ME 493 Drone Control Homework 2

Jack Donnellan (donnel2@cooper.edu)

September 28, 2020

1. Download Python files for numerical integration from MS Teams (see week 4 > Files > hw02). For this question you will Implement a Runge-Kutta 4 integration routine.

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

Script works as expected.

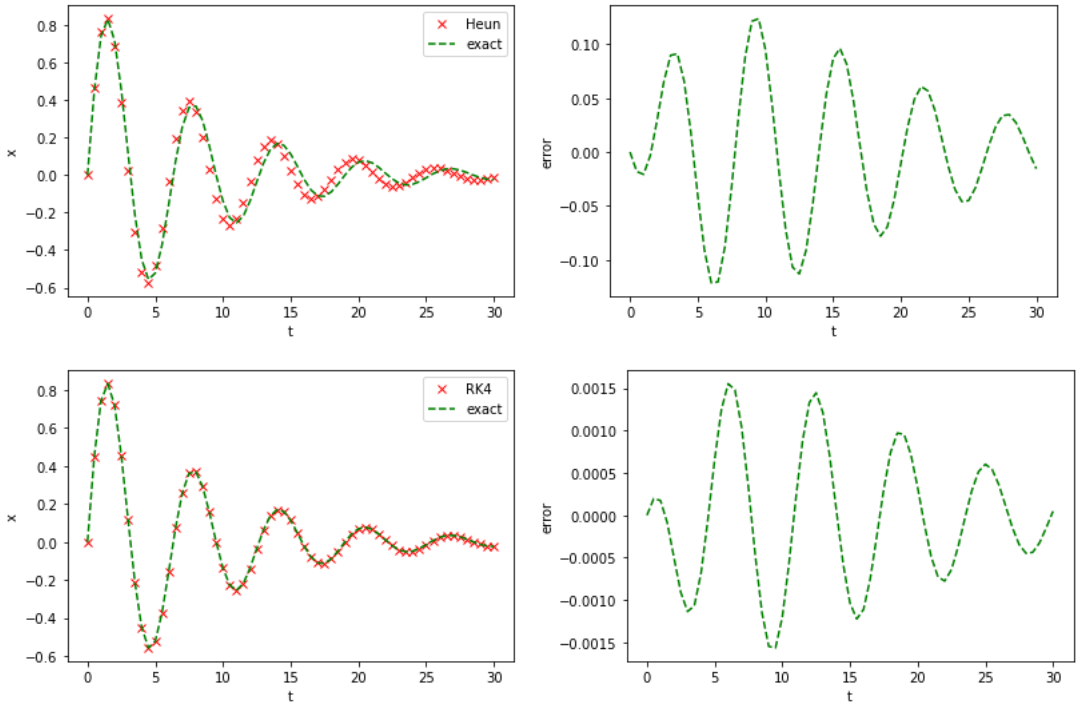
(b) Next, implement your version of the Runge Kutta 4 integrator in the file integrators.py

My Runge Kutta 4 follows the equations outlined on Wikipedia[1]. My code is attached in Appendix A.

(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.

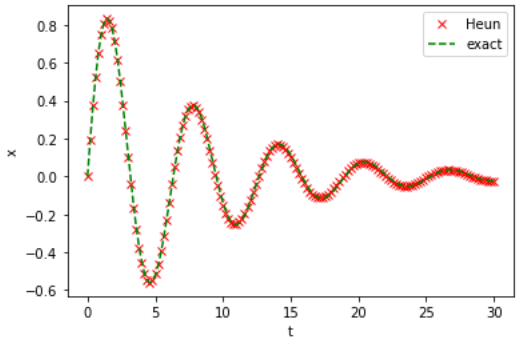
For my step-size experimentation, I used $\vec{x}_0 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$

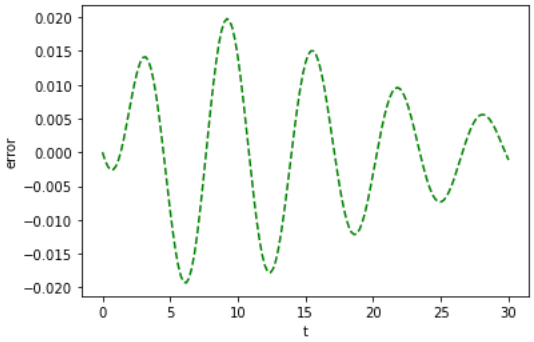
dt = 0.5

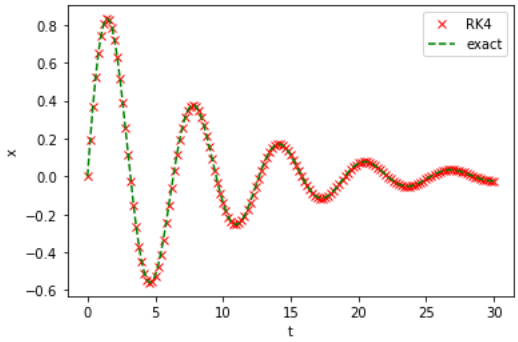


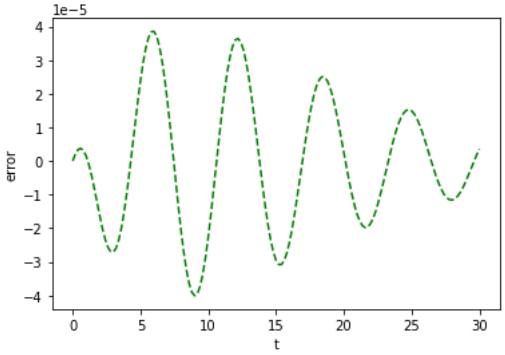
[1] https://en.wikipedia.org/wiki/Runge%E2%80%93Kutta_methods

dt = 0.2

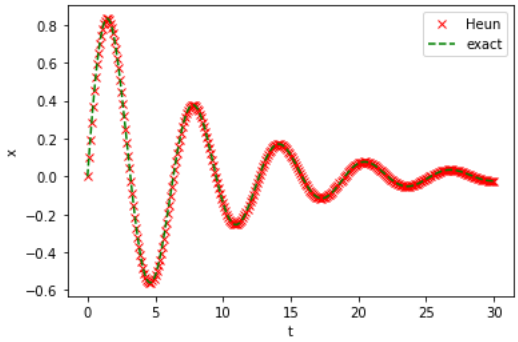

Heun
exact
x
t

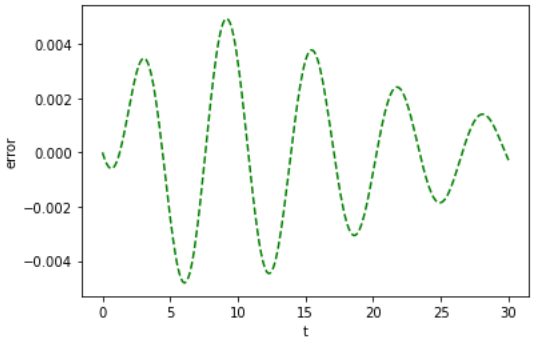

error
t

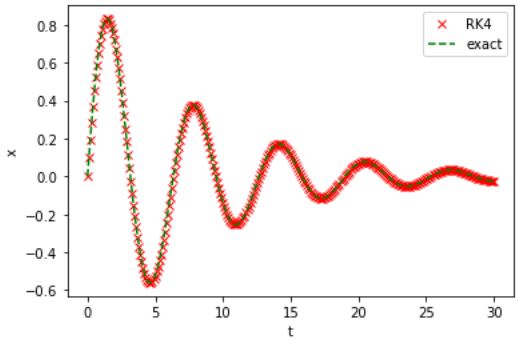

RK4
exact
x
t

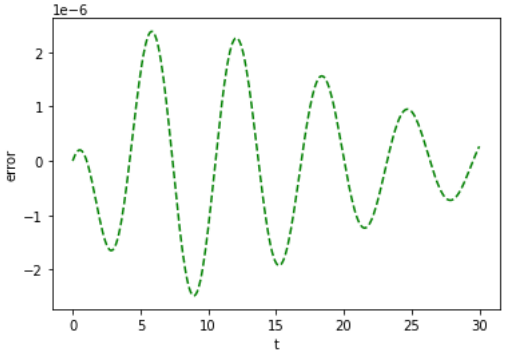

1e−5
error
t

dt = 0.1

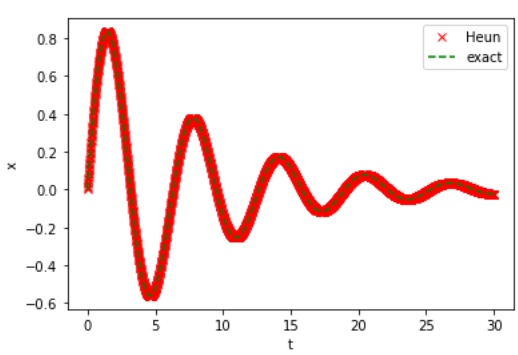

Heun
exact
x
t

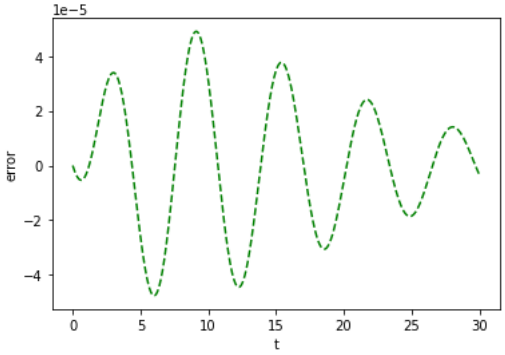

error
t

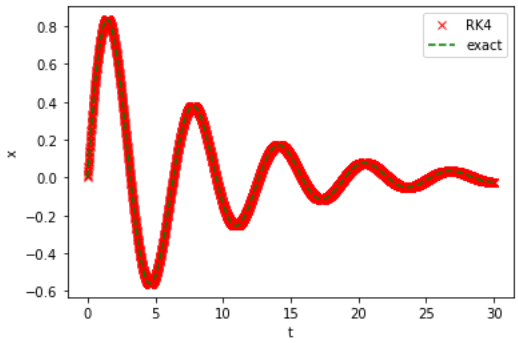

RK4
exact
x
t

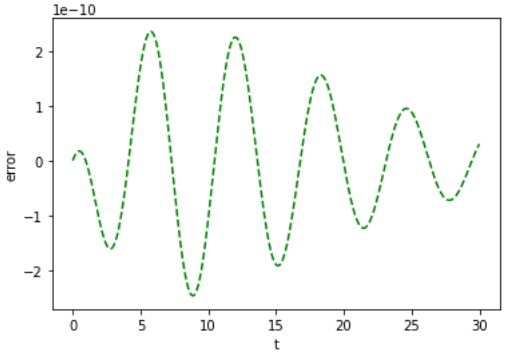

1e−6
error
t

dt = 0.01

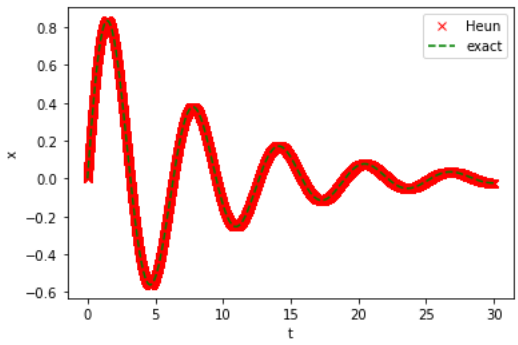


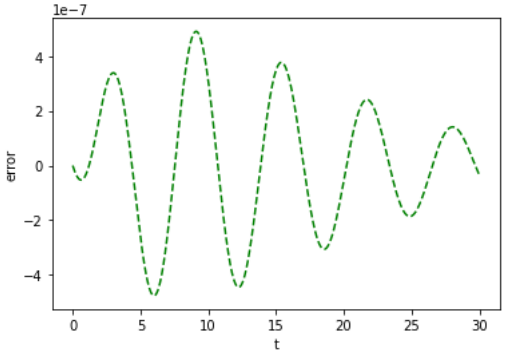


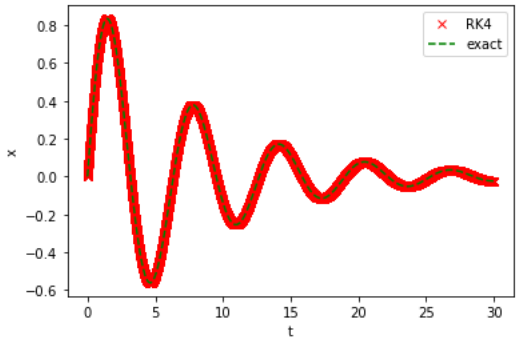


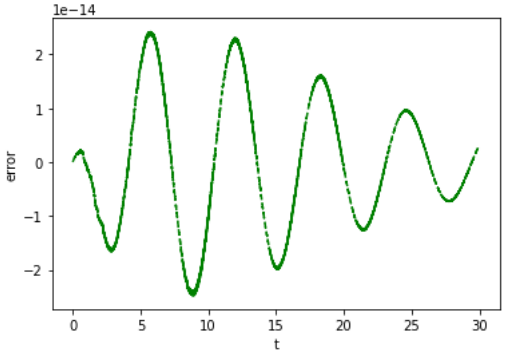


dt = 0.001









As the plots show, the RK4 method tends to give much smaller errors than the Heun method. For example, to match the dt = 0.5 RK4 error magnitudes with Heun, you'd need to use dt $\approx$ 0.1. By the time you get to dt = 0.1 for RK4, you'd need to use Heun with dt between 0.01 and 0.001, which starts to increase sim time. By the time you get to dt = 0.01, RK4 errors are about five orders of magnitude smaller than the errors from the Heun method.

2. 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:

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

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

Given:

- $\frac{\partial \boldsymbol{h}^b}{\partial t_b} + \boldsymbol{\omega}_{b/i}^b \times \boldsymbol{h}^b = \boldsymbol{m}^b$ [2]
- $\boldsymbol{J}_c^{bb}$ from the problem description.

Solution:

Because J is constant in the body frame, we have:

$$\frac{\partial \boldsymbol{h}^b}{\partial t_b} = \boldsymbol{J}_c^{bb} \frac{\partial \boldsymbol{\omega}_{b/i}^b}{\partial t_b}$$

Substituting and solving gives:

$$\boldsymbol{J}_c^{bb} \frac{\partial \boldsymbol{\omega}_{b/i}^b}{\partial t_b} + \boldsymbol{\omega}_{b/i}^b \times \boldsymbol{J}_c^{bb} \boldsymbol{\omega}_{b/i}^b = \boldsymbol{m}^b$$

$$\boldsymbol{m}^b = \begin{pmatrix} J_1 & 0 & 0 \\ 0 & J_2 & 0 \\ 0 & 0 & J_3 \end{pmatrix} \begin{pmatrix} \dot{p} \\ \dot{q} \\ \dot{r} \end{pmatrix} + \begin{pmatrix} p \\ q \\ r \end{pmatrix} \times \begin{pmatrix} J_1 & 0 & 0 \\ 0 & J_2 & 0 \\ 0 & 0 & J_3 \end{pmatrix} \begin{pmatrix} p \\ q \\ r \end{pmatrix}$$

$$\boldsymbol{m}^b = \begin{pmatrix} J_1\dot{p} \\ J_2\dot{q} \\ J_3\dot{r} \end{pmatrix} + \begin{pmatrix} p \\ q \\ r \end{pmatrix} \times \begin{pmatrix} J_1 p \\ J_2 q \\ J_3 r \end{pmatrix}$$

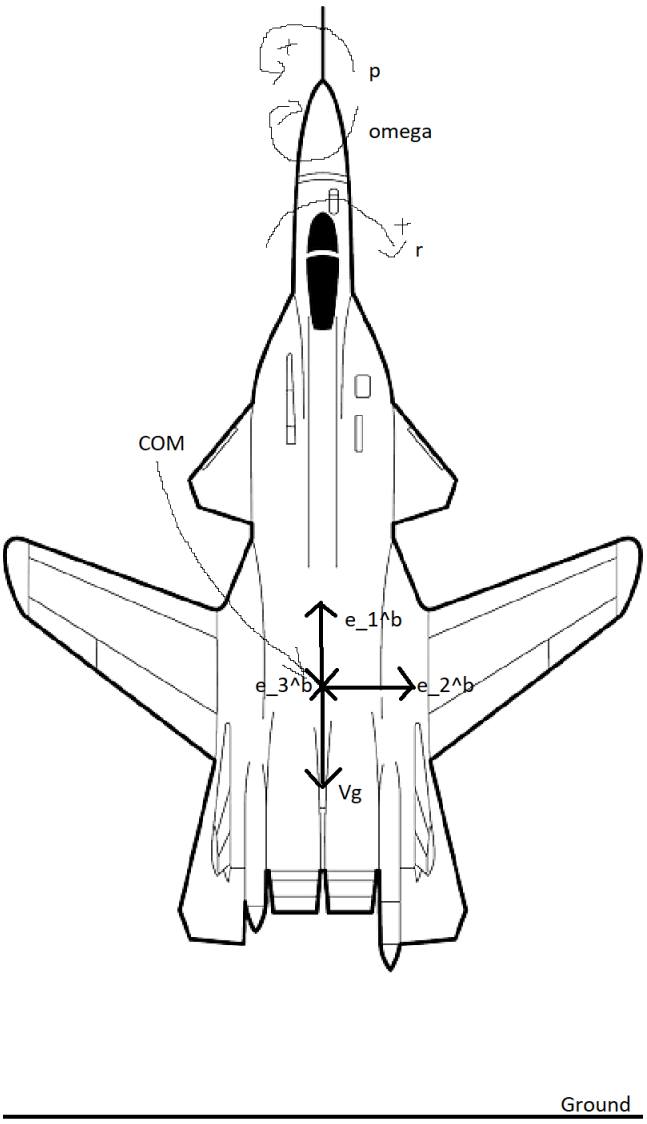
$$\begin{pmatrix} m_1^b \\ m_2^b \\ m_3^b \end{pmatrix} = \begin{pmatrix} J_1\dot{p} + qr(J_3 - J_2) \\ J_2\dot{q} + rp(J_1 - J_3) \\ J_3\dot{r} + pq(J_2 - J_1) \end{pmatrix}$$

[2] Beard and McLain, "Small Unmanned Aircraft: Theory and Practice", pg.33, Eqn. 3.8

Using these expressions, we now study a vertical spin motion. Let the aircraft velocity Vg point down (the angle of attack α is very large), and also the angular velocity ω . Assume that the the pitch velocity is zero, $q = 0$.

(b) Sketch an aircraft with body frame, and indicate V_g, ω, p and r

Base picture taken from www.kindpng.com[3]



[3]https://www.kindpng.com/imgv/obxmRm_fighter-plane-sketch-fighter-jet-plane-drawing-hd/

(c) Assume *stationary* vertical spin motion. What is needed to maintain the spin motion? Which part of the aircraft can provide this?

Taking stationary vertical spin motion to mean $\boldsymbol{V}_g = \mathbf{0}$, the ailerons which would typically roll the vehicle would have no control power (neglecting the small but potentially non-zero airspeed over the inner-wing which may arise from a propeller). Therefore to maintain spinning motion we would need some source of angular momentum on the aircraft in the opposite direction of the desired roll rate.

$$J_{prop}\omega_{prop} = -J_1 p$$

This can be provided by the propeller in a single-prop UAV configuration, or by the engine core on larger aircraft. An extreme example would be a the yawing motion of conventional helicopter with a failed tail-rotor.

To maintain this orientation, thr pitch and yaw axis would have to be stabilized. The elevator and rudder in a stationary vertical spin would also have negligible to no control power. For this configuration to be stabilized (at the relatively small roll rate induced by a propeller/engine core assuming $J_1 >> J_{prop}$), thrust vectoring would likely be required.

(d) Bonus For fighter jets with low aspect ratio wings and a heavy engine in the fuselage (e.g. F-100 Super Sabre): $J_1 << J_2$, and $J_2 \approx J_3$. What happens when the roll velocity is large? Do you think steady vertical spin can be maintained?

Assume:

- No net external torques applied.
- $q, r << p$

Given:

- Solution to part(a)
- $J_1 << J_2$, and $J_2 \approx J_3$

Solution:

Given our assumption of no net external torques and our solution to part (a), we have:

$$\begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} J_1\dot{p} + qr(J_3 - J_2) \\ J_2\dot{q} + rp(J_1 - J_3) \\ J_3\dot{r} + pq(J_2 - J_1) \end{pmatrix}$$

$$\begin{pmatrix} J_1\dot{p} \\ J_2\dot{q} \\ J_3\dot{r} \end{pmatrix} = \begin{pmatrix} qr(J_2 - J_3) \\ rp(J_3 - J_1) \\ pq(J_1 - J_2) \end{pmatrix}$$

Taking the time derivative of this expression we get:

$$\begin{pmatrix} J_1\ddot{p} \\ J_2\ddot{q} \\ J_3\ddot{r} \end{pmatrix} = \begin{pmatrix} (\dot{q}r + q\dot{r})(J_2 - J_3) \\ (\dot{r}p + r\dot{p})(J_3 - J_1) \\ (\dot{p}q + p\dot{q})(J_1 - J_2) \end{pmatrix}$$

From $J_1 << J_2$, $J_2 \approx J_3$, and $q, r << p$ we can say $\dot{p}, \ddot{p} \approx 0$

Substituting this into our expressions gives:

$$\begin{pmatrix} \dot{p} \\ \dot{q} \\ \dot{r} \end{pmatrix} \approx \begin{pmatrix} 0 \\ \frac{rp(J_3)}{J_2} \\ \frac{-pq(J_2)}{J_3} \end{pmatrix} \approx \begin{pmatrix} 0 \\ rp \\ -pq \end{pmatrix}$$

$$\begin{pmatrix} \ddot{p} \\ \ddot{q} \\ \ddot{r} \end{pmatrix} \approx \begin{pmatrix} 0 \\ \frac{\dot{r}pJ_3}{J_2} \\ \frac{-p\dot{q}J_2}{J_3} \end{pmatrix} \approx \begin{pmatrix} 0 \\ p\dot{r} \\ -p\dot{q} \end{pmatrix}$$

Substituting $\dot{q}$ and $\dot{r}$ from our first expression into our second gives:

$$\begin{pmatrix} \ddot{p} \\ \ddot{q} \\ \ddot{r} \end{pmatrix} \approx \begin{pmatrix} 0 \\ -p^2q \\ -p^2r \end{pmatrix} = \begin{pmatrix} 0 \\ -Cq \\ -Cr \end{pmatrix}$$

$$C = p^2 > 0$$

$\implies$ The vertical spin is stable, with stability increasing quickly as roll rate increases.

3. Loosely defined since I don't have my own Python template ready yet, and I don't like the one provided by Beard. I will share my code with you later this week.. Here is your challenge: 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. For possible templates / examples see the book's website Small Unmanned Aircraft: Theory and Practice.

I implemented the basic flight sim as follows:

- An "aircraft" class (flight_sim.py) which takes aircraft parameters (right now just mass and the J matrix) and contains the 6DOF EOMs. It also contains a "step" method to give the state of the aircraft after one time step.

- A run script (run_flight_sim.py), where initial conditions and sim parameters are defined. The aircraft step method is used in a loop to give the state evolution.

Code for the flight sim is attached as Appendix B and can be downloaded at my course Github[4]. I've tested it using easily hand-computed cases with and without forces/moments and it has passed thus far.

[4]https://github.com/donnel2-cooper/drone_control/tree/master/Homework/Homework_2/python

Appendix A: RK4 Implementation

```
class RungeKutta4(Integrator):
    def step(self, t, x, u):
        k1 = self.f(t,x,u)
        k2 = self.f(t+(self.dt/2),x+(self.dt*k1)/2,u)
        k3 = self.f(t+(self.dt/2),x+(self.dt*k2)/2,u)
        k4 = self.f(t+self.dt,x+self.dt*k3,u)
        return x + (1/6)*self.dt*(k1+2*k2+2*k3+k4)
```

Appendix B: Flight Simulation Implementation

flight_sim.py

```
import numpy as np
import integrators as intg

## Helper Functions
def sind(angle):
    #Compute sin of angle in degrees as composition of np functions
    return np.sin(np.deg2rad(angle))
def cosd(angle):
    #Compute cos of angle in degrees as composition of np functions
    return np.cos(np.deg2rad(angle))
def tand(angle):
    #Compute tan of angle in degrees as composition of np functions
    return np.tan(np.deg2rad(angle))

class aircraft():
    def __init__(self,mass,J):
        self.mass = mass
        self.J = J

    ## Define EOM's as diff eq's
    #  xdot = f(t, x, u),
    #   t: time
    #                                0   1   2  3  4  5    6      7    8  9 10 11
    #   x: state vector in book, [pn, pe, pd, u, v, w, phi, theta, psi, p, q, r]
    #       pn, pe, pd iniertial frame positions
    #       u, v, w ground speeds
    #       phi, theta, psi Euler angles (in degrees)
    #       p, q, r angular rates (in dps)
    #
    #                0  1  2  3  4  5
    #   u: input, [fx,fy,fz, l, m, n]
    def f_pos(self,t, x, u):
        # Equation 3.14
        R = np.matrix([[cosd(x[7])*cosd(x[8]),\
                        sind(x[6])*sind(x[7])*cosd(x[8])-cosd(x[6])*sind(x[8]),\
                        cosd(x[6])*sind(x[7])*cosd(x[8])+sind(x[6])*sind(x[8])],\
                       [cosd(x[7])*sind(x[8]),\
                        sind(x[6])*sind(x[7])*sind(x[8])+cosd(x[6])*cosd(x[8]),\
                        cosd(x[6])*sind(x[7])*sind(x[8])-sind(x[6])*cosd(x[8])],\
                       [-sind(x[7]),\
                        sind(x[6])*cosd(x[7]),\
                        cosd(x[6])*cosd(x[7])]])
        substate = np.array([[x[3]],[x[4]],[x[5]]])
        mat_mult_result = np.matmul(R,substate)
        return np.array([float(mat_mult_result[0]),float(mat_mult_result[1]),\
                         float(mat_mult_result[2]),0,0,0,0,0,0,0,0,0])

    def f_vel(self,t, x, u):
```

```
        # Equation 3.15
        f_over_m = np.array([[u[0]/self.mass],[u[1]/self.mass],[u[2]/self.mass]])
        first_term = np.array([[x[11]*x[4]-x[10]*x[5]],\
                               [x[9]*x[5]-x[11]*x[3]],\
                               [x[10]*x[3]-x[9]*x[4]]])
        summation = np.add(first_term,f_over_m)
        return np.array([0,0,0,float(summation[0]),float(summation[1]),\
                         float(summation[2]),0,0,0,0,0,0])

    def f_euler(self,t,x,u):
        # Equation 3.16
        R = np.matrix([[1,\
                        sind(x[6])*tand(x[7]),\
                        cosd(x[6])*tand(x[7])],\
                       [0,\
                        cosd(x[6]),\
                        -sind(x[6])],\
                       [0,\
                        sind(x[6])/cosd(x[7]),\
                        cosd(x[6])/cosd(x[7])]])
        substate = np.array([[x[9]],[x[10]],[x[11]]])
        mat_mul_result = np.matmul(R, substate)
        return np.array([0,0,0,0,0,0,float(mat_mul_result[0]),\
                         float(mat_mul_result[1]),float(mat_mul_result[2]),0,0,0])

    def f_rate(self,t,x,u):
        # Equations 3.13
        gamma = self.J[0,0]*self.J[2,2]-(-self.J[0,2])**2
        gamma_1 = ((-self.J[0,2])*(self.J[0,0]-self.J[1,1]+self.J[2,2]))/gamma
        gamma_2 = (self.J[2,2]*(self.J[2,2]-self.J[1,1])+(-self.J[0,2])**2)/gamma
        gamma_3 = self.J[2,2]/gamma
        gamma_4 = -self.J[0,2]/gamma
        gamma_5 = (self.J[2,2]-self.J[0,0])/self.J[1,1]
        gamma_6 = -self.J[0,2]/self.J[1,1]
        gamma_7 = ((self.J[0,0]-self.J[1,1])*self.J[0,0]+(-self.J[0,2])**2)/gamma
        gamma_8 = self.J[0,0]/gamma
        #Equation 3.17
        first_term = np.array([[gamma_1*x[9]*x[10]-gamma_2*x[10]*x[11]],\
                               [gamma_5*x[9]*x[11]-gamma_6*(x[9]**2-x[11]**2)],\
                               [gamma_7*x[9]*x[10]-gamma_1*x[10]*x[11]]])
        second_term = np.array([[gamma_3*u[3]+gamma_4*u[5]],\
                                [u[4]/self.J[1,1]],\
                                [gamma_4*u[3]+gamma_8*u[5]]])
        summation = np.add(first_term,second_term)
        return np.array([0,0,0,0,0,0,0,0,0,float(summation[0]),\
                         float(summation[1]),float(summation[2])])

    def step(self,t,x,u,dt = 0.01):
        # Integrate vehicle state forward one time step dt
        inertial_pos_intg = intg.RungeKutta4(dt,self.f_pos)
        velocity_intg = intg.RungeKutta4(dt,self.f_vel)
```

```
        euler_angles_intg = intg.RungeKutta4(dt, self.f_euler)
        rates_intg = intg.RungeKutta4(dt, self.f_rate)
        inertial_pos = inertial_pos_intg.step(t,x,u)
        velocity = velocity_intg.step(t,x,u)
        euler_angles = euler_angles_intg.step(t,x,u)
        rates = rates_intg.step(t,x,u)
        return np.concatenate((inertial_pos[:3], velocity[3:6],\
                               euler_angles[6:9], rates[9:]))
```

run_flight_sim.py

```
import numpy as np
import matplotlib.pyplot as plt
import flight_sim as fs

## Aircraft Parameters (Aerosonde UAV)
mass = 13.5 #kg
J = np.matrix([[0.8244,0,-0.1204],[0,1.135,0],[-0.1204,0,1.759]]) #kg*m^2

## Initialize arone as aircraft instance
drone = fs.aircraft(mass,J)

## Sim
# Sim Parameters
t0 = 0; tf = 20; dt = 0.01; n = int(np.floor(tf/dt));
#Initial State and Initial Input
x0 = np.array([0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0])
u0 = np.array([float(mass),0.0,0.0,0.0,0.0,0.0])

x = x0
u = u0
t = t0
t_history = [0]
x_history = [x]

# Sim Loop
for i in range(n):
    # Step sim
    x = drone.step(t,x,u,dt)

    # Step time
    t = (i+1) * dt

    # Append State and Time to History
    t_history.append(t)
    x_history.append(x)

print(x)
```