Brandon White

MAE 5010 – Autopilot Design and Test

Homework #1 (Due 08/29/2019)

==

2. Write a function that converts between quaternions and 3-2-1 Euler angles and one that converts back. These should look like EP = Euler3212EP([heading,pitch,roll]) EA = EP2Euler321([q0,q1,q2,q3]) Include an example of running your code to convert [ψ, θ, φ] = [150◦ , 15◦ , −30◦] to quaternions and ~q = [0.82205,0.26538,0.05601,0.50066] to Euler angles.

See Appendix for angle conversion code.

```
>>> import white_brandon_HW1 as HW1
>>> EP = HW1.Euler3212EP([150, 15, -30])
>>> EP
[0.21523, -0.1882, -0.21523, 0.93377]
>>> angles = HW1.EP2Euler321([0.82205, 0.26538, 0.05601, 0.50066])
>>> angles
[60.0, -10.0, 30.0]
```

Figure 1. Command Line I/O for Problem 2

3. Implement the kinematics and dynamics equations (using the quaternion formulation) in an integrator that takes in forces and moments. You should have a function that takes in state and returns xdot = derivatives(self, state, FM, MAV) where state = [pn,pe,pd, u,v,w, e0,e1,e2,e3, p,q,r] FM = [Fx,Fy,Fz, Ell,M,N] The function will contain computations of each state derivative, e.g.,

% position kinematics
  pn_dot = pe_dot = pd_dot =
% position dynamics
  u_dot = v_dot = w_dot =
% rotational kinematics
  e0_dot = e1_dot = 1 e2_dot = e3_dot =
% rotational dynamics
  p_dot = q_dot = r_dot =
% collect all the derivaties of the states
  xdot = [pn_dot; pe_dot; pd_dot; u_dot; v_dot; w_dot;... e0_dot; e1_dot; e2_dot; e3_dot; p_dot; q_dot; r_dot];

Include a gravitational force vector mg ˆfz. To make this into a simulation, you need to add initial conditions, vehicle parameters, forces and moments, and then integrate it. You'll add the first two in the step below.

See Appendix for derivative scripts.

```python
def update_FM(self, t):
    from math import sin, cos
    from white_brandon_HW1 import EP2Euler321

    #Angularize Gravity
    angles = EP2Euler321(self.state0[6:10])
    Fg = f2b(angles, [0, 0, 32.2*self.mass])

    #All Other Forcing Functions
    for i in range(6):
        try:
            self.FM[i] = self.FMeq[i](t)
        except:
            self.FM[i] = self.FMeq[i]

    #Add in Gravity
    self.FM[0] += Fg[0]
    self.FM[1] += Fg[1]
    self.FM[2] += Fg[2]

    return self.FM
```

Figure 2. Gravity Force Code

4. Select a candidate air vehicle you may use in your research, and approximate the mass and inertia of this vehicle. Define these in a separate file as terms like MAV.mass, MAV.Ix, MAV.Iy, MAV.Iz MAV.Ixz, and also define initial conditions for position, orientation, and rates. Move your definition of gravitational constant g here. Use consistent units.

```python
class MAV:
    def __init__(self, aircraft = "None"):
        #All units listed in English units as denoted
        self.name = aircraft
        self.mass = 10  # Mass (lbf)
        #Inert = [Ixz, Ix, Iy, Iz]
        self.inert = [20, 10, 10, 10]  # Moment of Inertia (lbf*ft^2)
        self.gravity_needed = False
        #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]
            #Gravity ONLY in base model
        self.FMeq = [0, 0, (lambda t: 32.2*self.mass), 0, 0, 0]
```

Figure 3. MAV Class Object Code

The rate of descent was verified for the gravity only case through solving the ODE via WolframAlpha.com.

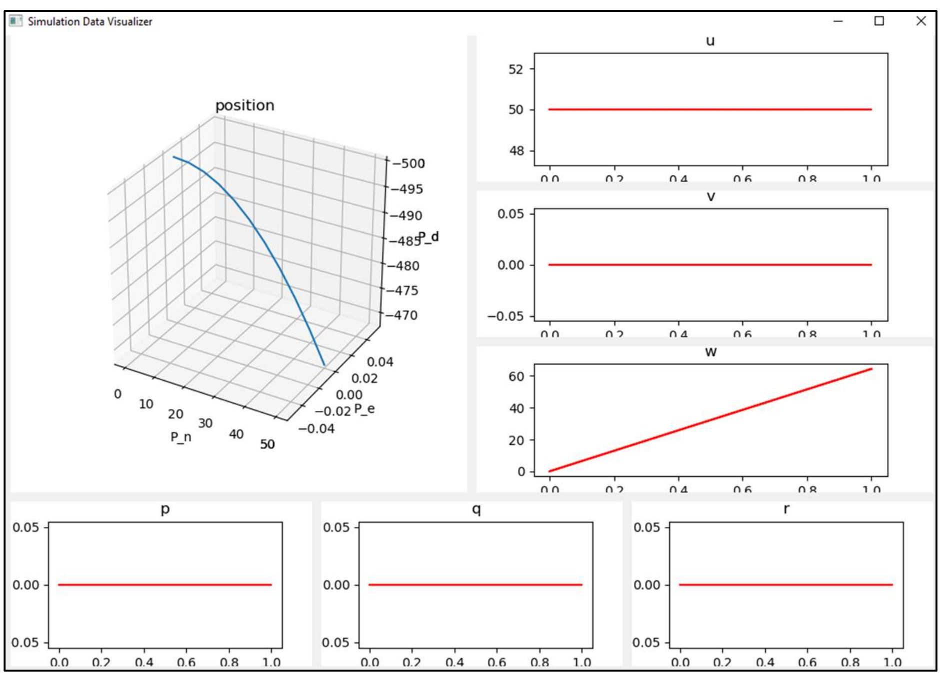

Figure 4. Gravity Only Simulation Results

5. Choose an integrator (you could try a Runge-Kutta integrator, or ode23), and connect it to the above dynamics function, having it read in the parameters to form a simple dynamics simulator. Simulate this vehicle from an initial condition of x, y, z = [100, 200, −500], ψ, θ, φ = [90, 15, 20]◦ (1) with F and M set to zero. Plot the positions, Euler angles, and rates. Does this make sense? Do you need to choose a different integrator? Simulate the system from the same initial conditions but use F = [sin(t),0,0] and M = [0,1e-4,0] Include plots of the output from these two cases and label your axes.

Selected integrator: scipy.integation.odeint()

The overall flight patterns make sense. As M is made non-zero, we see a resultant change in angular speed and quaternions not present in the purely translational motion of the first test. All axis are in terms of ft/s or rad/s as applicable.

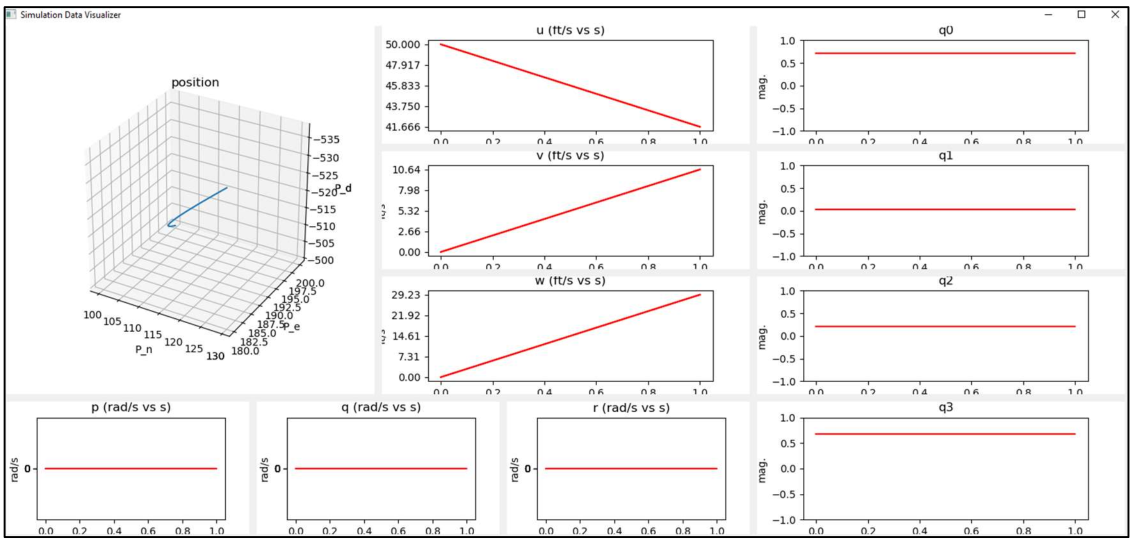

Figure 5. Outputs for Initial Condition with M = <0, 0, 0>

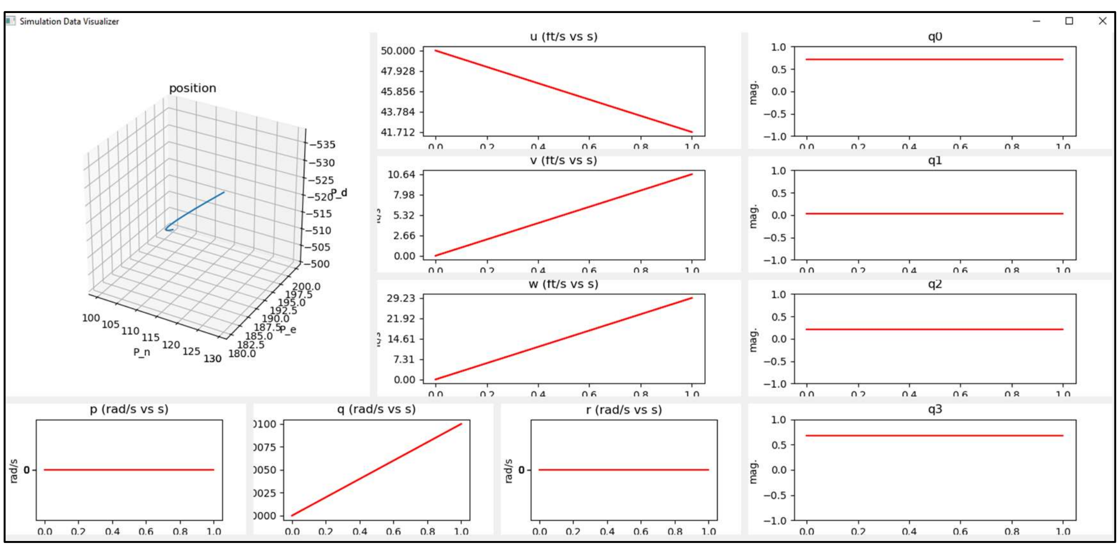

Figure 6. Outputs for Initial Condition with M = <0, 1e-4, 0>

6. Create a visualization scheme that shows the simulator states and rates, including a 3D visulation. You may use any interface or method you are happy with (displaying Euler angles is usually more intuitive that other parameterizations). Include a screen capture of its interface.

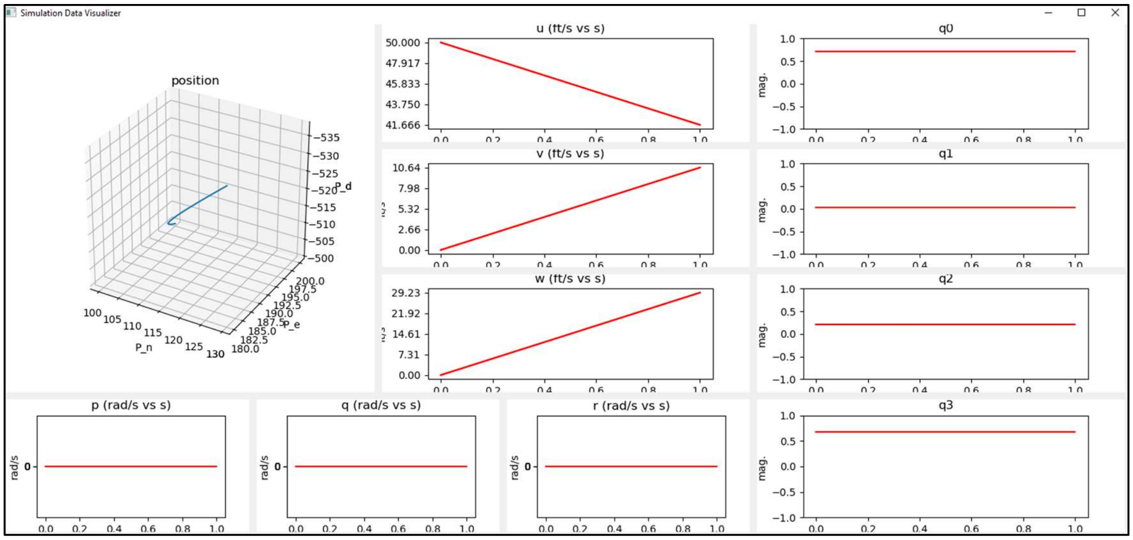

Figure 7. 3D and Parameter Visualization

Appendix A

Contents

```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

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

```python
    #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)*sin(psi) -
    cos(phi)*cos(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]
    return x_dot
```

```
def rot_kin(e0, e1, e2, e3, r, p, q):
    #d/dt( e )
    from numpy import matmul
    A3 = [[0, -p/2, -q/2, -r/2],
          [p/2, 0, r/2, -q/2],
          [q/2, -r/2, 0, p/2],
          [r/2, q/2, -p/2, 0]]
    b3 = [e0, e1, e2, e3]
    return matmul(A3, b3)

def rot_dyn(Gamma, p, q, r, L, M, N, Iy):
    #d/dt([p, q, r])
    #>>> Need to move to the body frame?
    x_dot = [Gamma[1]*p*q - Gamma[2]*q*r + Gamma[3]*L + Gamma[4]*N,
             Gamma[5]*p*r - Gamma[6]*(p**2-r**2) + 1/Iy*M,
             Gamma[7]*p*q - Gamma[1]*q*r + Gamma[4]*L + Gamma[8]*N]
    return x_dot

def derivatives(state, t, MAV):
    #state: [p_n, p_e, p_d, u, v, w, e0, e1, e2, e3, p, q, r]
    #FM:    [Fx, Fy, Fz, Ell, M, N]
    #MAV:   MAV.inert, MAV.m, MAV.gravity_needed

    from math import sin, cos

    #Unpack state, FM, MAV
    [p_n, p_e, p_d, u, v, w, e0, e1, e2, e3, p, q, r] = state
    storage = MAV.update_FM(t)
    [Fx, Fy, Fz, L, M, N] = MAV.FM
    [Ixz, Ix, Iy, Iz] = MAV.inert

    #Get angle measures
    angles = EP2Euler321([e0, e1, e2, e3])
    [psi, theta, phi] = angles

    #Get Xdot Terms
    d_dt = [[], [], [], []]
    d_dt[0] = pos_kin(psi, theta, phi, u, v, w)
    d_dt[1] = pos_dyn(p, q, r, u, v, w, Fx, Fy, Fz, MAV.mass)
    d_dt[2] = rot_kin(e0, e1, e2, e3, p, q, r)
    d_dt[3] = rot_dyn(make_gamma(MAV.inert), p, q, r, L, M, N, MAV.inert[2])

    #Build One Vector of Xdot
```

```python
    #Build the vector of xdot
    xdot = []
    for eqn_set in d_dt:
        for dot in eqn_set:
            xdot.append(dot)

    return xdot

def integrator(MAV, tf = 1, delta_t = 0.1, graphing = False):
    from numpy import linspace
    from scipy.integrate import odeint

    #Make the time values
    descrete_pts = (tf/delta_t) // 1  # force integer
    t = linspace(0, tf, descrete_pts + 1)

    #Integration Step
    outputs = odeint(derivatives, MAV.state0, t, args = (MAV,))

    #Optional 3D Path Graphing
    if graphing:
        from mpl_toolkits import mplot3d
        import matplotlib.pyplot as plt
        fig = plt.figure()
        ax = plt.axes(projection="3d")
        ax.plot3D(outputs[:,0], outputs[:,1], outputs[:,2],
        linestyle='-', marker='.')
        ax.set_xlabel('P_n')
        ax.set_ylabel('P_e')
        ax.set_zlabel('P_z')
        ax.invert_zaxis()
        plt.show()

    return [t, outputs]
```

```
#                   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
       self.name = aircraft
       self.mass = 10  # Mass (lbf)
       #Inert = [Ixz, Ix, Iy, Iz]
       self.inert = [20, 10, 10, 10]  # Moment of Inertia (lbf*ft^2)
       self.gravity_needed = False
       #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]
            #Gravity ONLY in base model
       self.FMeq = [0, 0, (lambda t: 32.2*self.mass), 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 update_mass(self, new_mass):
        #NOTE: Automatically updates gravity force in FM
        self.mass = new_mass

    def update_inert(self, new_inert):
        self.inert = new_inert

    def update_state0(self, new_state):
        if len(new_state) != 13:
```

```
        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 update_FM(self, t):
        from math import sin, cos
        from white_brandon_HW1 import EP2Euler321

        #Angularize Gravity
        angles = EP2Euler321(self.state0[6:10])
        Fg = f2b(angles, [0, 0, 32.2*self.mass])

        #All Other Forcing Functions
        for i in range(6):
            try:
                self.FM[i] = self.FMeq[i](t)
            except:
                self.FM[i] = self.FMeq[i]

        #Add in Gravity
        self.FM[0] += Fg[0]
        self.FM[1] += Fg[1]
        self.FM[2] += Fg[2]

        return self.FM

    #Add templated aircraft below this line to pregenerate aircraft
    def hw1_1(self):
        self.state0 = [100, 200, -500, 50, 0, 0,
                        0.70643, 0.03084, 0.21263, 0.67438, 0, 0, 0]
        self.FMeq = [0, 0, 0, 0, 0, 0]

    def hw1_2(self):
        from math import sin, cos
        self.state0 = [100, 200, -500, 50, 0, 0,
                        0.70643, 0.03084, 0.21263, 0.67438, 0, 0, 0]
        self.FMeq = [(lambda t: sin(t)), 0, 0,
                        0, 1e-4, 0]
```

```python
# NOTE:
#   Always import as:
#   from rotations import *

def make_Rbf(angles):
    from math import cos, sin, pi
    [psi, theta, phi] = angles

    psi = psi * (pi/180)
    theta = theta * (pi/180)
    phi = phi * (pi/180)

    R_bf = [[cos(psi)*cos(theta), sin(psi)*cos(theta), -sin(theta)],
            [cos(psi)*sin(theta)*sin(phi)-sin(psi)*cos(phi),
            cos(psi)*cos(phi)+sin(theta)*sin(psi)*sin(phi),
            cos(theta)*sin(phi)],
            [cos(psi)*sin(theta)*sin(phi)+sin(phi)*sin(psi),
            sin(theta)*sin(psi)*cos(phi)-sin(phi)*cos(psi),
            cos(theta)*cos(phi)]]

    return R_bf

def make_Rbw():
    from math import cos, sin, pi
    [alpha, beta] = angles

    # ***WARNING!***
    #If alpha or beta> 1 radian (60 deg), this breaks
    if alpha > 1 or beta > 1:
        alpha = alpha * (pi/180)
        beta = beta * (pi/180)

    R_bw = [[cos(alpha)*cos(beta), 0, 0],
            [sin(beta), 0, 0],
            [sin(alpha)*cos(beta), 0, 0]]

    return R_bw

def b2f(angles, vector):
    from numpy import matmul, transpose
    R_fb = transpose(make_Rbf(angles))
    return matmul(R_fb, vector)

def f2b(angles, vector):
    from numpy import matmul
```

```
    from numpy import matmul
    R_bf = make_Rbf(angles)
    return matmul(R_bf, vector)

def b2w(angles, vector):
    from numpy import matmul
    R_wb = transpose(make_Rbw(angles))
    return matmul(R_wb, vector)

def w2b(angles, vector):
    from numpy import matmul, transpose
    R_bw = make_Rbw(angles)
    return matmul(R_bw, vector)
```

```
#                   TITLE BLOCK
#***************************************************
#Author:     Brandon White
#Date:       08/28/2019
#Desc:       Creates a visual representation of
#            states vs time for sim data
#***************************************************
import sys

from PyQt5 import QtCore, QtGui, QtWidgets
from PyQt5.QtWidgets import QApplication, QMainWindow, QMenu, QVBoxLayout, QSizePolicy, QMessageBox, QWidget, QPushButton
from PyQt5.QtGui import QIcon

from matplotlib.backends.backend_qt5agg import FigureCanvasQTAgg as FigureCanvas
from matplotlib.figure import Figure
import matplotlib.pyplot as plt

import numpy, time

#Main GUI Class
class App(QMainWindow):

    def __init__(self):
        super().__init__()

        #Set form size (NOTE: graphs are [Pixels Width, Pixels Length] / 100)
        self.width = 1030 #40 for gaps
        self.height = 700 #50 for gapsS

        #GUI Position and Size
        self.left = 0
        self.top = 55
        self.title = 'Simulation Data Visualizer'
        self.graphs = []


    def initUI(self):
        self.setWindowTitle(self.title)
        self.setGeometry(self.left, self.top, self.width, self.height)

        #Create all graphs
```

```
        #create all graphs
        self.graphs.append(PlotCanvas(self, width=5, height=5,
            name_here = "position", given_data = self.data[:,0:3],
            t=self.t))
        self.graphs.append(PlotCanvas(self, width=5, height=1.6,
            name_here = "u", given_data = self.data[:, 3], t=self.t))
        self.graphs.append(PlotCanvas(self, width=5, height=1.6,
            name_here = "v", given_data = self.data[:, 4], t=self.t))
        self.graphs.append(PlotCanvas(self, width=5, height=1.6,
            name_here = "w", given_data = self.data[:, 5], t=self.t))
        self.graphs.append(PlotCanvas(self, width=3.3, height=1.8,
            name_here =  "p", given_data = self.data[:, 10],
            t=self.t))
        self.graphs.append(PlotCanvas(self, width=3.3, height=1.8,
            name_here = "q", given_data = self.data[:, 11],
            t=self.t))
        self.graphs.append(PlotCanvas(self, width=3.3, height=1.8,
            name_here = "r", given_data = self.data[:, 12],
            t=self.t))

        #Position Graphs
        self.graphs[0].move(10,0)
        self.graphs[1].move(520,0)
        self.graphs[2].move(520,170)
        self.graphs[3].move(520,340)
        self.graphs[4].move(10,510)
        self.graphs[5].move(350,510)
        self.graphs[6].move(690,510)

        for graph in self.graphs:
            graph.plot()

        self.show()

# Graphing Subclass
class PlotCanvas(FigureCanvas):

    def __init__(self, parent=None, width=5, height=4, dpi=100,
    name_here = "NO NAME GIVEN", given_data = [0], t=[0]):
        fig = Figure(figsize=(width, height), dpi=dpi)
        self.axes = fig.add_subplot(111)

        self.nombre = name_here
        self.t = t
        self.data = given_data
```

```python

        FigureCanvas.__init__(self, fig)
        self.setParent(parent)

        FigureCanvas.setSizePolicy(self,
                QSizePolicy.Expanding,
                QSizePolicy.Expanding)
        FigureCanvas.updateGeometry(self)
        self.plot()

    def plot(self):
        try:
            if self.nombre == "position":
                import matplotlib.pyplot as plt
                from mpl_toolkits.mplot3d import Axes3D
                ax = self.figure.add_subplot(111, projection = '3d')
                ax.cla()
                ax.plot3D(self.data[:, 0], self.data[:, 1], self.data[:, 2])
                ax.set_xlabel('P_n')
                ax.set_ylabel('P_e')
                ax.set_zlabel('P_d')
                ax.set_title(self.nombre)
                ax.invert_zaxis()
                self.draw()
            else:
                ax = self.figure.add_subplot(111)
                ax.plot(self.t,self.data, 'r')
                ax.set_title(self.nombre)
                self.draw()
        except:
            print('PLOT METHOD ERROR')
            return

def open_GUI(t = [0], sim_data = [0,0,0,0,0,0,0,0,0,0,0,0,0]):
    app = QApplication(sys.argv)
    ex = App()
    ex.data = sim_data
    ex.t = t
    ex.initUI()
    sys.exit(app.exec_())

if __name__ == '__main__':
    open_GUI()

```