [ 9.99985886e-01]  
 [ 0.00000000e+00]  
 [-5.31301475e-03]  
 [ 0.00000000e+00]  
 [ 0.00000000e+00]  
 [ 0.00000000e+00]  
 [ 0.00000000e+00]  
 [ 0.00000000e+00]]
```

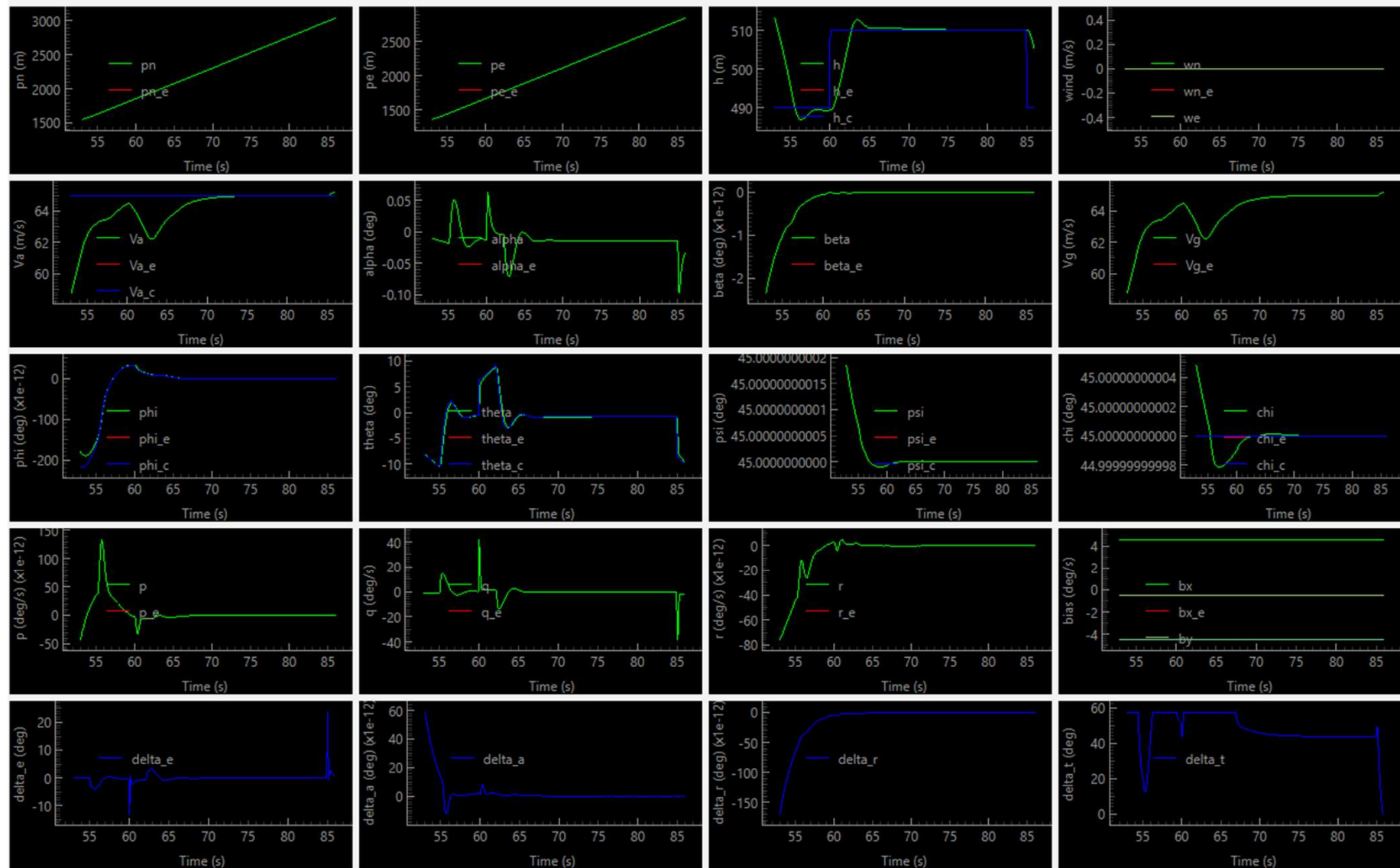
Trimmed State derivative:

```
[[ 6.28000000e+01]  
 [ 0.00000000e+00]  
 [ 9.67089567e-07]  
 [ 2.07714637e-01]  
 [-1.30346739e-07]  
 [-1.08925258e+00]  
 [ 0.00000000e+00]  
 [ 0.00000000e+00]  
 [ 0.00000000e+00]  
 [ 0.00000000e+00]  
 [ 0.00000000e+00]  
 [ 4.63706441e-07]  
 [-2.85605721e-01]  
 [ 2.75054945e-07]]
```

PID Tuning

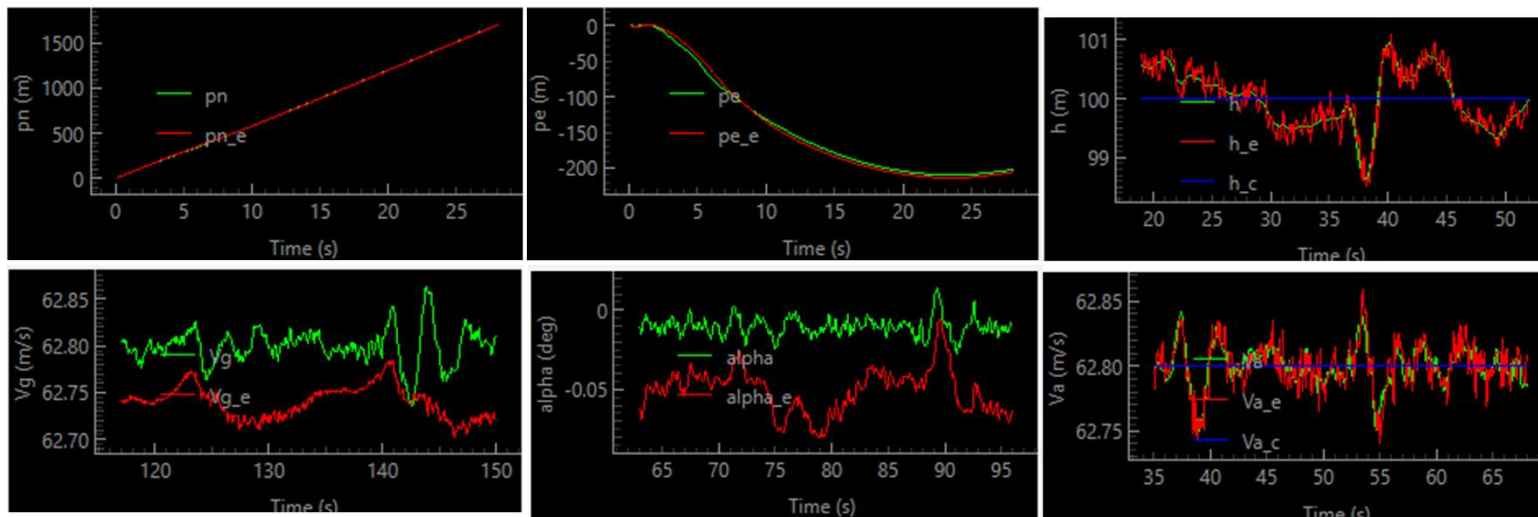
- Once we found the new parameters and implemented the transfer function equations to allow us to calculate PID gains from ζ and ω , our already-tuned PID worked well
- Changes we made:
 - Increased damping on altitude control to prevent overshoot
 - Decreased damping on course angle to allow for faster turns
 - Ended up dynamically changing this- more on this later!

python



Kalman filter tuning

- We had some bugs in our original Kalman filter which we ironed out.
 - Found a mistake in Beard's code! A vector was in the wrong order
- Once these were fixed, Kalman filter worked well



Realistic test scenario

- We also implemented routines to automatically test our autopilot and observer in a common scenario- takeoff, cruise, turning, and landing.
- This tests how our autopilot performs while transitioning between commands.

Takeoff

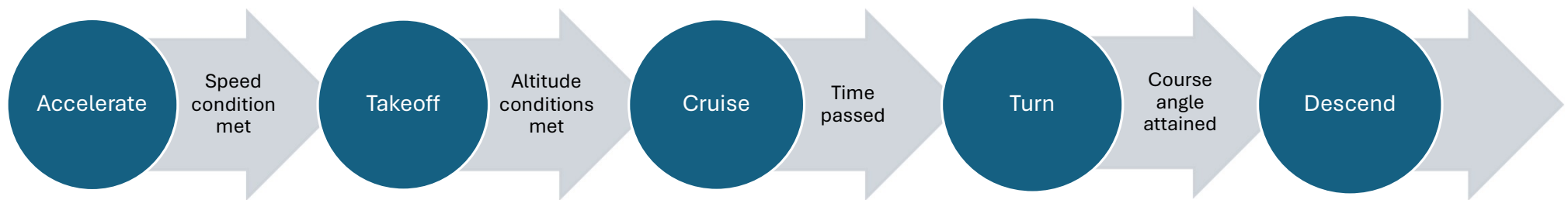
- Implemented simple ground interaction:

```
if mav.true_state.altitude <= 0.0:  
    mav.true_state.altitude = 0.01  
    mav._state[2] = 0.01
```

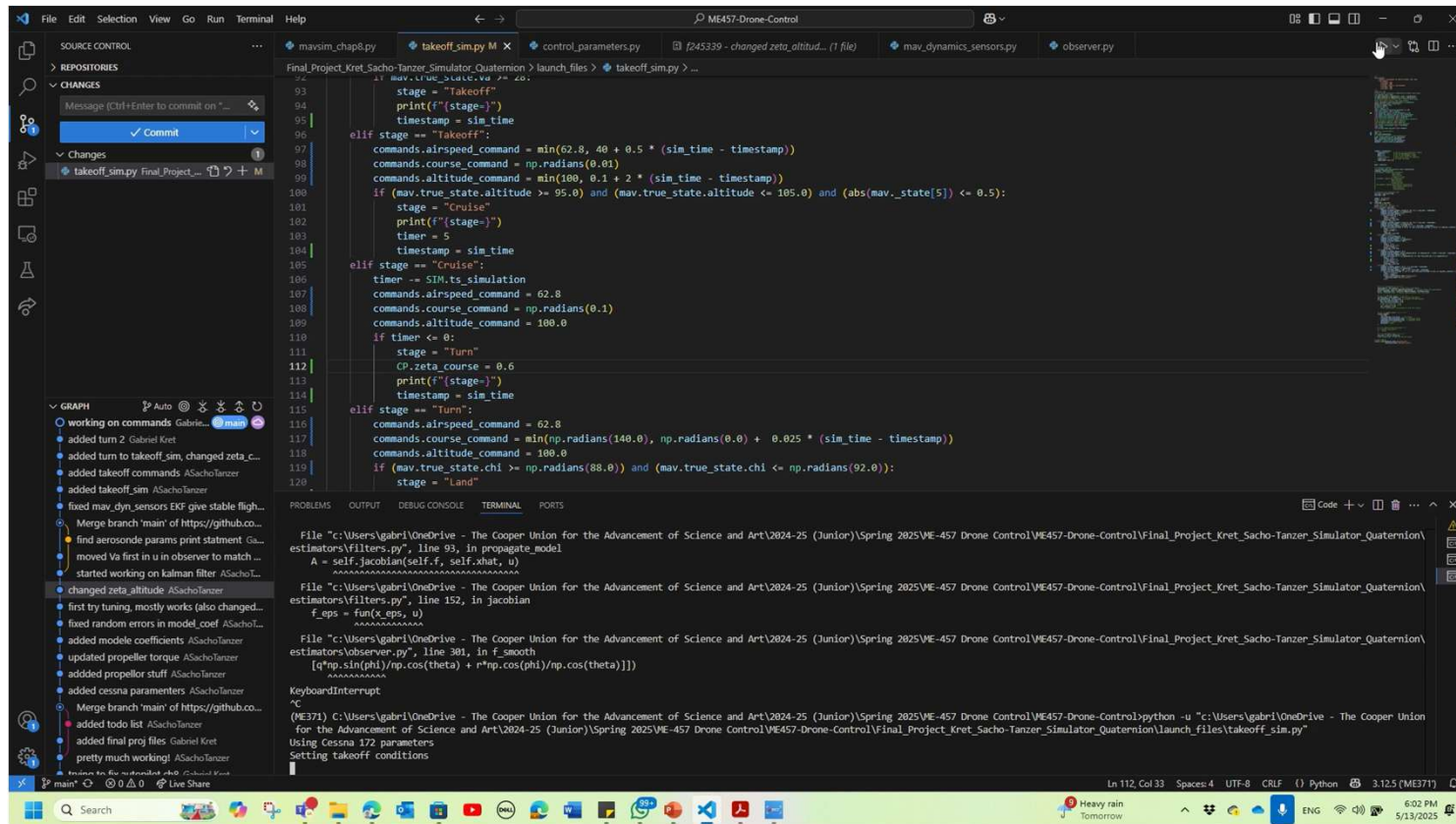
- This is not realistic:
 - Doesn't consider forces
 - No friction
 - Doesn't prevent tipping
- Good enough for our purposes

Autopilot structure

- Very basic “state machine”
- Change commanded V_a , h , χ based on state
- Dynamic change of PID damping based on the needs of the commands requested of the autopilot. This let's us control certain parameters more deliberately depending on the given command.



Autopilot demonstrations



The image shows a screenshot of a Visual Studio Code editor window. The main editor displays a Python file named `takeoff_sim.py` with the following code:

```
92. 11 mav.set_sim_state = 0.0
93. stage = "Takeoff"
94. print(f"stage:")
95. timestamp = sim_time
96.
97. elif stage == "Takeoff":
98.     commands.airspeed_command = min(62.8, 40 + 0.5 * (sim_time - timestamp))
99.     commands.course_command = np.radians(0.01)
100.    commands.altitude_command = min(100, 0.1 + 2 * (sim_time - timestamp))
101.    if (mav.true_state.altitude >= 95.0) and (mav.true_state.altitude <= 105.0) and (abs(mav.true_state[5]) <= 0.5):
102.        stage = "Cruise"
103.        print(f"stage:")
104.        timer = 5
105.        timestamp = sim_time
106.    elif stage == "Cruise":
107.        timer -= SIM.ts.simulation
108.        commands.airspeed_command = 62.8
109.        commands.course_command = np.radians(0.1)
110.        commands.altitude_command = 100.0
111.        if timer <= 0:
112.            stage = "Turn"
113.            CP.zeta_course = 0.6
114.            print(f"stage:")
115.            timestamp = sim_time
116.        elif stage == "Turn":
117.            commands.airspeed_command = 62.8
118.            commands.course_command = min(np.radians(140.0), np.radians(0.0) + 0.025 * (sim_time - timestamp))
119.            commands.altitude_command = 100.0
120.            if (mav.true_state.chi >= np.radians(88.0)) and (mav.true_state.chi <= np.radians(92.0)):
121.                stage = "Land"
```

The left sidebar shows the SOURCE CONTROL panel with a list of changes. The bottom panel shows the TERMINAL with the following output:

```
File "C:\Users\gabriel\OneDrive - The Cooper Union for the Advancement of Science and Art\2024-25 (Junior)\Spring 2025\VE-457 Drone Control\VE457-Drone-Control\Final_Project_Kret_Sacho-Tanzer_Simulator_Quaternion\
estimators\filters.py", line 93, in propagate_model
A = self.jacobian(self.f, self.xhat, u)
~~~~~
File "C:\Users\gabriel\OneDrive - The Cooper Union for the Advancement of Science and Art\2024-25 (Junior)\Spring 2025\VE-457 Drone Control\VE457-Drone-Control\Final_Project_Kret_Sacho-Tanzer_Simulator_Quaternion\
estimators\filters.py", line 152, in jacobian
f_eps = fun(x_eps, u)
~~~~~
File "C:\Users\gabriel\OneDrive - The Cooper Union for the Advancement of Science and Art\2024-25 (Junior)\Spring 2025\VE-457 Drone Control\VE457-Drone-Control\Final_Project_Kret_Sacho-Tanzer_Simulator_Quaternion\
estimators\observer.py", line 301, in f_smooth
[q*np.sin(phi)/np.cos(theta) + r*np.cos(phi)/np.cos(theta)]]
~~~~~
KeyboardInterrupt
^C
(ME371) C:\Users\gabriel\OneDrive - The Cooper Union for the Advancement of Science and Art\2024-25 (Junior)\Spring 2025\VE-457 Drone Control\VE457-Drone-Control\python -u "c:\Users\gabriel\OneDrive - The Cooper Union
for the Advancement of Science and Art\2024-25 (Junior)\Spring 2025\VE-457 Drone Control\VE457-Drone-Control\Final_Project_Kret_Sacho-Tanzer_Simulator_Quaternion\launch_files\takeoff_sim.py"
Using Cessna 172 parameters
Setting takeoff conditions
```

Future improvements

- Ground model is inaccurate
 - No friction, traction
 - No forces
- Aircraft model is incomplete
 - No wheels
 - No airbrakes/flaps
 - No aerodynamic ground effect
- Aerodynamic model is inaccurate at low speeds
 - Should be accurate from mid-acceleration to just before landing