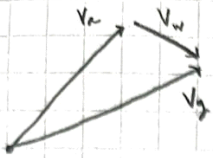
Kraft Worked w/ Adin Sacho-Tanzer Drone Control HW 2 Due: 2/13/25

Exercise 1:

a)



V_a = Vel airplane wrt moving air

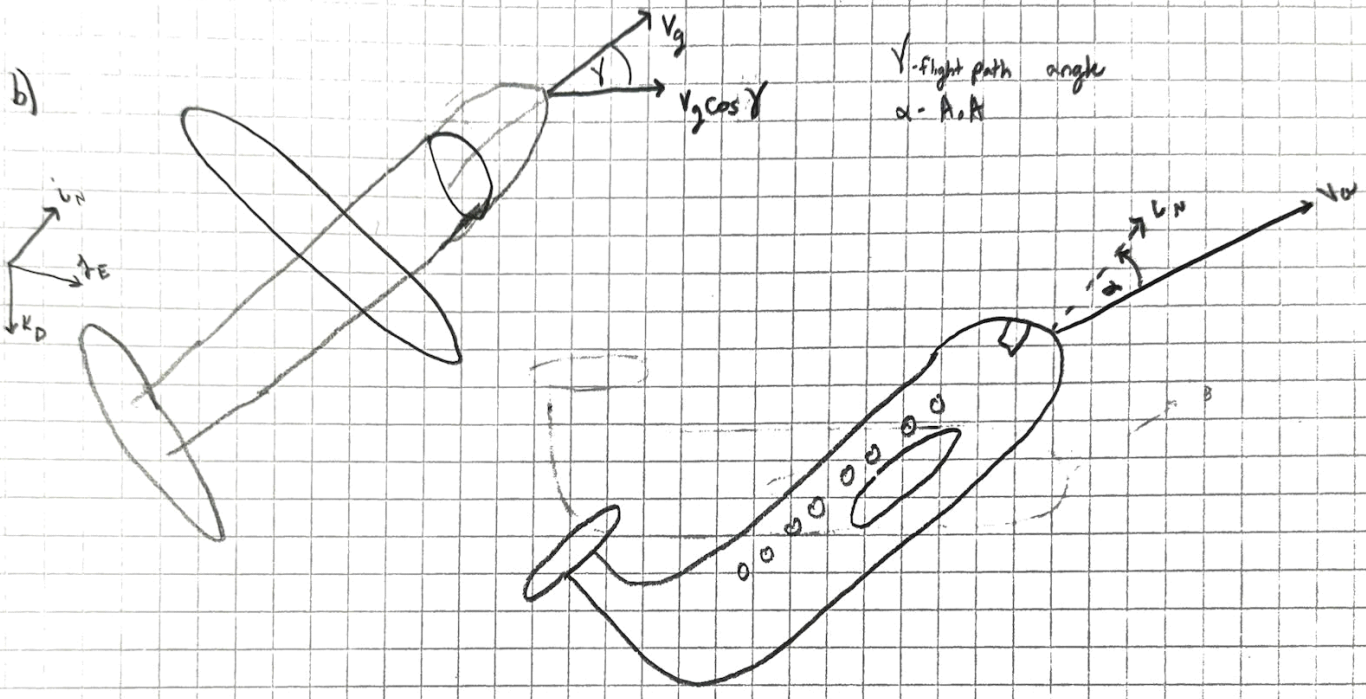
V_w = Vel wind wrt ground

V_g = Vel plane wrt ground

$V_g = V_a + V_w$ = Vel airplane wrt air + Vel air wrt ground = Vel airplane wrt ground

Since the velocities are all relative to one another, it is convinient to use linear algebra and express the velocities as vectors, thus, by vector addition, we can visualize it as a triangle

b)



c)

$$J_z = \int (x^2 + z^2)\, dm$$

x & z are coordinates of mass reltive to origin

dm is the differential mass

A larger J_z value would indicate mass is spread further from the 2-axis making rotations about the 2-axis (i.e. pitching the aircraft) more difficult. The lower J_z, the less energy or torque will be required to pitch the aircraft

d) we need 12 states, 6 DOF represent position while 6 represent velocities

P_n, P_e, P_d position in space (or x, y, z) in ground frame

ϕ, θ, ψ Euler angles; roll, pitch, yaw inert inertial frame

u, v, w linear velocities in body frame

p, q, r angular velocities in body frame

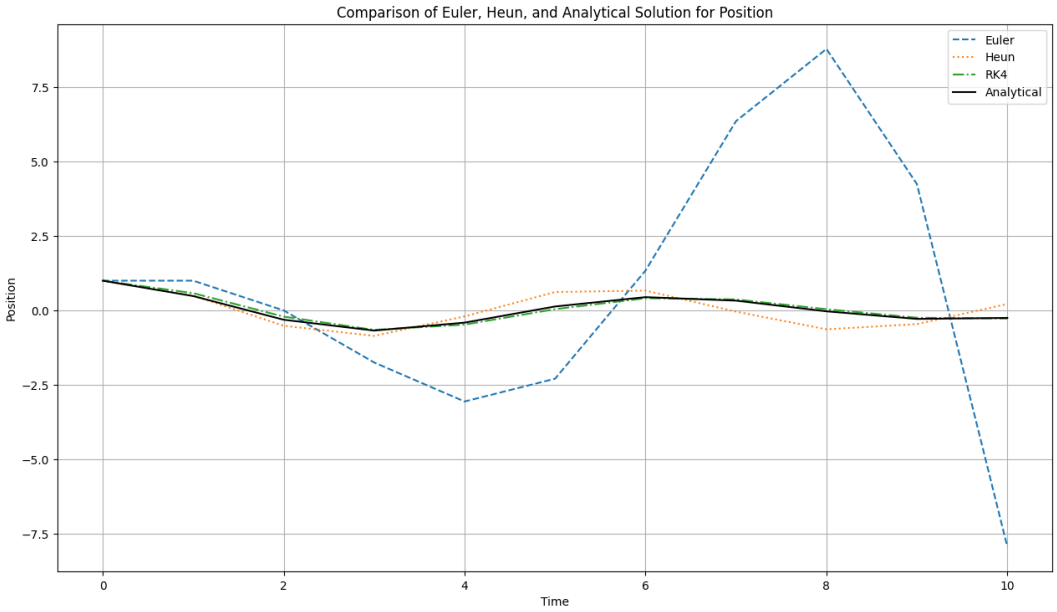
Exercise 2:

a) it works!

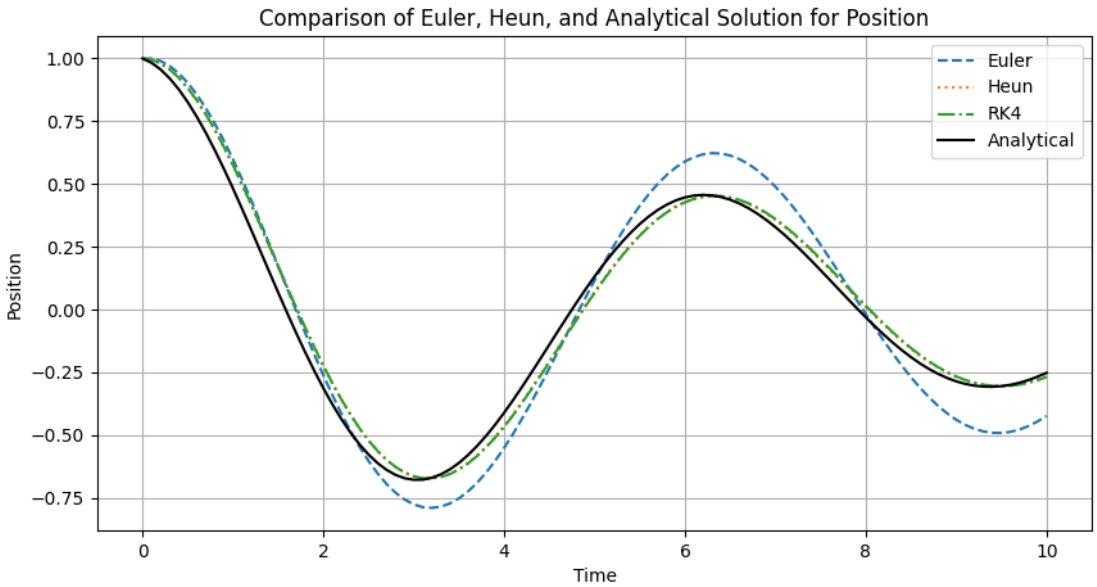
b) Done!

c) Even at really high step sizes (dt=1), RK4 tracks reasonably well where heun & euler don't

dt = 1:



dt =0.1:



Exercise 3

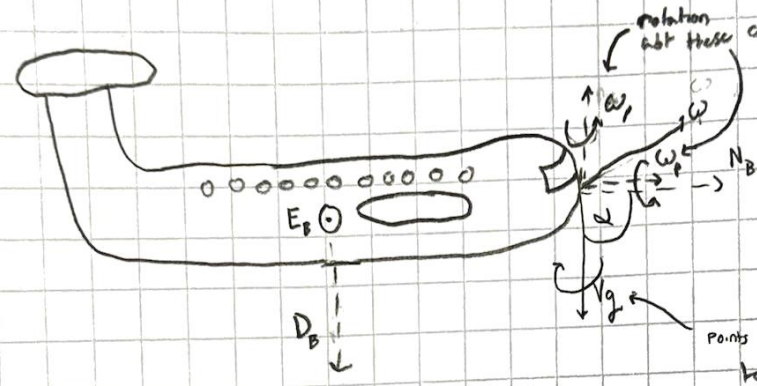
a) $\underline{\underline{J}}\,\dot{\underline{\omega}} = \underline{m} - \underline{\omega} \times (\underline{\underline{J}} \cdot \underline{\omega})$

$$\begin{pmatrix} m_1 \\ m_2 \\ m_3 \end{pmatrix} = \begin{pmatrix} J_1 & 0 & 0 \\ 0 & J_2 & 0 \\ 0 & 0 & J_3 \end{pmatrix} \begin{pmatrix} \dot{p} \\ \dot{q} \\ \dot{r} \end{pmatrix} + \begin{pmatrix} p \\ q \\ r \end{pmatrix} \times \left[\begin{pmatrix} J_1 & 0 & 0 \\ 0 & J_2 & 0 \\ 0 & 0 & J_3 \end{pmatrix} \begin{pmatrix} p \\ q \\ r \end{pmatrix} \right] = \begin{pmatrix} J_1 \dot{p} \\ J_2 \dot{q} \\ J_3 \dot{r} \end{pmatrix} + \begin{vmatrix} i & j & k \\ p & q & r \\ J_1 p & J_2 q & J_3 r \end{vmatrix}$$

$$\begin{vmatrix} + & - & + \\ - & + & - \\ + & - & + \end{vmatrix}$$

$$m_1 = J_1 \dot{p} + (J_2 qr - J_3 qr)i = J_1 \dot{p} + (J_3 - J_2) qr$$
$$m_2 = J_2 \dot{q} + (J_3 rp - J_1 rp)j = J_2 \dot{q} + (J_1 - J_3) pr$$
$$m_3 = J_3 \dot{r} + (J_1 pq - J_2 pq) = J_3 \dot{r} + (J_1 - J_2) pq$$

b)



α is the $\angle$ between V_B and i_B/N the diagram assume no α

Points downward in inertial frame, I chose to align inertial + Body Frame for the sake of the sketch. Can also be applied to cons

c) $\dot{p}=0 \quad \dot{r}=0 \quad q=0$ so $\dot{q}=0$

Since $q=0$, we need to look @ eqn 2 b/c w/ m. 1 & 3 zero out

$$m_2 = \cancel{J_2 \dot{q}}^{\,0} + (J_1 - J_3) pr$$

If $J_3 = J_1 + J_2$:

$$J_1 - J_3 = J_1 - (J_1 + J_2) = -J_2$$

$$m_2 = -J_2 pr$$

this is the requirement to have $q=0$

this is the pitch motion so the elevator should be used to provide this

d) $J_1 \ll J_2 \approx J_3$

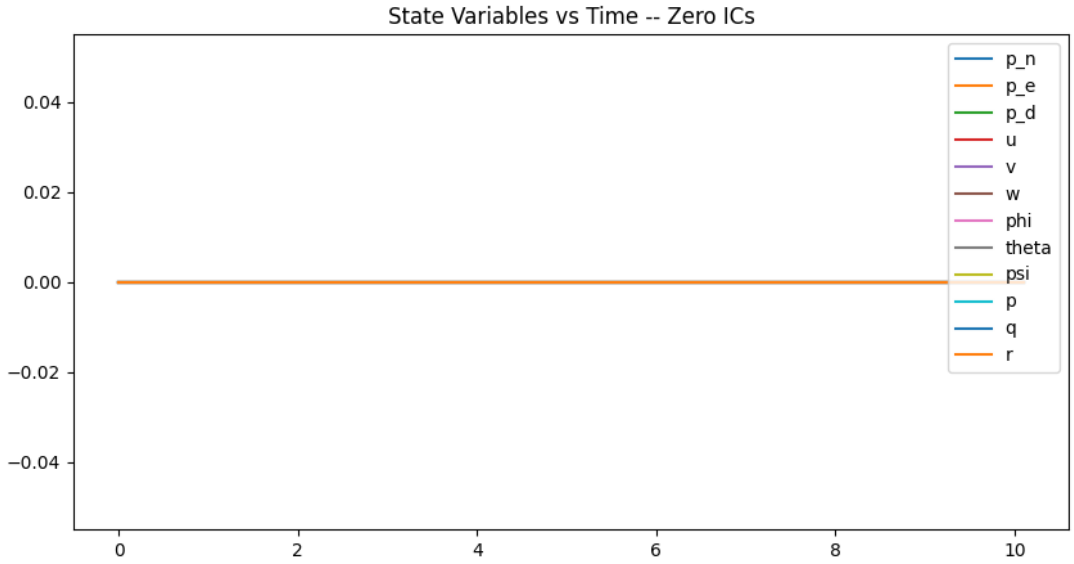
$$\therefore m_2 = \cancel{J_2 \dot{q}}^{\,0} + (J_1 - J_3) rp \approx -J_3 rp$$

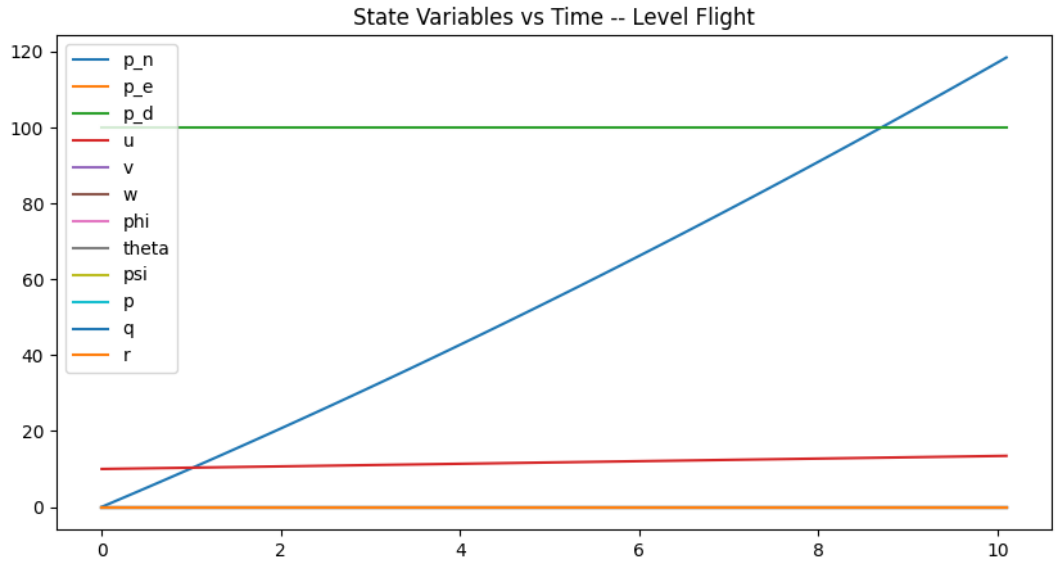
Since J_3 is "large" this indicates the pilot must apply a large negative pitch w/ the elevator to counteract the planes inertia

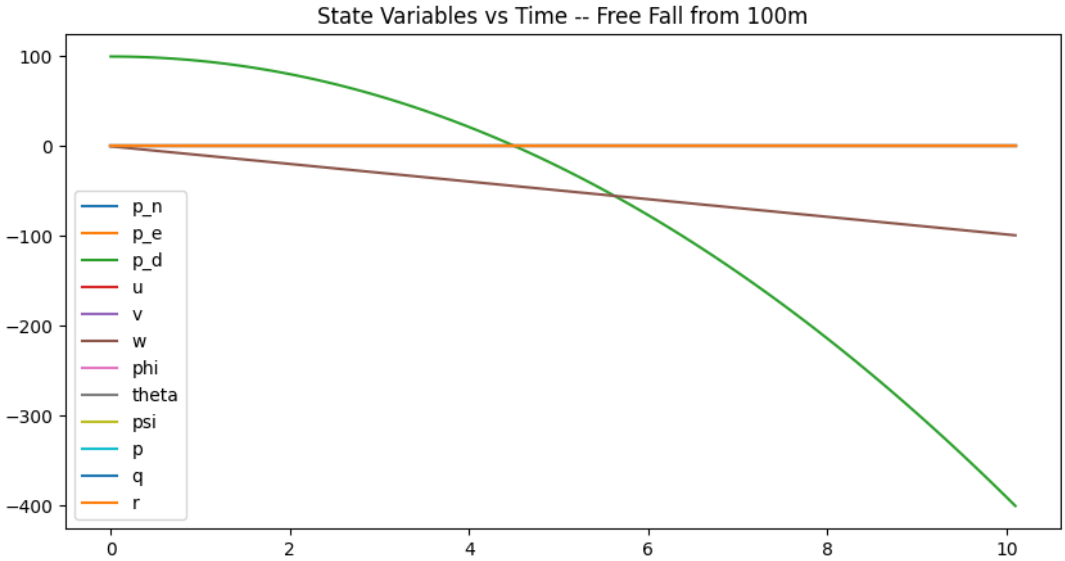
if $|p|$ is large, this further increases the required pitch moment m_2. If the elevator cannot accomodate the required pitch, $\dot{q}$ will physically not be able to equal zero & stationary roll would be impossible

Exercise 4:

Below are 3 simple test cases to test our simulator:







```
Exercise 2:
class RK4(Integrator): # Runge-Kutta 4th order
    def step(self, t, x, u):
        k1 = self.f(t, x, u)
        k2 = self.f(t + 0.5*self.dt, x + 0.5*self.dt*k1, u)
        k3 = self.f(t + 0.5*self.dt, x + 0.5*self.dt*k2, u)
        k4 = self.f(t + self.dt, x + self.dt*k3, u)
        return x + self.dt/6 * (k1 + 2*k2 + 2*k3 + k4)
```

```
Exercise 4:

# -*- coding: utf-8 -*-
"""
Created on Thu Feb  6 14:45:37 2025

@author: Adin Sacho-Tanzer & Gabriel Kret
The Propeller Heads
"""

import numpy as np
import matplotlib.pyplot as plt
import integrators_HW2 as intg
import Parameters_HW2 as p

#U is the input vector [f_x, f_y, f_z, l, m, n]
#u,v,w is the velocity in the body frame
#p,q,r is the angular velocity in the body frame
#p_n, p_e, p_d is the position in the NED frame
#phi, theta, psi is the euler angles


# define matrices and sub-equations


class rigid_body:
    def __init__(self, mass, J_xx, J_yy, J_zz, J_xz, S, b, c, S_prop, rho,
k_motor, k_T_p, k_Omega, e):
        self.mass = mass
        self.J_xx = J_xx
        self.J_yy = J_yy
        self.J_zz = J_zz
        self.J_xz = J_xz
        self.S = S
        self.b = b
        self.c = c
```

```
        self.S_prop = S_prop
        self.rho = rho
        self.k_motor = k_motor
        self.k_T_p = k_T_p
        self.k_Omega = k_Omega
        self.e = e

    def x_dot(self, t, x, U):
        #state: x = [p_n, p_e, p_d, u, v, w, phi, theta, psi, p, q, r]
        #inputs: U = [f_x, f_y, f_z, l, m, n]
        f_x, f_y, f_z, l, m, n = U

        p_n = x[0]
        p_e = x[1]
        p_d = x[2]
        u = x[3]
        v = x[4]
        w = x[5]
        phi = x[6]
        theta = x[7]
        psi = x[8]
        p = x[9]
        q = x[10]
        r = x[11]


        #forces and moments
        A = np.array([[np.cos(theta)*np.cos(psi),
np.sin(phi)*np.sin(theta)*np.cos(psi)-np.cos(phi)*np.sin(psi),
np.cos(phi)*np.sin(theta)*np.cos(psi)+np.sin(phi)*np.sin(psi)],
            [np.cos(theta)*np.sin(psi),
np.sin(phi)*np.sin(theta)*np.sin(psi)+np.cos(phi)*np.cos(psi),
np.cos(phi)*np.sin(theta)*np.sin(psi)-np.sin(phi)*np.cos(psi)],
            [-np.sin(theta), np.sin(phi)*np.cos(theta),
np.cos(phi)*np.cos(theta)]])

        B = np.array([[r*v - q*w],
                      [p*w - r*u],
                      [q*u - p*v]])

        C = 1/self.mass * np.array([[f_x],
                                    [f_y],
                                    [f_z]])

        D = np.array([[1, np.sin(phi)*np.tan(theta), np.cos(phi)*np.tan(theta)],
```

```
                        [0, np.cos(phi), -np.sin(phi)],
                        [0, np.sin(phi)/np.cos(theta),
np.cos(phi)/np.cos(theta)]])

        T = self.J_xx*self.J_zz - self.J_xz**2
        T1 = self.J_xz*(self.J_xx - self.J_yy + self.J_zz)/T
        T2 = (self.J_zz*(self.J_zz - self.J_yy) + self.J_xz**2)/T
        T3 = self.J_zz/T
        T4 = (self.J_zz - self.J_xx)/T
        T5 = (self.J_xx - self.J_yy)/T
        T6 = self.J_xz/T
        T7 = (self.J_zz - self.J_xx)/T
        T8 = self.J_xx*(self.J_xx - self.J_yy + self.J_zz)/T

        E = np.array([[T1*p*q - T2*q*r],
                      [T5*p*r - T6*(p**2 - r**2)],
                      [T7*p*q - T1*q*r]])


        F = np.array([[T3*l + T4*n],
                      [1/self.J_yy*m],
                      [T4*l + T8*n]])

        p_dot = A @ np.array([u,v,w])
        V_dot = B + C
        Angle_dot = D @ np.array([p,q,r])
        Omega_dot = E + F

        p_dot = p_dot.flatten()
        V_dot = V_dot.flatten()
        Angle_dot = Angle_dot.flatten()
        Omega_dot = Omega_dot.flatten()

        return np.concatenate((p_dot, V_dot, Angle_dot, Omega_dot), axis = 0)

    def simulate(self, x0, U, t_start, t_stop, dt=0.1):
        # maybe make U a function later
        # maybe have gravity later
        rk4_integrator = intg.RK4(dt, self.x_dot)

        t_history = [t_start]
        x_rk4_history = [x0]

        x_rk4 = x0
        t = t_start
```

```
        while t < t_stop:
            x_rk4 = rk4_integrator.step(t, x_rk4, U)
            t += dt
            t_history.append(t)
            x_rk4_history.append(x_rk4)

        return t_history, x_rk4_history

myPlane = rigid_body(p.mass, p.J_xx, p.J_yy, p.J_zz, p.J_xz, p.S, p.b, p.c,
p.S_prop, p.rho, p.k_motor, p.k_T_p, p.k_Omega, p.e)
t, x = myPlane.simulate(np.array([0,0,0,0,0,0,0,0,0,0,0,0]),
np.array([p.mass*2,0,0,0,0,0]), 0, 1, 0.1)
        #state: x = [p_n, p_e, p_d, u, v, w, phi, theta, psi, p, q, r]
        #inputs: U = [f_x, f_y, f_z, l, m, n]
plt.figure(figsize=(10, 5))
plt.plot(t, x)
plt.legend(["p_n", "p_e", "p_d", "u", "v", "w", "phi", "theta", "psi", "p", "q",
"r"])
plt.title("State Variables vs Time -- Taxiing and Takeoff")
plt.show()
```

Submit your solution on MS Teams as one (1) PDF file

Goals

- Use the equations of rigid-body motion to qualitatively explain aircraft behavior.
- Be able to implement the equations of motion in Python and simulate the motion of an aircraft subject to given forces and moments (next assignment...)
- Familiarize yourself with some custom aircraft frame and angle conventions.
- Be able to implement your own numerical integration method in Python.

Reading

- Beard and McLain (BM), *Small Unmanned Aircraft: Theory and Practice*: Chapter 3

Excercise 1: Conceptual questions

(a) Draw the wind triangle and explain briefly why $\mathbf{V}_g = \mathbf{V}_a + \mathbf{V}_w$.

(b) Sketch an aircraft and indicate the flight path angle and angle of attack.

(c) Provide the definition of the mass moment inertia of a rigid body around the 2-axis of the body frame, and explain its physical meaning.

(d) How many states are necessary to predict aircraft (rigid body) motion?

Excercise 2: Runge Kutta Implementation Download Python files for numerical integration from MS Teams (see hw01). For this question you will Implement a Runge-Kutta 4 integration routine.

(a) First, run the downloaded simulation and make sure it works.

(b) Next, implement your version of the Runge Kutta 4 integrator in the file `integrators.py`.

(c) Integrate the mass-spring system with both the `Heun` and `RungeKutta4` method. Experiment with suitable step sizes `dt`. Compare the numerical solutions with the analytical solution. Attach your plots and describe your findings.

Excercise 3: Inertia coupling Let the body-fixed frame of an aircraft be chosen such that it coincides with the principal axes, i.e. the mass moment of inertia matrix is diagonal:

$$\mathbf{J}_c^{bb} = \begin{pmatrix} J_1 & 0 & 0 \\ 0 & J_2 & 0 \\ 0 & 0 & J_3 \end{pmatrix}.$$

(a) Write down the three equations that follow from application of Euler's second law in the body frame, i.e.

$$\begin{aligned} m_1^b &= \dots \\ m_2^b &= \dots \\ m_3^b &= \dots, \end{aligned}$$

where m_i^b, $i = 1, 2, 3$, are the components of the total applied moment expressed in the body frame.

Using these expressions, we now study a vertical spin motion: the aircraft's velocity vector $\mathbf{V}_g$ points down and the angle of attack α is large. The aircraft rotates around the axis aligned with the velocity vector, i.e. $\boldsymbol{\omega}^b = (p, 0, r)^T$ (pitch velocity is zero).

(b) Sketch an aircraft with body frame, and indicate $\mathbf{V}_g$, $\boldsymbol{\omega}$, p and r.

(c) Assume *stationary* vertical spin motion. What is needed to maintain the spin motion? Which part of the aircraft can provide this? You may assume that $J_3 = J_1 + J_2$.

(d) For fighter jets with low aspect-ratio wings and a heavy engine in the fuselage (e.g. F-100 Super Sabre): $J_1 \ll J_2$, and $J_2 \approx J_3$. What kind of steering input is needed for a velocity vector roll (stationary)?[1] What happens when the roll velocity is large? Do you think a steady roll can be maintained?

Excercise 4: Implement the equations of motion given in the supplementary chapter 3 notes (see MS Teams) in Python code. The inputs are the forces and moments applied to the drone in the body frame. Parameters include the mass, the moments and products of inertia, and the initial conditions for each state. Use the parameters given in Beard's book, appendix E or make up your own. For possible templates / examples see the book's website Small Unmanned Aircraft: Theory and Practice.

[1] The fighter jet rotates around its velocity vector.