Brandon White

MAE 5010 – Autopilot Design and Test

Homework #2 (Due 09/14/2019)

==========

2)

BRANDON WHITE

Per BGARD + McLAIN:

$$\begin{pmatrix} f_x \\ f_y \\ f_z \end{pmatrix} = \frac{1}{2}\rho V_T^2 S \begin{pmatrix} C_x(\alpha) + C_{xq}(\alpha)\frac{c}{2V_T} q + C_{x\delta_e}(\alpha)\delta_e \\ C_{y_0}^{\text{sym.}} + C_{y_\beta}^{(\text{sym. wrt } \alpha)} \beta + C_{y_p}\frac{b}{2V_T} p + C_{y_r}\frac{b}{2V_T} r + C_{y_{\delta_a}}\delta_a + C_{y_{\delta_r}}\delta_r \\ C_Z(\alpha) + C_{Zq}(\alpha)\frac{c}{2V_T} q + C_{Z_{\delta_e}}(\alpha)\delta_e \end{pmatrix}$$

Where:

$$C_x(\alpha) = -C_D(\alpha)\cos(\alpha) + C_L(\alpha)\sin(\alpha)$$

$$C_{xq}(\alpha) = -C_{D_q}\cos(\alpha) + C_{L_q}\sin(\alpha)$$

$$C_{x\delta_e}(\alpha) = -C_{D_{\delta_e}}\cos(\alpha) + C_{D_{\delta_e}}\sin(\alpha)$$

$$C_Z(\alpha) = -\left(C_D(\alpha)\sin(\alpha) + C_L(\alpha)\cos(\alpha)\right)$$

$$C_{Z_q}(\alpha) = -\left(C_{D_q}\sin(\alpha) + C_{L_q}\cos(\alpha)\right)$$

$$C_{Z_{\delta_e}}(\alpha) = -\left(C_{D_{\delta_e}}\sin(\alpha) + C_{L_{\delta_e}}\cos(\alpha)\right)$$

and $C_L(\alpha) \triangleq$ eq(6) on HW #2.

and $C_D(\alpha) = C_{D_0} + \frac{(C_{L_0} + C_{L_\alpha}\alpha)^2}{\pi(0.8)AR}$

where $e = 0.8$ by assumption.

$$\begin{bmatrix} \dot{X} \\ \dot{Y} \\ \dot{Z} \end{bmatrix} = R_{BF}(\phi,\theta,\psi)\begin{bmatrix} 0 \\ 0 \\ mg \end{bmatrix} + \begin{pmatrix} f_x \\ f_y \\ f_z \end{pmatrix} + \begin{bmatrix} T_{max}\,\delta_T \\ 0 \\ 0 \end{bmatrix}$$

3)

Discussion

To obtain the stability derivatives for the Boeing 737, the moments of inertia were calculated using provided radii of gyration. These and other provided geometric properties were compiled into a .mass file. Utilizing the Boeing 737 .avl file provided for the class and custom .mass file, the stability derivatives were calculated through use of Athena Vortex Lattice (AVL) software provided through the MIT website. There are some areas for concern related to the coefficients (particularly $C_{m\alpha}, C_{mq}, and\ C_{Lq}$), however, I have not had an introduction to AVL sufficient for debugging potential issues.

$$I_{xx} = mK_x^2 = 1767812.6\ kg * m\text{^}2$$
$$I_{yy} = mK_y^2 = 6414510.5\ kg * m\text{^}2$$
$$I_{zz} = mK_z^2 = 7480654.9\ kg * m\text{^}2$$

```
Stability-axis derivatives...

                            alpha                beta
                    ----------------    ----------------
z' force CL |    CLa =   6.247231    CLb =   0.000000
y  force CY |    CYa =  -0.000000    CYb =  -0.609577
x' mom.  Cl'|    Cla =  -0.000000    Clb =  -0.045212
y  mom.  Cm |    Cma = -29.242979    Cmb =   0.000000
z' mom.  Cn'|    Cna =   0.000000    Cnb =   0.420378

                   roll rate  p'      pitch rate  q'        yaw rate  r'
                ----------------    ----------------    ----------------
z' force CL |    CLp =   0.000000    CLq =  71.850571    CLr =   0.000000
y  force CY |    CYp =  -0.039097    CYq =  -0.000000    CYr =   1.242737
x' mom.  Cl'|    Clp =  -0.487152    Clq =   0.000000    Clr =   0.095460
y  mom.  Cm |    Cmp =   0.000000    Cmq =-379.958527    Cmr =  -0.000000
z' mom.  Cn'|    Cnp =   0.047273    Cnq =   0.000000    Cnr =  -1.020538

Neutral point  Xnp =  19.823826

Clb Cnr / Clr Cnb  =   1.149798    (  > 1 if spirally stable )
```

Fig 1. Results of AVL Analysis of 737-800

4)

Discussion

Integration of the functions for lift, drag, and sideslip were successful and operated like expected with changes to the control inputs as seen in figure 4. Integration of the moments, however, threw the MAV simulation "for a loop" as seen in figure 5. It is unknown at this time, why the moment equations from Beard and McLain are not operating as expected. Review of the derivation of equations has not provided any artifacts of sign convention error, nor any inconsistencies in the code's equations. Further areas of investigation will include reorganization of MAV class to make code more visually appealing and integration of a Eulerian integration method for debugging.

$$\begin{pmatrix} f_x \\ f_y \\ f_z \end{pmatrix} = \frac{1}{2}\rho V_a^2 S \begin{pmatrix} C_X(\alpha) + C_{X_q}(\alpha)\frac{c}{2V_a}q + C_{X_{\delta_e}}(\alpha)\delta_e \\ C_{Y_0} + C_{Y_\beta}\beta + C_{Y_p}\frac{b}{2V_a}p + C_{Y_r}\frac{b}{2V_a}r + C_{Y_{\delta_a}}\delta_a + C_{Y_{\delta_r}}\delta_r \\ C_Z(\alpha) + C_{Z_q}(\alpha)\frac{c}{2V_a}q + C_{Z_{\delta_e}}(\alpha)\delta_e \end{pmatrix}$$

$$\begin{pmatrix} l \\ m \\ n \end{pmatrix} = \frac{1}{2}\rho V_a^2 S \begin{pmatrix} b\left[C_{l_0} + C_{l_\beta}\beta + C_{l_p}\frac{b}{2V_a}p + C_{l_r}\frac{b}{2V_a}r + C_{l_{\delta_a}}\delta_a + C_{l_{\delta_r}}\delta_r\right] \\ c\left[C_{m_0} + C_{m_\alpha}\alpha + C_{m_q}\frac{c}{2V_a}q + C_{m_{\delta_e}}\delta_e\right] \\ b\left[C_{n_0} + C_{n_\beta}\beta + C_{n_p}\frac{b}{2V_a}p + C_{n_r}\frac{b}{2V_a}r + C_{n_{\delta_a}}\delta_a + C_{n_{\delta_r}}\delta_r\right] \end{pmatrix}$$

$$C_X(\alpha) \triangleq -C_D(\alpha)\cos\alpha + C_L(\alpha)\sin\alpha$$
$$C_{X_q}(\alpha) \triangleq -C_{D_q}\cos\alpha + C_{L_q}\sin\alpha$$
$$C_{X_{\delta_e}}(\alpha) \triangleq -C_{D_{\delta_e}}\cos\alpha + C_{L_{\delta_e}}\sin\alpha$$
$$C_Z(\alpha) \triangleq -C_D(\alpha)\sin\alpha - C_L(\alpha)\cos\alpha$$
$$C_{Z_q}(\alpha) \triangleq -C_{D_q}\sin\alpha - C_{L_q}\cos\alpha$$
$$C_{Z_{\delta_e}}(\alpha) \triangleq -C_{D_{\delta_e}}\sin\alpha - C_{L_{\delta_e}}\cos\alpha,$$

Fig 2. Beard and McLain Body-Frame Aerodynamic Equations

```
#Rotated coefficients
Cx = -self.CD(alpha, w_coeff[4], w_coeff[0:2], self.wing[0]/self.wing[1])*cos(alpha) + self.CL_stall(alpha, self.wing[5], w_coeff[0:2])*sin(alpha)
Cxq = -w_coeff[6]*cos(alpha) + w_coeff[2]*sin(alpha)
Cxdele = -h_coeff[7]*cos(alpha) + h_coeff[3]*sin(alpha)
Cz = -self.CD(alpha, w_coeff[4], w_coeff[0:2], self.wing[0]/self.wing[1])*sin(alpha) - self.CL_stall(alpha, self.wing[5], w_coeff[0:2])*cos(alpha)
Czq = -w_coeff[6]*sin(alpha) - w_coeff[2]*cos(alpha)
Czdele = -h_coeff[7]*sin(alpha) - h_coeff[3]*cos(alpha)

#NEEDS REVISION
#FORCES
X = Q*self.wing[0]*self.wing[1]*(Cx + Cxq*self.wing[1]/(2*V_t)*q + Cxdele*self.controls[0])
Y = Q*self.wing[0]*self.wing[1]*(v_coeff[3]*self.controls[3])
Z = Q*self.wing[0]*self.wing[1]*(Cz + Czq*self.wing[1]/(2*V_t)*q + Czdele*self.controls[0])

#MOMENTS
L = Q*self.wing[0]**2*self.wing[1]*(w_coeff[8][0]*self.controls[2] + v_coeff[8][0]*self.controls[3])
M = Q*self.wing[0]*self.wing[1]**2*(w_coeff[8][3] + w_coeff[8][4]*alpha + w_coeff[8][5]*self.wing[1]/(2*V_t)*q + h_coeff[8][1]*self.controls[0])
N = Q*self.wing[0]**2*self.wing[1]*(w_coeff[8][2]*self.controls[2])
```

Fig 3. Code Execution of Beard and McLain Equations

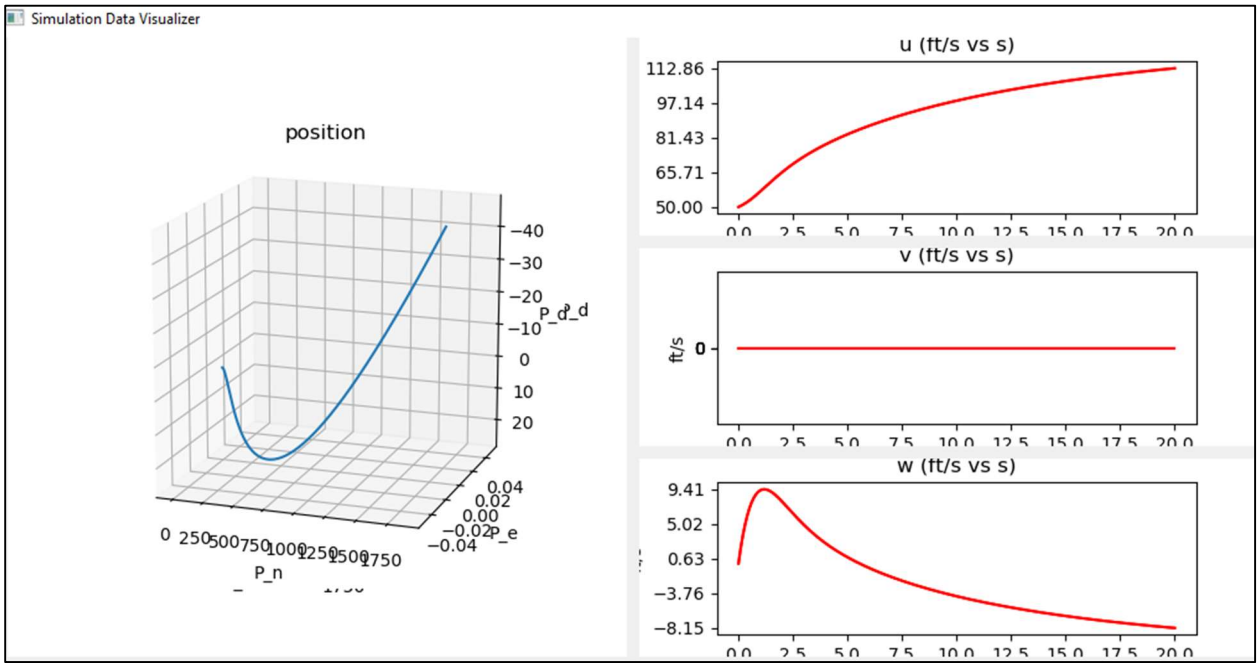

Fig 4. Simulation Results for Model with Only Force Terms

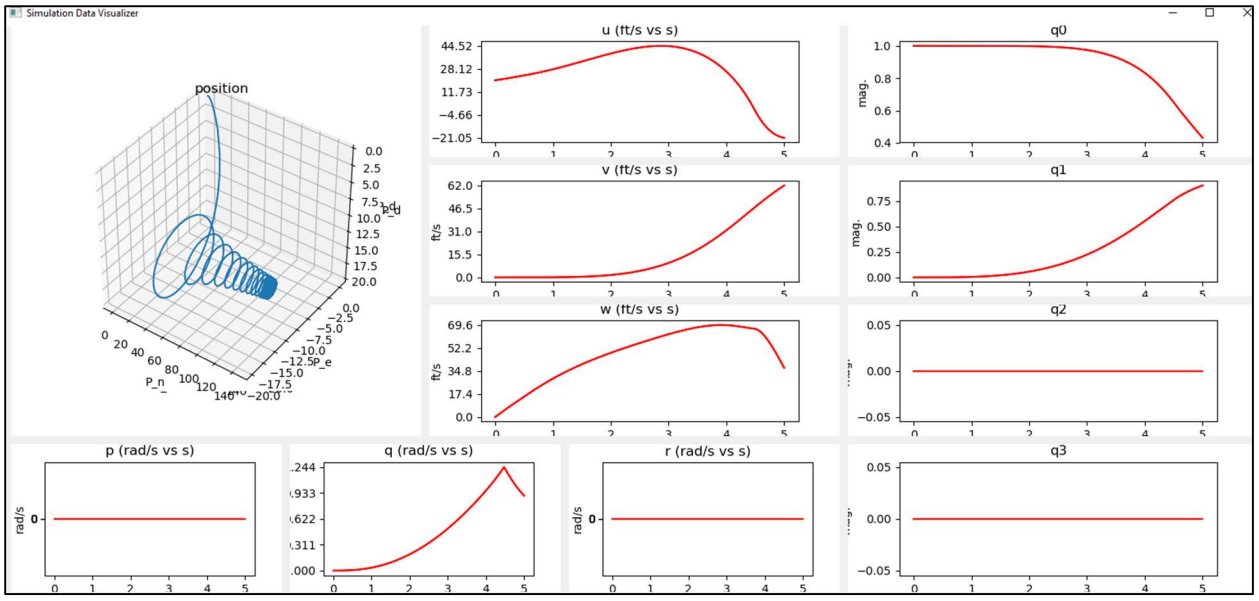

Fig 5. Simulation Results for Model with Force and Moment Terms

5)

Discussion

No amount of control inputs or good intentions were able to stabilize the simulation with moments added. The use of control inputs to alter the "No Moment Simulation" was considered instead, to verify the operation of controls in the sim. A combination of conditions for which the perturbation to altitude was mitigated over a two-minute flight time was found with u = 120ft/s, $\delta = .14\, rad$, and $T = T_{max}\delta_T|_{\delta_T=40\%}$. Note that there is oscillation of the flight path at the beginning as the MAV falls ¾ ft, but over the two minutes only nets a positive ¾ ft. It is well-expected that any gusts would play a role larger than this disturbance on a sub-30lb aircraft, so this was considered successful control configuration (albeit impractical due to the high speed requirement). This need for high speed is likely due to the neglect of moment terms in the simulation.

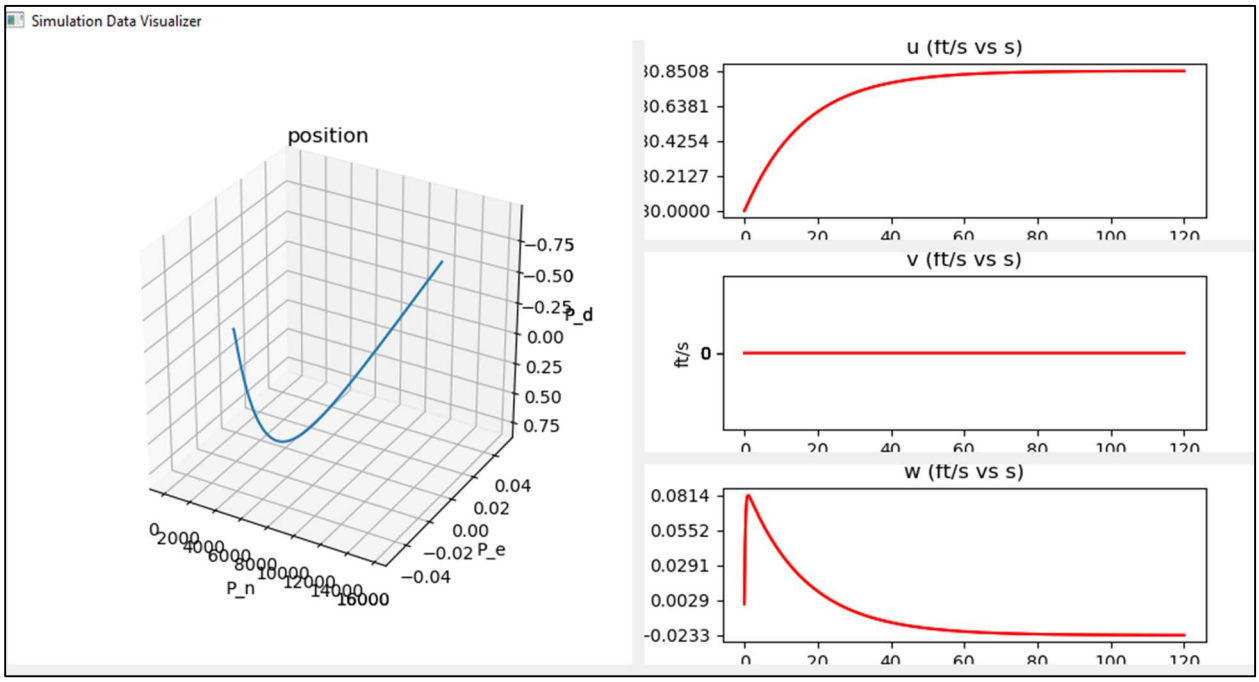

Fig. 6 Flight Path After Stabilization

```python
#                   TITLE BLOCK
#************************************************
#Author:    Brandon White
#Date:      08/28/2019
#Desc:      This set of functions to solve the problems
#          given on HW1 of MAE 5010.
#************************************************
from rotations import *

def Euler3212EP(angles, radians = False, rounding = True):
    #Expects degrees but can accept radians with flag set

    import math

    #convert to radians for math
    if not radians:
        for i in range(3):
            angles[i] = angles[i] * (math.pi/180)

    [psi, theta, phi] = angles

    e = [math.cos(psi/2)*math.cos(theta/2)*math.cos(phi/2) +
    math.sin(psi/2)*math.sin(theta/2)*math.sin(phi/2),
            math.cos(psi/2)*math.cos(theta/2)*math.sin(phi/2) -
            math.sin(psi/2)*math.sin(theta/2)*math.cos(phi/2),
            math.cos(psi/2)*math.sin(theta/2)*math.cos(phi/2) +
            math.sin(psi/2)*math.cos(theta/2)*math.sin(phi/2),
            math.sin(psi/2)*math.cos(theta/2)*math.cos(phi/2) -
            math.cos(psi/2)*math.sin(theta/2)*math.sin(phi/2)]

    if rounding:
        for i in range(4):
            e[i] = round(e[i],5)

    return e

def EP2Euler321(e, rounding = True):
    import math

    #outputs psi, theta, phi
    angles = [math.atan2(2 * (e[0]*e[3] + e[2]*e[1]), e[0]**2 +
    e[1]**2 - e[2]**2 - e[3]**2),
                math.asin(2 * (e[0]*e[2] - e[1]*e[3])),
                math.atan2(2 * (e[0]*e[1] + e[2]*e[3]), e[0]**2 +
                e[3]**2 - e[1]**2 - e[2]**2)]
```

```
                e[3]**2 - e[1]**2 - e[2]**2)]

    #Convert to degrees
    for i in range(3):
        angles[i] = angles[i] * (180/math.pi)
        if rounding:
            angles[i] = round(angles[i],2)

    return [angles[2], angles[1], angles[0]] #returns degrees

def make_gamma(I):
    [Ixz, Ix, Iy, Iz] = I
    Gamma = [Ix*Iz - Ixz**2, 0, 0, 0, 0, 0, 0, 0, 0]
    Gamma[1] = Ixz*(Ix-Iy-Iz)/Gamma[0]
    Gamma[2] = (Iz*(Iz-Iy)+Ixz**2)/Gamma[0]
    Gamma[3] = Iz/Gamma[0]
    Gamma[4] = Ixz/Gamma[0]
    Gamma[5] = (Iz - Ix)/Iy
    Gamma[6] = Ixz/Iy
    Gamma[7] = (Ix*(Ix-Iy)+Ixz**2)/Gamma[0]
    Gamma[8] = Ix/Gamma[0]
    return Gamma

def pos_kin(psi, theta, phi, u, v, w):
    #d/dt([p_n, p_e, p_d])
    from math import cos, sin
    from numpy import matmul

    A1 = [[cos(theta)*cos(psi), sin(phi)*sin(theta)*cos(psi) -
    cos(phi)*sin(psi), cos(phi)*sin(theta)*cos(psi) +
    sin(phi)*sin(psi)],
                [cos(theta)*sin(psi), sin(phi)*sin(theta)*sin(psi) +
                cos(phi)*cos(psi), cos(phi)*sin(theta)*sin(psi) -
                sin(phi)*cos(psi)],
                [-sin(theta), sin(phi)*cos(theta),
                cos(phi)*cos(theta)]]
    b1 = [u, v, w]
    return matmul(A1, b1)

def pos_dyn(p, q, r, u, v, w, Fx, Fy, Fz, m):
    #d/dt([u, v, w])
    #>>> Need to move to the body frame?
    x_dot = [r*v-q*w + Fx/m,
            p*w-r*u + Fy/m,
            q*u-p*v + Fz/m]
```

```
79      return x_dot
80
81  def rot_kin(e0, e1, e2, e3, r, p, q):
82      #d/dt( e )
83      from numpy import matmul
84      A3 = [[0, -p/2, -q/2, -r/2],
85              [p/2, 0, r/2, -q/2],
86              [q/2, -r/2, 0, p/2],
87              [r/2, q/2, -p/2, 0]]
88      b3 = [e0, e1, e2, e3]
89      return matmul(A3, b3)
90
91  def rot_dyn(Gamma, p, q, r, L, M, N, Iy):
92      #d/dt([p, q, r])
93      #>>> Need to move to the body frame?
94      x_dot = [Gamma[1]*p*q - Gamma[2]*q*r + Gamma[3]*L + Gamma[4]*N,
95              Gamma[5]*p*r - Gamma[6]*(p**2-r**2) + 1/Iy*M,
96              Gamma[7]*p*q - Gamma[1]*q*r + Gamma[4]*L + Gamma[8]*N]
97      return x_dot
98
99  def normalize_quaterions(e):
100     import numpy
101     from numpy.linalg import norm
102     [e0, e1, e2, e3] = numpy.array(e)/norm(numpy.array(e))
103     return [round(e0,3), round(e1,3), round(e2,3), round(e3,3)]
104
105
106 def derivatives(state, t, MAV):
107     #state: [p_n, p_e, p_d, u, v, w, e0, e1, e2, e3, p, q, r]
108     #FM:    [Fx, Fy, Fz, Ell, M, N]
109     #MAV:   MAV.inert, MAV.m, MAV.gravity_needed
110
111     state[6:10] = normalize_quaterions(state[6:10])
112
113     MAV.state0 = state
114
115     from math import sin, cos
116     print()
117     print('Time: ' + str(t))
118     print('State: ' + str(state))
119     #Unpack state, FM, MAV
120     [p_n, p_e, p_d, u, v, w, e0, e1, e2, e3, p, q, r] = state
121     storage = MAV.update_FM(t)
122     [Fx, Fy, Fz, L, M, N] = MAV.FM
123     [Ixz, Ix, Iy, Iz] = MAV.inert
```

```
124
125     #Get angle measures
126     angles = EP2Euler321([e0, e1, e2, e3])
127     [psi, theta, phi] = angles
128
129     #Get Xdot Terms
130     d_dt = [[], [], [], []]
131     d_dt[0] = pos_kin(psi, theta, phi, u, v, w)
132     d_dt[1] = pos_dyn(p, q, r, u, v, w, Fx, Fy, Fz, MAV.mass)
133     d_dt[2] = rot_kin(e0, e1, e2, e3, p, q, r)
134     d_dt[3] = rot_dyn(make_gamma(MAV.inert), p, q, r, L, M, N,
    •   MAV.inert[2])
135
136     #Build One Vector of Xdot
137     xdot = []
138     for eqn_set in d_dt:
139         for dot in eqn_set:
140             xdot.append(dot)
141
142     return xdot
143
144 def integrator(MAV, tf = 1, delta_t = 0.05, graphing = False):
145     from numpy import linspace
146     from scipy.integrate import odeint
147
148     #Make the time values
149     descrete_pts = (tf/delta_t) // 1  # force an integer
150     t = linspace(0, tf, descrete_pts + 1)
151
152     MAV.delta_t = delta_t
153     #Integration Step
154     outputs = odeint(derivatives, MAV.state0, t, args = (MAV,))
155
156     #Optional 3D Path Graphing
157     if graphing:
158         from mpl_toolkits import mplot3d
159         import matplotlib.pyplot as plt
160         fig = plt.figure()
161         ax = plt.axes(projection="3d")
162         ax.plot3D(outputs[:,0], outputs[:,1], outputs[:,2],
    •    linestyle='-', marker='.')
163         ax.set_xlabel('P_n')
164         ax.set_ylabel('P_e')
165         ax.set_zlabel('P_z')
```

```
166             ax.invert_zaxis()
167             plt.show()
168 
169         return [t, outputs]
170 
```

```
#                   TITLE BLOCK
#************************************************
#Author:    Brandon White
#Date:      08/26/2019
#Desc:      Creates a MAV object with mass, moment
#          of inertia, and gravity properties
#************************************************

from rotations import *

#Calling the class with an aircraft name below creates an MAV object
class MAV:
    def __init__(self, aircraft = "None"):
       #All units listed in English units as denoted
       #NOTE: alpha and beta in radians
       self.name = aircraft
       self.last_update = 0
       self.delta_t = 10
       self.mass = 10  # Mass (slug)
       self.dynamic_density = False #True uses lapse rate for
       rho=f(h)

       #Inert = [Ixz, Ix, Iy, Iz]
       self.inert = [20, 10, 10, 10]  # Moment of Inertia (lbf*ft^2)

       #State = [p_n, p_e, p_d, u, v, w, e0, e1, e2, e3, p, q, r]
       self.state0 = [0, 0, -500, 50, 0, 0, 1, 0, 0, 0, 0, 0, 0]
            #Level flight at 500 ft at 50 ft/s

       #FM = [Fx, Fy, Fz, Ell, M, N]
       self.FM = [0, 0, 0, 0, 0, 0]
            #Equations for time-variant forces and moments
       self.FMeq = [0, 0, 0, 0, 0, 0]

       self.thrust_max = 10  #lbf

       #Geometric Properties and Coefficients
        #Order: b, c, x_cg, y_cg, z_cg, stall, coefficients
            #stall: True/False, M, alpha_0
            #coeff: CL0, CLa/b, CLq, CL_del_control, CD0, CDa/b,
            CDq, CD_del_control, [spare]
       self.wing = [0, 0, 0, 0, 0, [True, 50, 0.471],
                    [0, 0, 0, 0, 0, 0, 0, 0, [0,0,0]]]
       self.hstab = [0, 0, 0, 0, 0, [False, 0, 0],
                    [0, 0, 0, 0, 0, 0, 0, 0, [0,0,0]]]
```

```
                    [0, 0, 0, 0, 0, 0, 0, 0, [0,0,0]]]
        self.vstab = [0, 0, 0, 0, 0, [False, 0, 0],
                      [0, 0, 0, 0, 0, 0, 0, 0, [0,0,0]]]

        #Controls Deflections: del_e, del_t(<1), del_a, del_r
        self.controls = [ 0, 1, 0, 0]

        #self.coeffeq = [0, 0, 0, 0, 0, 0, 0, 0,
         #                 0, 0, 0, 0, 0, 0, 0, 0,
         #                 0, 0, 0, 0, 0, 0, 0, 0]

        if aircraft != "None":
            try:
                method_to_call = getattr(self, aircraft.lower())
                method_to_call()
            except:
                print("No preconfig by given name: " +
                aircraft.lower())

    def density(self):
        if self.dynamic_density:
            #Define STD SL Terms
            ##P_0 = 101325 #Pa
            L = 0.0065 #K/m
            M = 0.0289644 #kg/mol
            R = 8.31447 #J/(mol K)
            g = 9.80665 #m/s^2
            T_0 = 288.15 #K
            p = P_0*(1 + (L*0.3048*self.state0[2])/T_0)**(g*M/(R*L))
            return p*M/(R*T) * (0.00194) #Covert to slug/ft^3
        else:
            return 0.002377 #SL slug/ft^3

    def update_state0(self, new_state):
        if len(new_state) != 13:
            print("Error - Not 13 items! \n You might need to
            convert angular\
                    values to quaternions...")
        else:
            self.state0 = new_state

    def CL_stall(self, alpha, model, coeff):
        from math import exp, sin, cos
        from numpy import sign
        [discard, M, alpha_star] = model
```

```
86          del_alpha = alpha - alpha_star
87          add_alpha = alpha + alpha_star
88          sig = (1+exp(-M*del_alpha)+exp(M*add_alpha))/((1+exp(-
            M*del_alpha))*(1+exp(M*add_alpha)))
89          CL = (1-sig)*(coeff[0]+coeff[1]*alpha) +
            sig*2*sign(alpha)*(sin(alpha)**2)*cos(alpha)
90          return CL
91
92      def CD(self,alpha, cd0, coeff, AR):
93          #Assume Oswald = 0.8
94          from math import pi
95          return cd0 + (coeff[0] + coeff[1]*alpha)**2/(pi*0.8*AR)
96
97      def aero_terms(self):
98          from math import atan, sin, cos
99          from numpy import matmul, transpose
100
101         [p_n, p_e, p_d, u, v, w, e0, e1, e2, e3, p, q, r] =
            self.state0
102         V_t = (u**2 + v**2 + w**2)**(1/2)  #NOTE: No wind included
103         Q = 0.5 * V_t**2 *self.density()
104
105         #print("Q: " + str(Q))
106
107         alpha = atan(w/u)  #NOTE: Negative sign since Pd is inverted
108         beta = atan(v/u)
109         angles = [alpha, beta]
110
111         w_coeff = self.wing[6]
112         h_coeff = self.hstab[6]
113         v_coeff = self.vstab[6]
114
115         #Rotated coefficients
116         Cx = -self.CD(alpha, w_coeff[4], w_coeff[0:2], self.wing[0]/
            self.wing[1])*cos(alpha) + self.CL_stall(alpha,
            self.wing[5], w_coeff[0:2])*sin(alpha)
117         Cxq = -w_coeff[6]*cos(alpha) + w_coeff[2]*sin(alpha)
118         Cxdele = -h_coeff[7]*cos(alpha) + h_coeff[3]*sin(alpha)
119         Cz = -self.CD(alpha, w_coeff[4], w_coeff[0:2], self.wing[0]/
            self.wing[1])*sin(alpha) - self.CL_stall(alpha,
            self.wing[5], w_coeff[0:2])*cos(alpha)
120         Czq = -w_coeff[6]*sin(alpha) - w_coeff[2]*cos(alpha)
121         Czdele = -h_coeff[7]*sin(alpha) - h_coeff[3]*cos(alpha)
122
123         #NEEDS REVISION
```

```
        #NEEDS REVISION
        #FORCES
        X = Q*self.wing[0]*self.wing[1]*(Cx + Cxq*self.wing[1]/
        (2*V_t)*q + Cxdele*self.controls[0])
        Y = Q*self.wing[0]*self.wing[1]*(v_coeff[3]*self.controls[3])
        Z = Q*self.wing[0]*self.wing[1]*(Cz + Czq*self.wing[1]/
        (2*V_t)*q + Czdele*self.controls[0])

        #MOMENTS
        L =
        Q*self.wing[0]**2*self.wing[1]*(w_coeff[8][0]*self.controls[2
        ] + v_coeff[8][0]*self.controls[3])
        M = Q*self.wing[0]*self.wing[1]**2*(w_coeff[8][3] +
        w_coeff[8][4]*alpha + w_coeff[8][5]*self.wing[1]/(2*V_t)*q +
        h_coeff[8][1]*self.controls[0])
        N =
        Q*self.wing[0]**2*self.wing[1]*(w_coeff[8][2]*self.controls[2
        ])

        #return [X, Y, Z, L, M, N]
        return [X, Y, Z, 0, 0, 0]

    def update_FM(self, t):
        from math import sin, cos
        from integrator import EP2Euler321

        #Angularize Gravity
        [psi, theta, phi] = EP2Euler321(self.state0[6:10])
        print('Angles:' + str([psi, theta, phi]))

        #All Forcing Functions
        for i in range(6):
            try:
                self.FM[i] = self.FMeq[i](t)
            except:
                self.FM[i] = self.FMeq[i]
        #print("FM w/Forcing:" + str(self.FM))

        #Add in Aero Terms
        aero_b = self.aero_terms()
        self.FM[0] += -32.2*self.mass*sin(theta) + aero_b[0] +
        self.thrust_max*self.controls[1]
        self.FM[1] += 32.2*self.mass*cos(theta)*sin(phi) + aero_b[1]
        self.FM[2] += 32.2*self.mass*cos(phi)*cos(phi) + aero_b[2]
        self.FM[3] += aero_b[3]
```

```
159         self.FM[4] += aero_b[4]
160         self.FM[5] += aero_b[5]
161         print("TOTAL FM" + str(self.FM))
162         return self.FM
163 
164     #================================================================
165     #Add templated aircraft below this line to pre-generate aircraft
166     #================================================================
167     def hw1_1(self):
168         import warnings
169         warnings.warn("This aircraft is depreciated",
    •         DeprecationWarning)
170         self.state0 = [100, 200, -500, 50, 0, 0,
171                        0.70643, 0.03084, 0.21263, 0.67438, 0, 0, 0]
172         self.FMeq = [0, 0, 0, 0, 0, 0]
173 
174     def hw1_2(self):
175         import warnings
176         warnings.warn("This aircraft is depreciated",
    •         DeprecationWarning)
177         from math import sin, cos
178         self.state0 = [100, 200, -500, 50, 0, 0,
179                        0.70643, 0.03084, 0.21263, 0.67438, 0, 0, 0]
180         self.FMeq = [(lambda t: sin(t)), 0, 0,
181                        0, 1e-4, 0]
182 
183     def hw2(self):
184         #All units listed in English units as denoted
185         #NOTE: alpha and beta in radians
186         self.mass = 0.925  # Mass (slug)
187         self.dynamic_density = False #True uses lapse rate for
    •         rho=f(h)
188 
189         #Inert = [Ixz, Ix, Iy, Iz]
190         self.inert = [2.857, 19.55, 26.934, 14.74]  # Moment of
    •         Inertia (lbf*ft^2)
191 
192         #State = [p_n, p_e, p_d, u, v, w, e0, e1, e2, e3, p, q, r]
193         self.state0 = [0, 0, 0, 20, 0, 0, 1, 0, 0, 0, 0, 0, 0]
194              #Level flight at 0ft at 20 ft/s
195 
196         #FM = [Fx, Fy, Fz, Ell, M, N]
197         self.FM = [0, 0, 0, 0, 0, 0]
198              #Equations for time-variant forces and moments
199         self.FMeq = [0, 0, 0, 0, 0, 0]
```

```
200
201         self.thrust_max = 5.62  #lbf
202
203         #Geometric Properties and Coefficients
204          #Order: b, c, x_cg, y_cg, z_cg, stall, coefficients
205              #stall: True/False, M, alpha_0
206              #coeff: CL0, CLa/b, CLq, CL_del_control, CD0, CDa/b,
  •              CDq, CD_del_control, [spare]
207         self.wing = [9.413, 0.6791, 0.9843, 0, 0, [True, 50, 0.471],
208                      [0.28, 3.5, 0, 0, 0.015, 0, 0, 0, [0.08, 0,
  •                      0.06, -0.02, -0.38, -3.6]]]
209         self.hstab = [2.297, 0.381, 0.8202, 0, 0, [False, 0, 0],
210                      [0.1, 5.79, 0, -0.36, 0.01, 0, 0, 0, [0, -0.5,
  •                      0]]]
211         self.vstab = [.9843, 0.381, 0.8202, 0, 0, [False, 0, 0],
212                      [0, 5.79, 0, -0.17, 0.01, 0, 0, 0, [0.105, 0,
  •                      0]]]
213
214         #Controls Deflections: del_e, del_t(<1), del_a, del_r
215         self.controls = [ 0, 1, 0, 0]
216
```