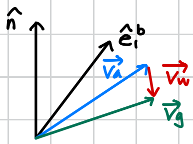
HW02

Calcifer Kim, Myisha Hassan, Wongee Hong

Github: https://github.com/freddyhong/Drone-Controls

#1.

(a) $\vec{V_g} = \vec{V_a} + \vec{V_w}$



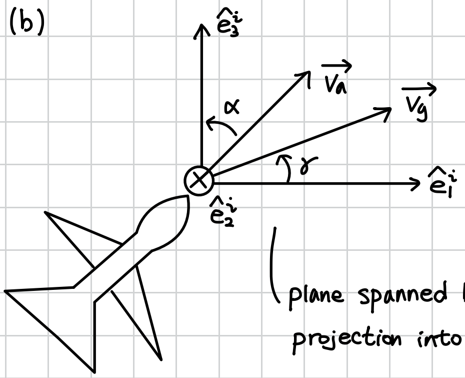
$\vec{V_a}$: velocity of plane w.r.t surrounding air $\vec{V_w}$: wind velocity w.r.t inertial frame

↳ In surrounding air perspective, plane flies in $\vec{V_g} - \vec{V_w}$ direction towards it.

$$\begin{cases} \vec{V_g} = \vec{V_a} + \vec{V_w} \\ \vec{V_a} = \vec{V_g} - \vec{V_w} \end{cases}$$

$\vec{V_g}$: velocity of plane w.r.t inertial frame

(b)



angle of attack α : LHR about $\hat{e}_2^b$

$\hat{e}_1^s$ aligns with projection of $\vec{V_a}$ onto plane spanned by $\hat{e}_1^b$ & $\hat{e}_3^b$

Flight Path Angle γ : angle btw horizontal plane ($\hat{e}_1^i$ & $\hat{e}_2^i$) and $\vec{V_g}$

plane spanned by $\hat{e}_1^i$ & $\hat{e}_3^i$ and $\hat{e}_1^b$ & $\hat{e}_3^b$

projection into vertical plane

(c) Moment of Inertia : how much angular acceleration by the applied torque

$$J = \begin{bmatrix} J_{xx} & J_{xy} & J_{xz} \\ J_{yx} & J_{yy} & J_{yz} \\ J_{zx} & J_{zy} & J_{zz} \end{bmatrix}$$

ex) $\hat{e}_y^b$ $\sum \vec{M}_y^b = J_{yx}\alpha_x + J_{yy}\alpha_y + J_{yz}\alpha_z$

$J_{\underline{a}\,\underline{b}}$ a: direction of moment

b: direction of angular acceleration that is affected

J_{yz} : how net moment applied in $\hat{e}_y^b$ affects the angular acceleration about $\hat{e}_z^b$ direction

(d) 3D rigid body motion $\begin{cases} \text{3 translational motion } (x, y, z) \\ \text{3 rotational motion } (\psi, \theta, \phi) \end{cases}$ 12 states

$$[x, \dot{x}, y, \dot{y}, z, \dot{z}, \psi, \dot{\psi}, \theta, \dot{\theta}, \phi, \dot{\phi}]^T$$

#3.

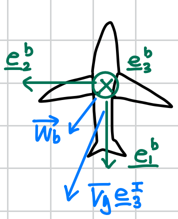
(a) $J_c^b = \begin{pmatrix} J_1 & 0 & 0 \\ 0 & J_2 & 0 \\ 0 & 0 & J_3 \end{pmatrix}$ Suppose $w_{b/i}^b = \begin{pmatrix} p \\ q \\ r \end{pmatrix}$

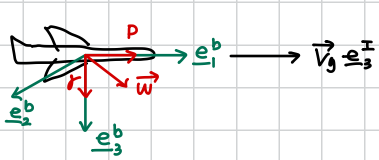
Euler's 2nd Law: $J_c^b \dot{w}_{b/i}^b \times (J_c^b w_{b/i}^b) = m^b$

$$\begin{pmatrix} J_1 & 0 & 0 \\ 0 & J_2 & 0 \\ 0 & 0 & J_3 \end{pmatrix}\begin{pmatrix} \dot{p} \\ \dot{q} \\ \dot{r} \end{pmatrix} + \begin{pmatrix} p \\ q \\ r \end{pmatrix} \times \begin{pmatrix} J_1 p \\ J_2 q \\ J_3 r \end{pmatrix} = m^b \Rightarrow \begin{pmatrix} J_1 \dot{p} \\ J_2 \dot{q} \\ J_3 \dot{r} \end{pmatrix} + \begin{pmatrix} J_3 qr - J_2 qr \\ -J_3 pr + J_1 pr \\ J_2 pq - J_1 pq \end{pmatrix} = m^b = \begin{pmatrix} m_1^b \\ m_2^b \\ m_3^b \end{pmatrix}$$

$$m_1^b = J_1 \dot{p} + J_3 qr - J_2 qr$$
$$m_2^b = J_2 \dot{q} - J_3 pr + J_1 pr$$
$$m_3^b = J_3 \dot{r} + J_2 pq - J_1 pq$$

(b) Assumption : vertical spin V_g points down, α is large, $w_{b/i}^b = (p, 0, r)^T$





(C) Stationary Vertical Spin → no translational motions $\quad q \to 0 \quad \dot{q} \to 0$

$\dot{p}$ roll
$\dot{q}$ pitch
$\dot{r}$ yaw

$$\begin{cases} m_1^b = J_1 \dot{p} & \text{aileron} \\ m_2^b = -J_3 pr + J_1 pr = -J_2 pr & \text{elevator} \\ m_3^b = J_3 \dot{r} & \text{rudder} \end{cases}$$

For Spin Motion, we need m_2^b.

negative pitch
moments by ailerons & rudders that are proportional to $\dot{\varphi}$ & $\dot{r}$
$\bar{M} = I\alpha$

(d) Velocity vector roll is spinning about velocity vector
$(J_1 << J_2 \ , \ J_2 \approx J_3)$

$\rightarrow w^i_{b/i} = (0 \ 0 \ w)^T$ using θ, ϕ, ψ $\rightarrow w^b_{b/i} = [-\sin\theta w, \sin\phi\cos\theta w, \cos\phi\cos\theta w]$
$= [p, q, r]$

$\rightarrow m_1^b \approx 0$
$m_2^b = J_2 \dot{q} - J_2 pr$
$m_3^b = J_3 \dot{r} + J_2 pq$

Need Pitch & Yaw Moments for steering inputs.
When velocity roll is high, $m_1 b$ will no longer be 0
↳ nonlinear effects cause an unsteady flight.

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.

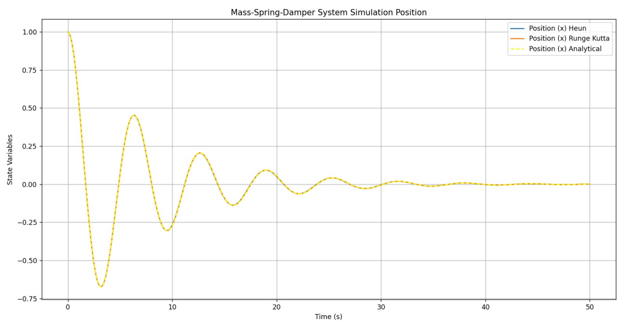
(a) First, run the downloaded simulation and make sure it works.

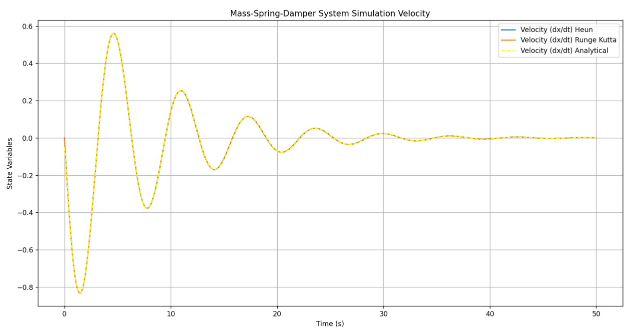
(b) Next, implement your version of the Runge Kutta 4 integrator in the file `integrators.py`.

(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.

At higher dt values, the RungeKutta method is more accurate to the analytical solution whereas the Heun method starts to oscillate incorrectly.
At dt=0.1, both numerical methods are very close to analytical, but when zooming in the RK4 method better.
At smaller step sizes, RK4 still remains more accurate!

```python
class Integrator:
    """Integrator for a system of first-order ordinary differential equations
    of the form \dot x = f(t, x, u).
    """
    def __init__(self, dt, f):
        self.dt = dt
        self.f = f

    def step(self, t, x, u):
        raise NotImplementedError

class Euler(Integrator):
    def step(self, t, x, u):
        return x + self.dt * self.f(t, x, u)

class Heun(Integrator):
    def step(self, t, x, u):
        intg = Euler(self.dt, self.f)
        xe = intg.step(t, x, u) # Euler predictor step
        return x + 0.5*self.dt * (self.f(t, x, u) + self.f(t+self.dt, xe, u))

class RungeKutta(Integrator):
    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)
```





Q4.

Implementing EoM with inputs of variables such as mass, J_x, J_y, J_z, J_{xz} and matrices such as initial states, Forces and Moments. Then this function return array of $\{\dot{u}, \dot{v}, \dot{w}, \dot{p}, \dot{q}, \dot{r}\}$

```
def equations_of_motion(t, state, mass, Jx, Jy, Jz, Jxz, forces, moments):

    u, v, w, p, q, r = state

    fx, fy, fz = forces
    Mx, My, Mz = moments

    u_dot = r * v - q * w + fx / mass
    v_dot = p * w - r * u + fy / mass
    w_dot = q * u - p * v + fz / mass

    Gamma1 = (Jxz * (Jx - Jy + Jz)) / (Jx * Jz - Jxz**2)
    Gamma2 = (Jz * (Jz - Jy) + Jxz**2) / (Jx * Jz - Jxz**2)
    Gamma3 = Jz / (Jx * Jz - Jxz**2)
    Gamma4 = Jxz / (Jx * Jz - Jxz**2)
    Gamma5 = (Jz - Jx) / Jy
    Gamma6 = Jxz / Jy
    Gamma7 = ((Jx - Jy) * Jx + Jxz**2) / (Jx * Jz - Jxz**2)
    Gamma8 = Jx / (Jx * Jz - Jxz**2)

    p_dot = Gamma1 * p * q - Gamma2 * q * r + Gamma3 * Mx + Gamma4 * Mz
    q_dot = Gamma5 * p * r - Gamma6 * (p**2 - r**2) + My / Jy
    r_dot = Gamma7 * p * q - Gamma1 * q * r + Gamma4 * Mx + Gamma8 * Mz

    return np.array([u_dot, v_dot, w_dot, p_dot, q_dot, r_dot])
```

```
t_span = (0, 10)
t_eval = np.linspace(0, 10, 100)

sol = solve_ivp(equations_of_motion, t_span, initial_state, t_eval=t_eval, args=(mass, Jx, Jy, Jz, Jxz, forces, moments))
```

Then the solution is obtained through scipy.integrator.solve_ivp function.

With given constant values for mass, J_x, J_y, J_z, J_{xz} on the textbook and arbitrary values for initial state, forces and moments, we got the result.

```
initial_state = np.array([0, 0, 0, 0, 0, 0])

mass = 11  #kg*m^2
Jx = 0.824  #kg*m^2
Jy = 1.135  #kg*m^2
Jz = 1.759  #kg*m^2
Jxz = 0.12  #kg*m^2
forces = np.array([10, 0, -9.81 * mass])  # These are example forces
moments = np.array([0, 0, 5])  # These are example moments
```