ch04_forces_moments

February 3, 2022

0.1 Problem 0: Implementation of Forces and Moments

Implement the `forces_moments`, `motor_thrust_torque`, and `update_velocity_data` functions in `chap4/mav_dynamics.py` as described in chapter 4. For the aerodynamic coefficients $C_L(\alpha)$ and $C_D(\alpha)$, use equations (4.9) and (4.11), respectively.

Below are some useful code snippets and hints: * R = Quaternion2Rotation(self._state[IND.QUAT]) # computes the rotation from body to world frame * uvw = state[IND.VEL] # Extracts the u-v-w body axis velocities from the state vector * MAV contains the UAV parameters (i.e., MAV.mass gives the mass and MAV.gravity is the gravity constant) * In `forces_moments`, you will need to calculate the following: * gravitaional forces * Lift and Drag coefficients * Lift and Drag Forces * longitudinal forces and torques * lateral forces and torques * In `motor_thrust_torque` you will do the following * Use the quadratic formulate to solve for motor speed * Use the resulting values to compute the angular speed of the propeller * Compute and return the thruse and torque due to the propeller * In `update_velocity_data` you will do the following * Convert the wind vector to the body frame * Compute the airspeed * Compute the angle of attack * Compute the sideslip angle

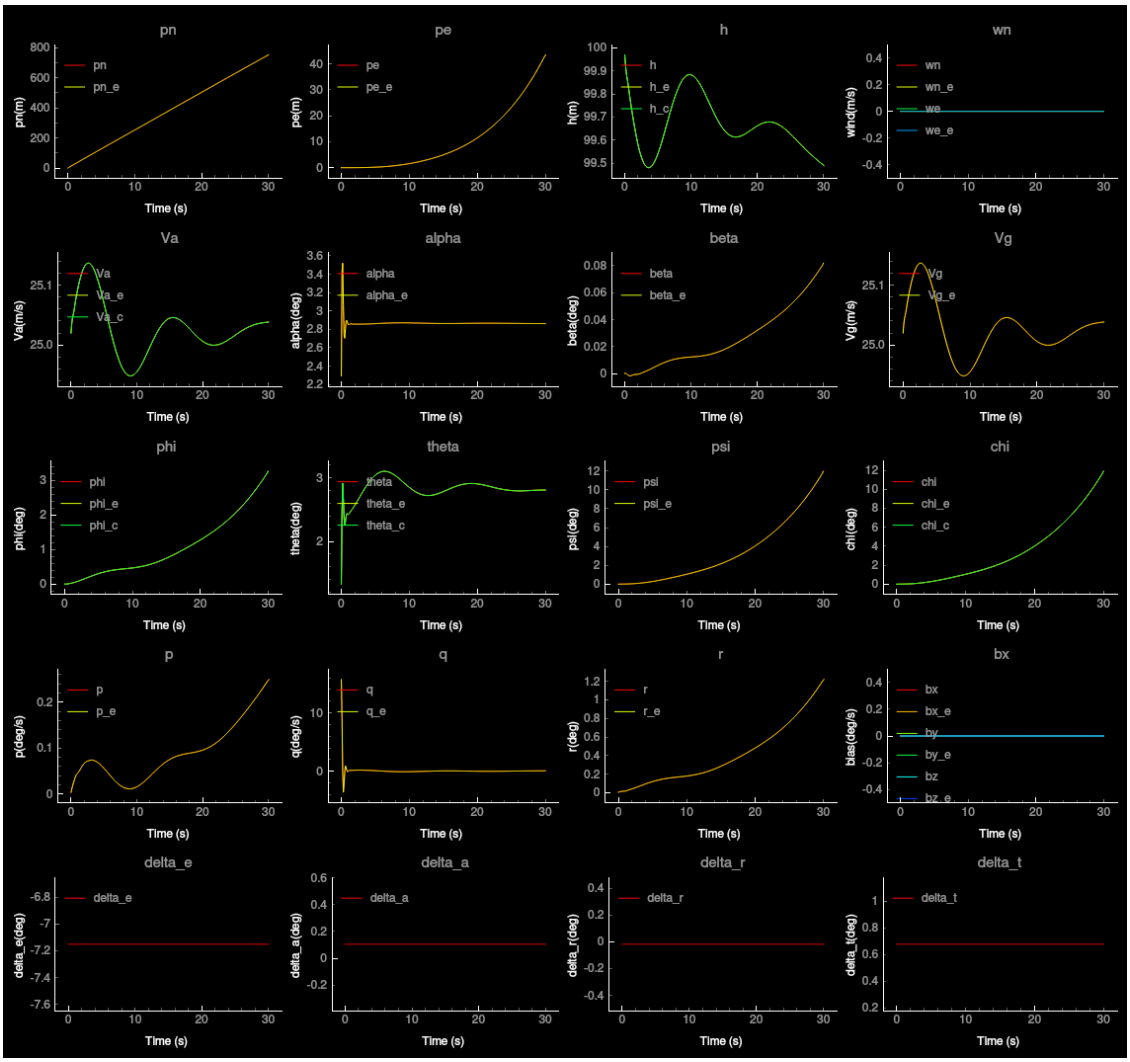
Running the simulator requires passing a function that takes in a time value and produces a command. We'll learn about trim later, but trim trajectories are trajectories that can be flown with constant inputs. In reality the trim inputs can only be used for a small amount of time without any feedback due to disturbances. Once implemented, the following code will produce a trajectory that flights straight and steady without any need for modification.

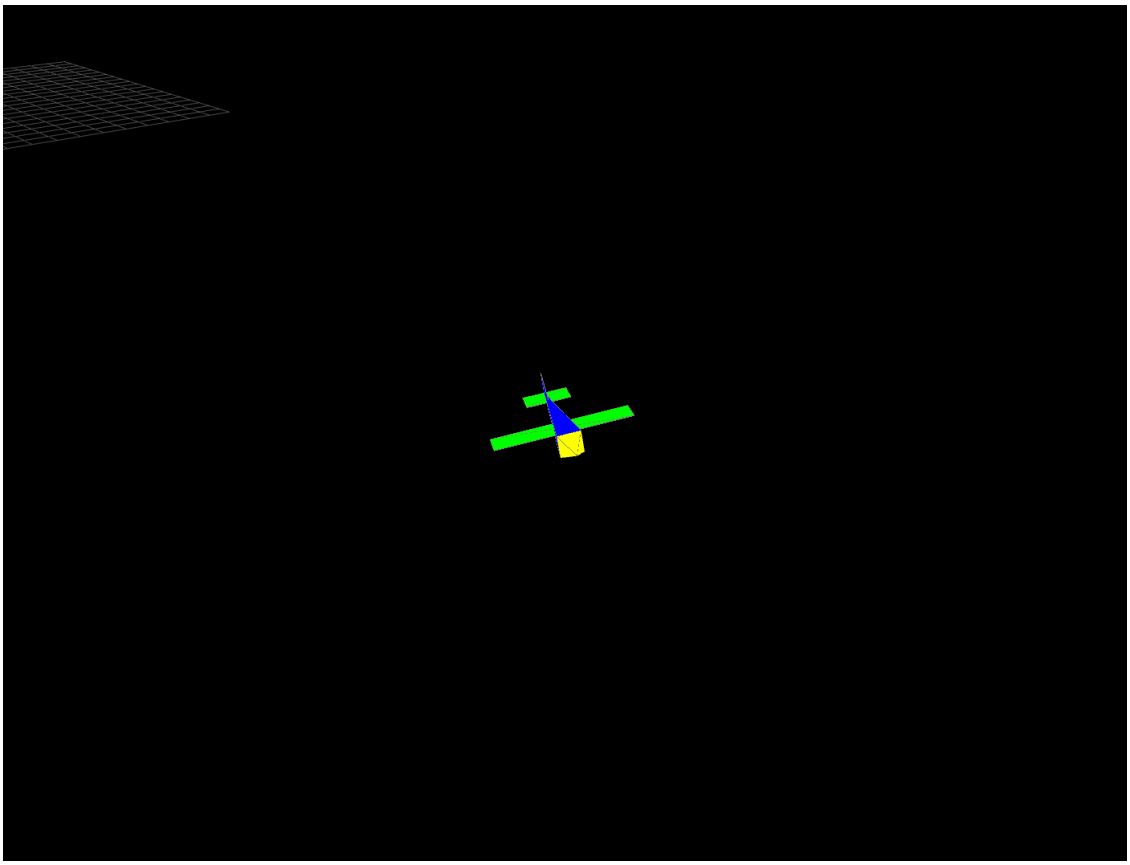
[1]:
```
# Note that this cell can be run separately to initialize for other cell blocks
import numpy as np
from mav_sim.chap3.mav_dynamics import DynamicState
from mav_sim.chap4.run_sim import run_sim
from mav_sim.message_types.msg_delta import MsgDelta
from mav_sim.message_types.msg_sim_params import MsgSimParams
from mav_sim.message_types.msg_gust_params import MsgGustParams
from mav_sim.tools.display_figures import display_data_view, display_mav_view
from mav_sim.chap2.mav_viewer import MavViewer
from mav_sim.chap3.data_viewer import DataViewer

# The viewers need to be initialized once due to restart issues with qtgraph
if 'mav_view' not in globals():
    print("Initializing mav_view")
    global mav_view
```

```
    mav_view = MavViewer()  # initialize the mav viewer
if 'data_view' not in globals():
    print("Initializing data_view")
    global data_view
    data_view = DataViewer()  # initialize view of data plots

# Initialize state values
sim_params = MsgSimParams(end_time=40., video_name="chap4.avi") # Sim ending in␣
 ↪10 seconds
state = DynamicState()

# Functions used below
def run_sim_and_display(delta_fnc, use_wind = False, gust_params = None):
    global mav_view
    global data_view
    data_view.reset(sim_params.start_time)
    (mav_view, data_view) = run_sim(sim_params, delta_fnc, state, mav_view,␣
 ↪data_view, use_wind, gust_params)
    display_data_view(data_view)
    display_mav_view(mav_view)

def trim(time: float)->MsgDelta:
    """Passes out the constant trim command
    """
    # Set control surfaces
    delta = MsgDelta()
    delta.elevator = -0.1248
    delta.aileron = 0.001836
    delta.rudder = -0.0003026
    delta.throttle = 0.6768
    return delta
```

```
Initializing mav_view
Initializing data_view
```

[2]:
```
# Run a "straight" trim trajectory
run_sim_and_display(trim)
```


pn
pn
pn_e
pn(m)
Time (s)
pe
pe
pe_e
pe(m)
Time (s)
h
h
h_e
h_c
h(m)
Time (s)
wn
wn
wn_e
we
we_e
wind(m/s)
Time (s)
Va
Va
Va_e
Va_c
Va(m/s)
Time (s)
alpha
alpha
alpha_e
alpha(deg)
Time (s)
beta
beta
beta_e
beta(deg)
Time (s)
Vg
Vg
Vg_e
Vg(m/s)
Time (s)
phi
phi
phi_e
phi_c
phi(deg)
Time (s)
theta
theta
theta_e
theta_c
theta(deg)
Time (s)
psi
psi
psi_e
psi(deg)
Time (s)
chi
chi
chi_e
chi_c
chi(deg)
Time (s)
p
p
p_e
p(deg/s)
Time (s)
q
q
q_e
q(deg/s)
Time (s)
r
r
r_e
r(deg)
Time (s)
bx
bx
bx_e
by
by_e
bz
bz_e
bias(deg/s)
Time (s)
delta_e
delta_e
delta_e(deg)
Time (s)
delta_a
delta_a
delta_a(deg)
Time (s)
delta_r
delta_r
delta_r(deg)
Time (s)
delta_t
delta_t
delta_t(deg)
Time (s)



0.2 Problem 1: Effect of the elevator control surface

This problem varies the elevator value during flight of the aircraft. Design a function that will perturb the elevator control surface by 5 degrees in the negative direction after 5 seconds and 5 degrees in the positive direction after 15 seconds.

0.2.1 Question: Given the definitions of positive control surface deflection from chapter 4, what behavior do you expect for the altitude, Y-axis angular velocity, airspeed, and pitch angle?

Answer: (Answer before simulation)

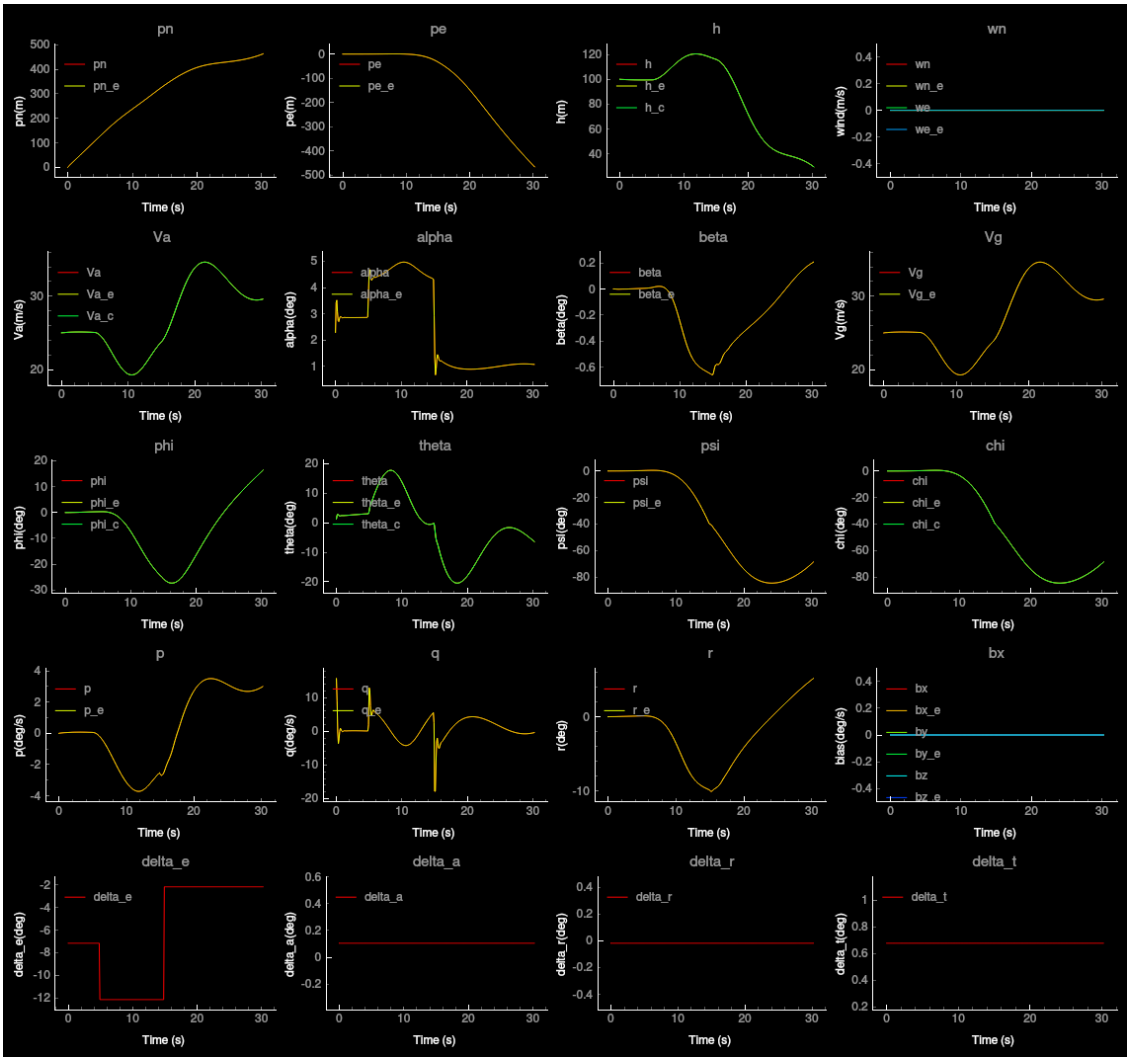
Just changing the pitch angle will cause the aircraft to slow down but increase in altitude. As it decrases it will increase in velocity

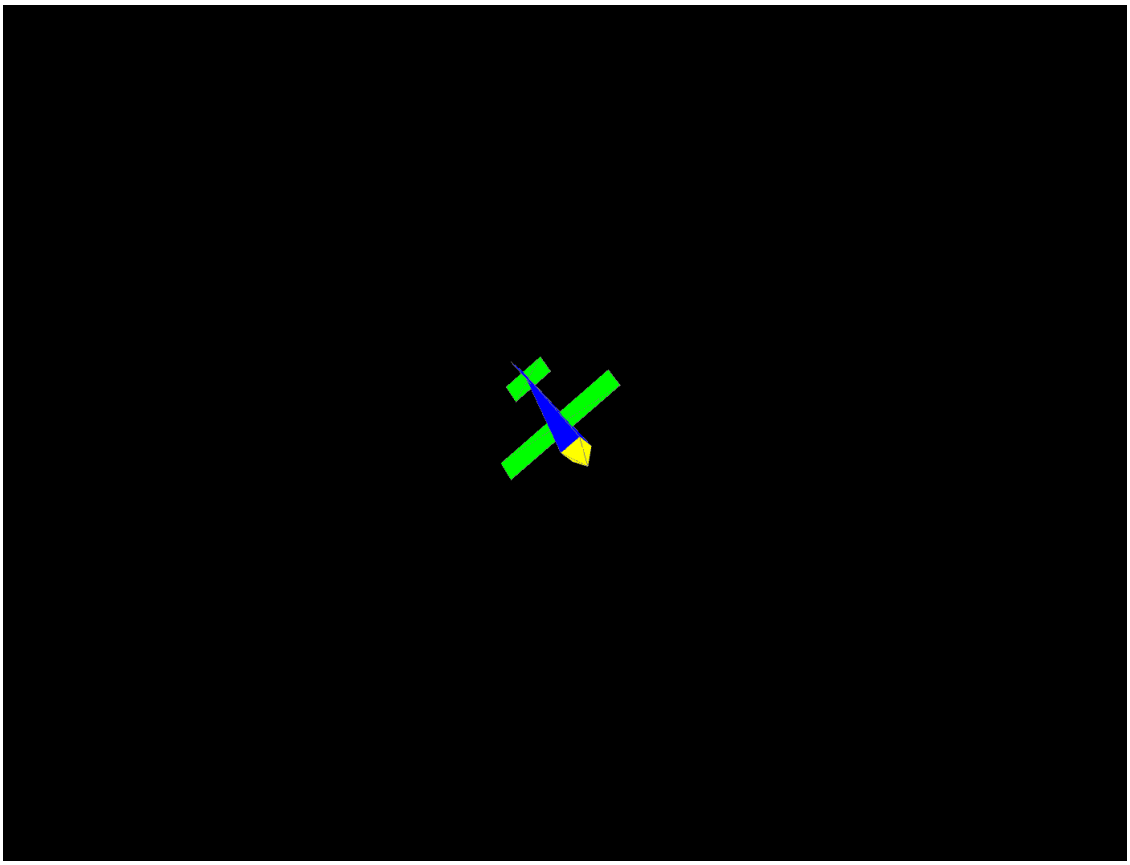
0.2.2 Question: Was the behavior as expected? If not, what did you observe differently from what you thought you would? Why did this difference occur?

Answer: (Up to individual)

[3]:
```
def perturb_elevator(time: float) -> MsgDelta:
    """ Perturb the elevator trim commands by
```

```
            -5 degrees      5 < time <= 15
            5  degrees      15 < time
    """
    delta = trim(time)
    if time > 15.:
        delta.elevator += np.deg2rad(5) # Perturb up by 5 degrees (make sure␣
 ↪you convert degrees to radians!!!)
    elif time > 5:
        delta.elevator += np.deg2rad(-5) # Perturb down by 5 degrees
    return delta

# Perturb the elevator
run_sim_and_display(perturb_elevator)
```





0.3 Problem 2: Effect of the aileron control surface

This problem varies the aileron value during flight of the aircraft. Design a function that will perturb the aileron control surface by 0.2 degrees in the negative direction after 5 seconds and 0.2 degrees in the positive direction after 15 seconds.

0.3.1 Question: Given the definitions of positive control surface deflection from chapter 4, what behavior do you expect for X-axis angular velocity, roll angle, and heading?

Answer: (Answer before simulation)

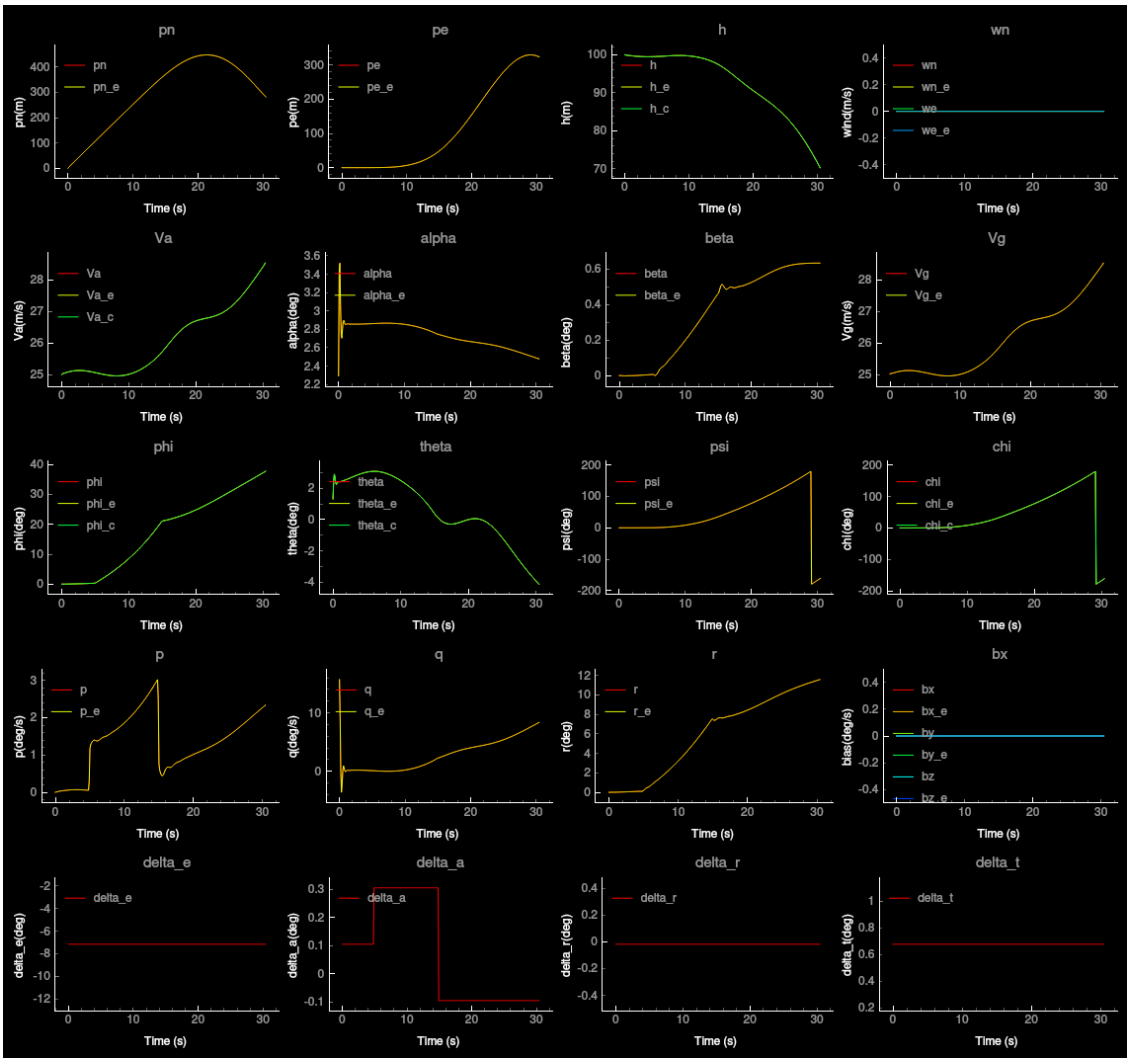
Pivot the plain about the body. It will turn left and right

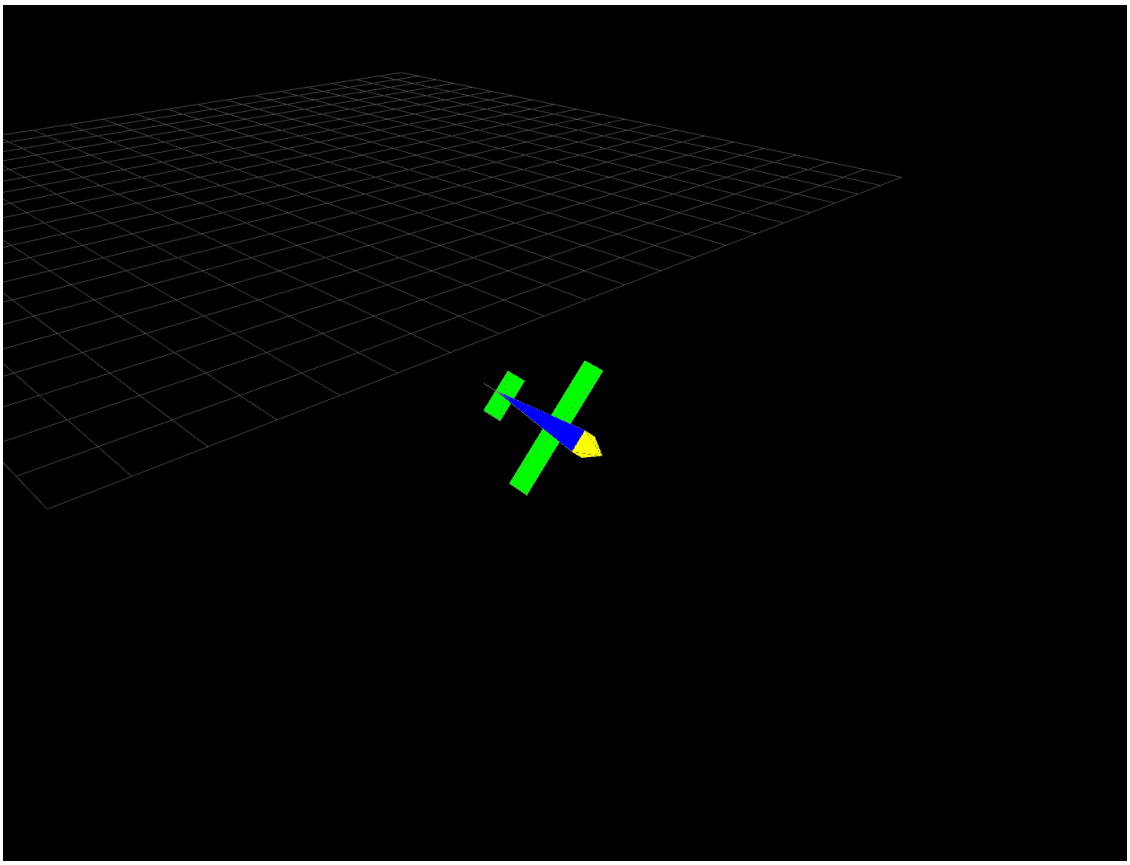
Run the simulation and verify the expected behavior.

0.3.2 Question: Was the behavior as expected? If not, what did you observe differently from what you thought you would? Why did this difference occur?

Answer: (Up to individual)

[4]:
```
def perturb_aileron(time: float) -> MsgDelta:
    """ Perturb the aileron trim commands by
            -0.2 degrees      5 < time <= 15
            0.2  degrees      15 < time
    """
    # Calculate the trim trajectory
    delta = trim(time)

    # Perturb the control
    if time > 15.:
        delta.aileron += np.deg2rad(-0.2) # Perturb up by 5 degrees (make sure you convert degrees to radians!!!)
    elif time > 5:
        delta.aileron += np.deg2rad(0.2) # Perturb down by 5 degrees
    return delta
    # Return the perturbed control
    return delta

# Perturb the elevator
run_sim_and_display(perturb_aileron)
```


pn
pn
pn_e
pn(m)
Time (s)
pe
pe
pe_e
pe(m)
Time (s)
h
h
h_e
h_c
h(m)
Time (s)
wn
wn
wn_e
we
we_e
wind(m/s)
Time (s)
Va
Va
Va_e
Va_c
Va(m/s)
Time (s)
alpha
alpha
alpha_e
alpha(deg)
Time (s)
beta
beta
beta_e
beta(deg)
Time (s)
Vg
Vg
Vg_e
Vg(m/s)
Time (s)
phi
phi
phi_e
phi_c
phi(deg)
Time (s)
theta
theta
theta_e
theta_c
theta(deg)
Time (s)
psi
psi
psi_e
psi(deg)
Time (s)
chi
chi
chi_e
chi_c
chi(deg)
Time (s)
p
p
p_e
p(deg/s)
Time (s)
q
q
q_e
q(deg/s)
Time (s)
r
r
r_e
r(deg)
Time (s)
bx
bx
bx_e
by
by_e
bz
bz_e
bias(deg/s)
Time (s)
delta_e
delta_e
delta_e(deg)
Time (s)
delta_a
delta_a
delta_a(deg)
Time (s)
delta_r
delta_r
delta_r(deg)
Time (s)
delta_t
delta_t
delta_t(deg)
Time (s)



0.4 Problem 3: Effect of the rudder control surface

This problem varies the rudder value during flight of the aircraft. Design a function that will perturb the rudder control surface by 0.2 degrees in the negative direction after 5 seconds and 0.2 degrees in the positive direction after 15 seconds.

0.4.1 Question: Given the definitions of positive control surface deflection from chapter 4, what behavior do you expect for Z-axis angular velocity, heading angle, and roll angle?

Answer: (Answer before simulation) It will spin the plain about the body z axis

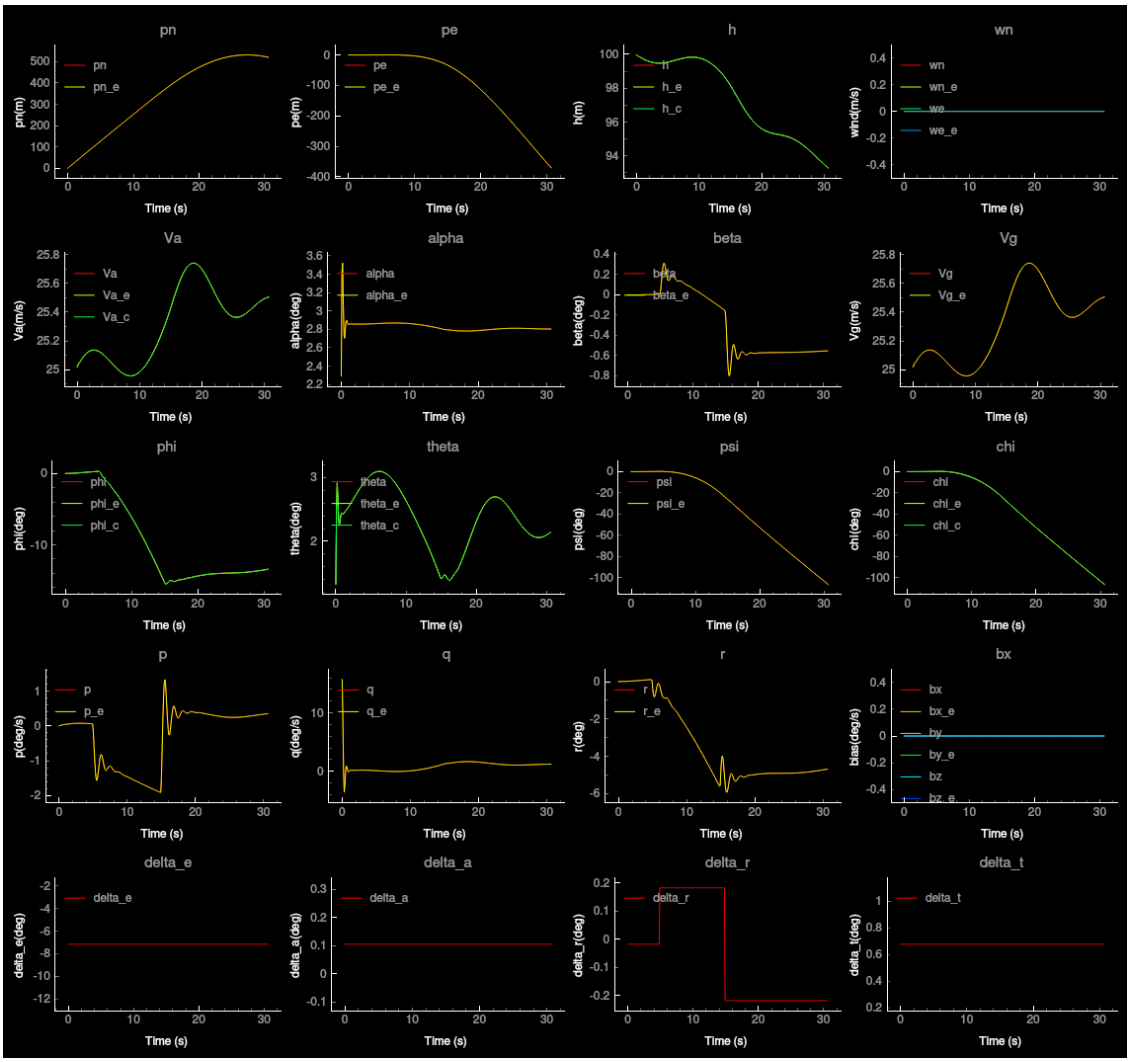
It will pivot the plain left and right

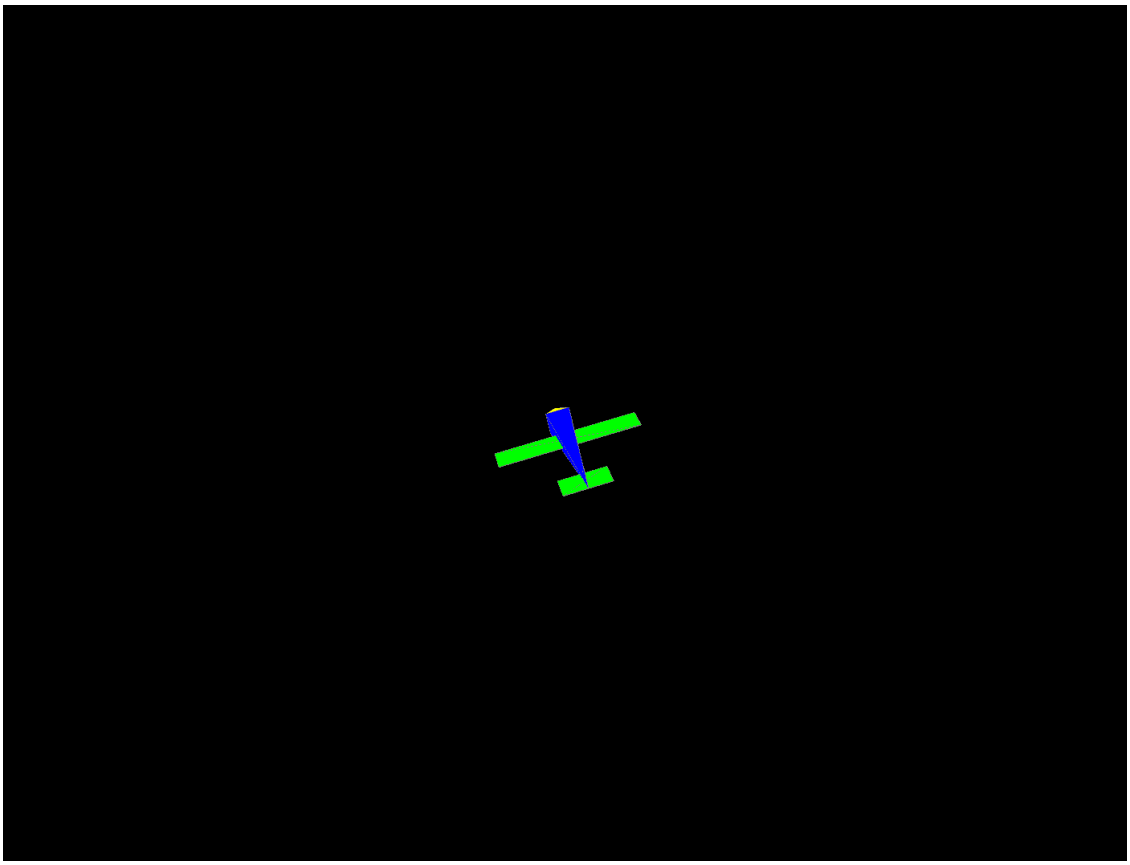
Run the simulation and verify the expected behavior.

0.4.2 Question: Was the behavior as expected? If not, what did you observe differently from what you thought you would? Why did this difference occur?

Answer: (Up to individual)

[5]:
```
def perturb_rudder(time: float) -> MsgDelta:
    """ Perturb the rudder trim commands by
            -0.2 degrees      5 < time <= 15
            0.2  degrees      5 < time
    """
    # Calculate the trim trajectory
    delta = trim(time)

    # Perturb the control
    if time > 15.:
        delta.rudder += np.deg2rad(-0.2) # Perturb up by 5 degrees (make sure␣
 ↪you convert degrees to radians!!!)
    elif time > 5:
        delta.rudder += np.deg2rad(0.2) # Perturb down by 5 degrees
    return delta

    # Return the perturbed control
    return delta

# Perturb the elevator
run_sim_and_display(perturb_rudder)
```


pn
pn
pn_e
pn(m)
Time (s)
pe
pe
pe_e
pe(m)
Time (s)
h
h
h_e
h_c
h(m)
Time (s)
wn
wn
wn_e
we
we_e
wind(m/s)
Time (s)
Va
Va
Va_e
Va_c
Va(m/s)
Time (s)
alpha
alpha
alpha_e
alpha(deg)
Time (s)
beta
beta
beta_e
beta(deg)
Time (s)
Vg
Vg
Vg_e
Vg(m/s)
Time (s)
phi
phi
phi_e
phi_c
phi(deg)
Time (s)
theta
theta
theta_e
theta_c
theta(deg)
Time (s)
psi
psi
psi_e
psi(deg)
Time (s)
chi
chi
chi_e
chi_c
chi(deg)
Time (s)
p
p
p_e
p(deg/s)
Time (s)
q
q
q_e
q(deg/s)
Time (s)
r
r
r_e
r(deg)
Time (s)
bx
bx
bx_e
by
by_e
bz
bz_e
bias(deg/s)
Time (s)
delta_e
delta_e
delta_e(deg)
Time (s)
delta_a
delta_a
delta_a(deg)
Time (s)
delta_r
delta_r
delta_r(deg)
Time (s)
delta_t
delta_t
delta_t(deg)
Time (s)



0.5 Problem 4: Effect of the throttle control input

This problem varies the throttle value during flight of the aircraft. Design a function that will perturb the throttle by 0.2 in the negative direction after 5 seconds and 0.2 in the positive direction after 15 seconds.

0.5.1 Question: Given the definitions of positive thrust from chapter 4, what behavior do you expect for airspeed, altitude, and roll rates?

Answer: (Answer before simulation)

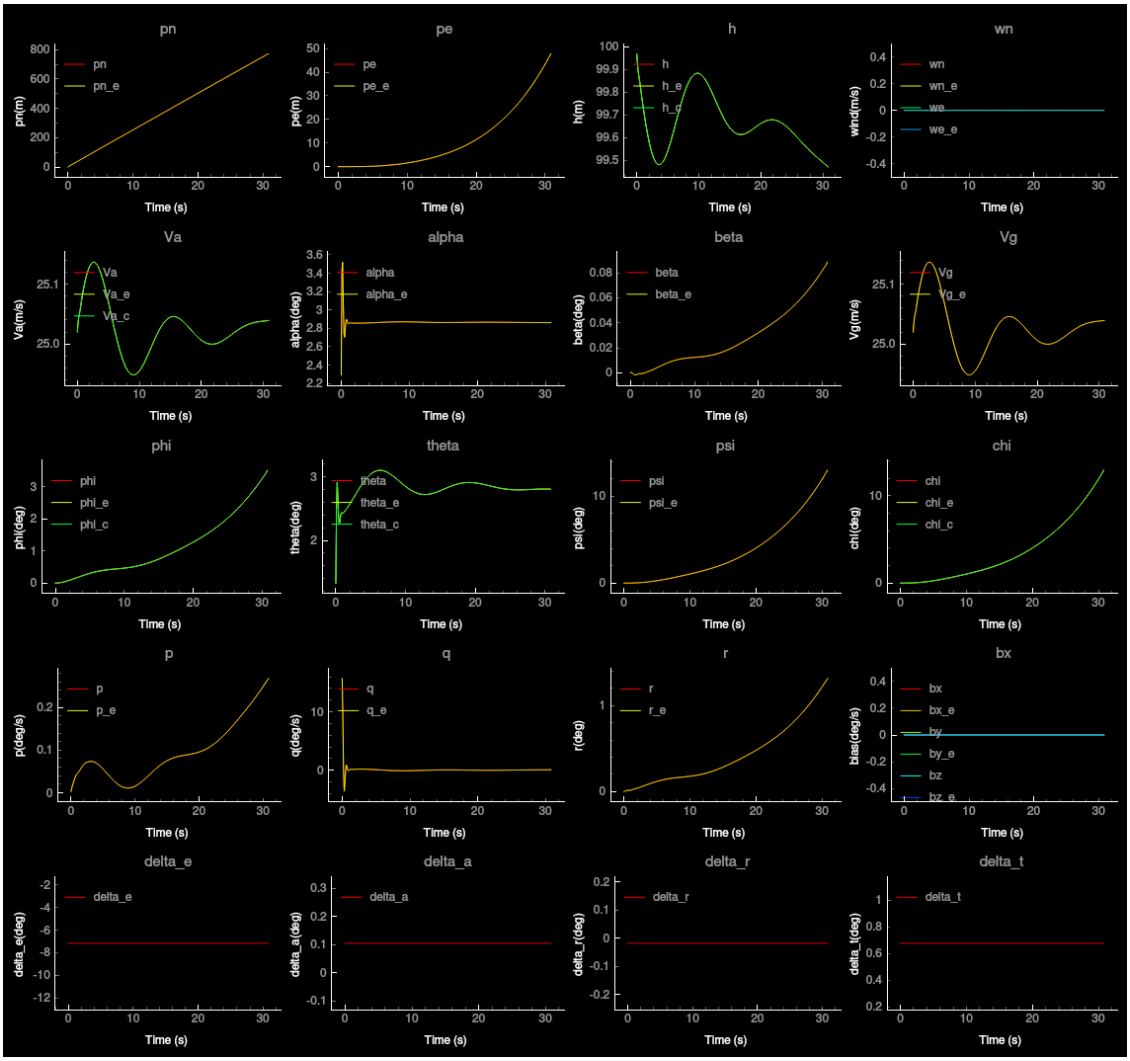
Run the simulation and verify the expected behavior.

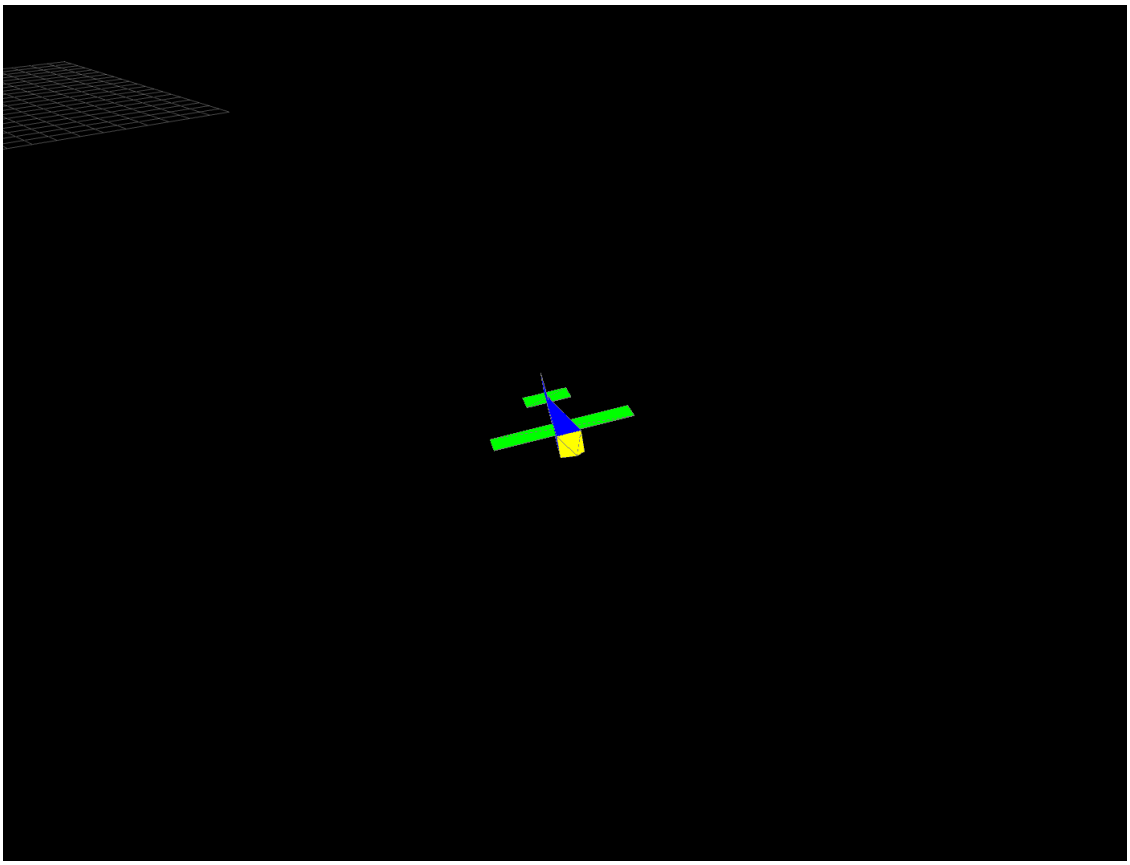
0.5.2 Question: Was the behavior as expected? If not, what did you observe differently from what you thought you would? Why did this difference occur?

Answer: (Up to individual)

[6]:
```
def perturb_throttle(time: float) -> MsgDelta:
    """ Perturb the throttle trim commands by
            -0.2        5 < time <= 15
            0.2         5 < time
```

```
    """
    # Calculate the trim trajectory
    delta = trim(time)

    # Perturb the control

    # Return the perturbed control
    return delta

# Perturb the elevator
run_sim_and_display(perturb_throttle)
```





0.6 Problem 5: Wind model implementation

The previous problems did not consider wind. You will now implement the Dryden gust model as defined in Chapter 4.4 of the book. Namely, you will implement the transfer functions in the `__init__` function of `wind_siumulation.py`.

For example, the parameter *self.u_w* is a transfer function of the form

$$H_u(s) = \sigma_u \sqrt{\frac{2V_a}{L_u}} \frac{1}{s + \frac{V_a}{L_u}}$$

This results in a transfer function with the numerator and denominator defined as

$$numerator = \sigma_u \sqrt{\frac{2V_a}{L_u}}$$

$$denominator = s + \frac{V_a}{L_u}$$

The numerators and denominators are passed into a `TransferFunction` class that is provided to you by specifying their coefficients. The above equation for $H_u(s)$ can be implemented as follows:

```
a1 = sigma_u*np.sqrt(2.*Va/Lu)
b1 = Va/Lu
self.u_w = TransferFunction(num=np.array([[a1]]),
                            den=np.array([[1, b1]]),
                            Ts=Ts)
```

In `wind_siumulation.py`, you will initialize the following * `self.u_w` is the transfer function $H_u(s)$ * `self.v_w` is the transfer function $H_v(s)$ * `self.w_w` is the transfer function $H_w(s)$

0.7 Problem 6: Using non-zero wind parameters

You will now run the simulation with a set of parameters for the Dresden gust model from Table 4.1 of the book. Use the medium altitude, moderate turbulence model parameters.

0.7.1 Question: What effects do you expect to see on the states for the medium altitude, moderate turbulence model as compared to the results of Problem 0?

Answer: (Answer before simulation)

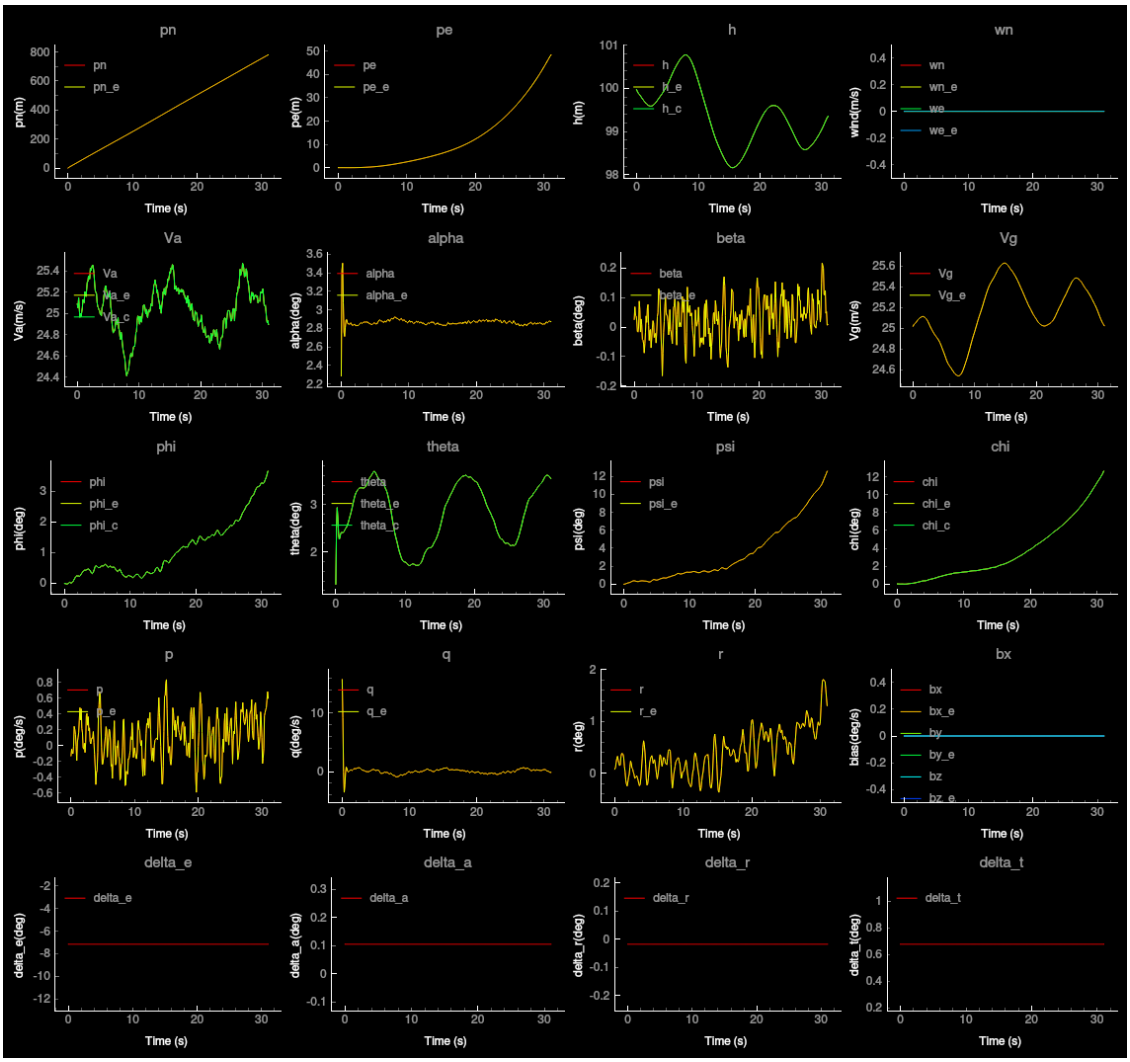
It will cause side slip and instability

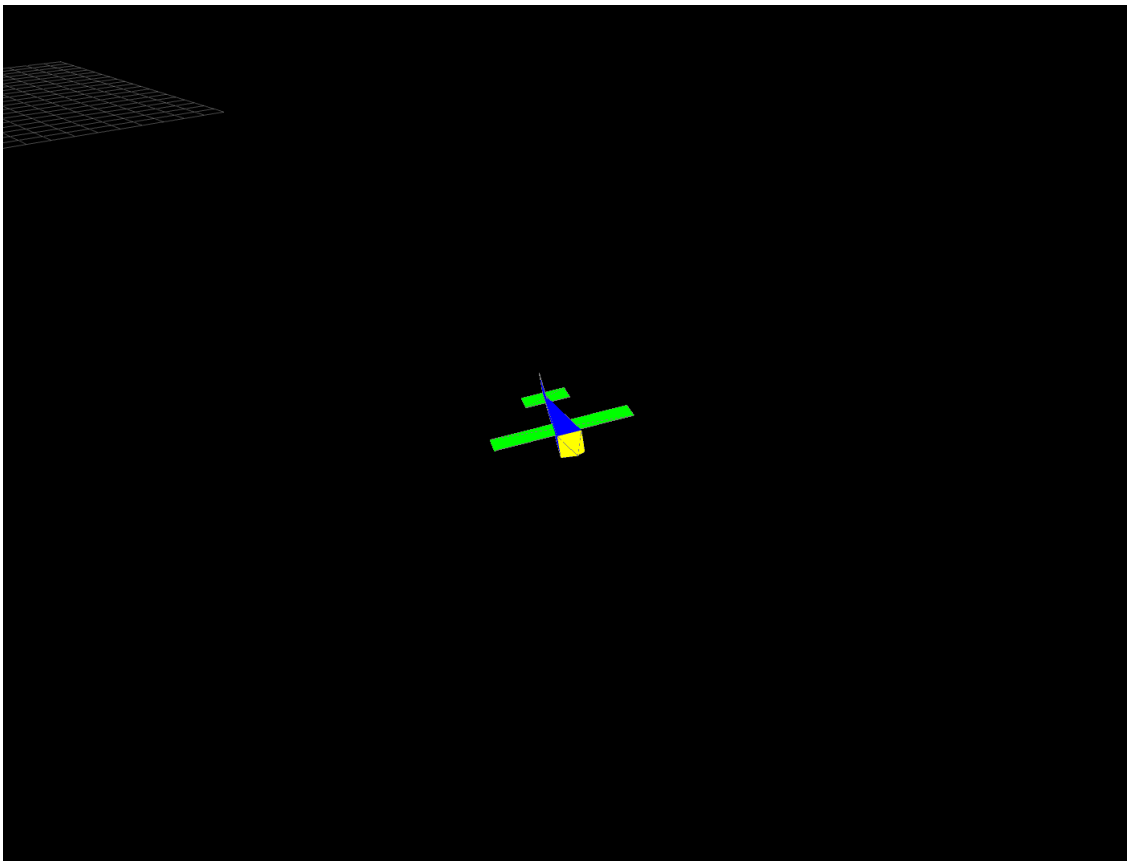
Run the simulation and verify the expected behavior.

0.7.2 Question: Was the behavior as expected? If not, what did you observe differently from what you thought you would? Why did this difference occur?

Answer: (Up to individual)

[7]:
```
# Run a "straight" trim trajectory with medium altitude, moderate turbulence
gust_params = MsgGustParams()
gust_params.sigma_u = 3 # Set values for the remaining variables
run_sim_and_display(trim, use_wind=True, gust_params=gust_params)
```


pn
pn
pn_e
pn(m)
Time (s)
pe
pe
pe_e
pe(m)
Time (s)
h
h
h_e
h_c
h(m)
Time (s)
wn
wn
wn_e
we
we_e
wind(m/s)
Time (s)
Va
Va
Va_e
Va_c
Va(m/s)
Time (s)
alpha
alpha
alpha_e
alpha(deg)
Time (s)
beta
beta
beta_e
beta(deg)
Time (s)
Vg
Vg
Vg_e
Vg(m/s)
Time (s)
phi
phi
phi_e
phi_c
phi(deg)
Time (s)
theta
theta
theta_e
theta_c
theta(deg)
Time (s)
psi
psi
psi_e
psi(deg)
Time (s)
chi
chi
chi_e
chi_c
chi(deg)
Time (s)
p
p
p_e
p(deg/s)
Time (s)
q
q
q_e
q(deg/s)
Time (s)
r
r
r_e
r(deg)
Time (s)
bx
bx
bx_e
by
by_e
bz
bz_e
bias(deg/s)
Time (s)
delta_e
delta_e
delta_e(deg)
Time (s)
delta_a
delta_a
delta_a(deg)
Time (s)
delta_r
delta_r
delta_r(deg)
Time (s)
delta_t
delta_t
delta_t(deg)
Time (s)



0.8 Simple code checking

The following code does not need to change. It should just be used as a sanity check so that you know the code is implemented properly. The output should not have any lines reading `Failed test!`

[8]:
```
from mav_sim.unit_tests.ch4_dynamics_test import run_all_tests
run_all_tests()
```

```
Starting motor_thrust_torque test

Calculated output:
(-0.033654137677838994, -0.0028236828549583586)
Expected output:
(-0.033654137677838994, -0.002823682854958359)
Passed test
Calculated output:
(-17.913526836048952, -0.7758235361365514)
Expected output:
(-17.91352683604895, -0.7758235361365506)
Passed test
```

```
End of test

Starting forces_moments test

Calculated output:
[[-3.40821628e-02]
 [ 0.00000000e+00]
 [ 1.07829786e+02]
 [ 2.82368285e-03]
 [ 8.94274083e-04]
 [ 0.00000000e+00]]
Expected output:
[[-3.40821628e-02]
 [ 0.00000000e+00]
 [ 1.07829786e+02]
 [ 2.82368285e-03]
 [ 8.94274083e-04]
 [ 0.00000000e+00]]
Passed test
Calculated output:
[[  -4.71982679]
 [  87.12283471]
 [-113.4856833 ]
 [ -44.44992726]
 [ -31.38114459]
 [   8.16544191]]
Expected output:
[[  -4.71982679]
 [  87.12283471]
 [-113.4856833 ]
 [ -44.44992726]
 [ -31.38114459]
 [   8.16544191]]
Passed test
End of test

Starting update_velocity_data test

Calculated output:
(0.0, 0.0, 0.0, array([[0.],
       [0.],
       [0.]]))
Expected output:
(0, 0, 0, array([[0.],
       [0.],
       [0.]]))
Passed test
Calculated output:
```

```
(10.379686170818202, 1.252059888018447, -0.3430148226710437, array([ 1.00712983,
-2.00500799,  3.00104193]))
Expected output:
(10.379686170818202, 1.252059888018447, -0.34301482267104366, array([
1.00712983, -2.00500799,  3.00104193]))
Passed test
End of test

Starting WindSimulation test

Calculated output:
[0.00529669 0.01058676 0.01587022]
Expected output:
[0.00529669 0.01058676 0.01587022]
Passed test
Calculated output:
[0.01320205 0.0150394  0.01030656]
Expected output:
[0.01320205 0.0150394  0.01030656]
Passed test
End of test
```