AOM-HW3 with Code

Pierce Jackson

September 30, 2020

I display relevant portions of code throughout the document. At the very end of the document, I include the entire code in one piece - See Appendix A

1 Problem 1

Most of the explanation for problem 1 was written on a sheet of paper, which is attached to the submitted pdf document.

Here is the code for the COE2RV and RV2COE functions:

```
def COE2RV(a, e, i, RAAN, w, ta):
    # Used https://web.archive.org/web/20160418175843/https://ccar.colorado.edu/asen5070/handouts/cart2kep2002.pdf as a reference

    r = (a * (1 - e ** 2)) / (1 + e * np.cos(ta))  # Get position from orbit formula
    h = np.sqrt(mu * a * (1 - e ** 2))  # Magnitude of specific angular momentum
    X = r * (np.cos(RAAN) * np.cos(w + ta) - np.sin(RAAN) * np.sin(w + ta) * np.cos(i))  # Position X-component
    Y = r * (np.sin(RAAN) * np.cos(w + ta) + np.cos(RAAN) * np.sin(w + ta) * np.cos(i))  # Position Y-component
    Z = r * (np.sin(i) * np.sin(w + ta))  # Position Z-component
    p = a * (1 - e ** 2)  # Semilatus Rectum
    X_dot = ((X * h * e) / (r * p)) * np.sin(ta) - ((h / r) * (np.cos(RAAN) * np.sin(w + ta) + np.sin(RAAN) * np.cos(w + ta) * np.cos(i)))  # Velocity X-component
    Y_dot = ((Y * h * e) / (r * p)) * np.sin(ta) - ((h / r) * (np.sin(RAAN) * np.sin(w + ta) - np.cos(RAAN) * np.cos(w + ta) * np.cos(i)))  # Velocity Y-component
    Z_dot = ((Z * h * e) / (r * p)) * np.sin(ta) + ((h / r) * (np.sin(i) * np.cos(w + ta)))  # Velocity Z-component

    r_vector_from_COE2RV = np.array([X, Y, Z])  # Put X,Y,Z into an array to create a vector
    v_vector_from_COE2RV = np.array([X_dot, Y_dot, Z_dot])  # Put X_dot, Y_dot, Z_dot into an array to create a vector

    #print("Position Vector (km)= " + str(r_vector_from_COE2RV))
    #print("Velocity Vector (km/s) = " + str(v_vector_from_COE2RV))
    return(r_vector_from_COE2RV, v_vector_from_COE2RV)


def RV2COE(r_vector, v_vector):
    # Used "Orbital Mechanics for Engineering Students" 3rd Edition by Howard D. Curtis; Also used our class notes

```

```python
    r = np.linalg.norm(r_vector)  # Distance
    v = np.linalg.norm(v_vector)  # MAgnitude of Velocity or Speed
    v_r = np.dot(v_vector, r_vector) / r  # radial velocity
    h_vector = np.cross(r_vector, v_vector)  # specific angular momentum vector
    h = np.linalg.norm(h_vector)  # magnitude of specific angular momentum
    i = np.arccos(h_vector[2] / h)  # inclination
    n_vector = np.cross([0, 0, 1], h_vector)  # vector pointing to ascending node
    n = np.linalg.norm(n_vector)  # magnitude of n
    if n_vector[1] > 0:
        RAAN = np.arccos(n_vector[0] / n)  # Right Ascension of the Ascending node
    if n_vector[1] < 0:
        RAAN = 2 * np.pi - np.arccos(n_vector[0] / n)
    e_vector = (1 / mu) * ((((v ** 2) - (mu / r)) * (r_vector)) - ((r * v_r) * v_vector))  # eccentricity vector
    e = np.linalg.norm(e_vector)  # eccentricity
    if e_vector[2] > 0:
        w = np.arccos(np.dot(n_vector, e_vector) / (n * e))  # Argument of periapse
    if e_vector[2] < 0:
        w = 2 * np.pi - np.arccos(np.dot(n_vector, e_vector) / (n * e))
    if v_r > 0:
        ta = np.arccos((np.dot(e_vector, r_vector) / (e * r)))  # True anomaly
    if v_r < 0:
        ta = 2 * pi - np.arccos((r_vector / r) * (e_vector / e))
    Energy = (v ** 2 / 2) - (mu / r)
    if e == 1:
        p = h ** 2 / mu  # Semilatus Rectum
        return "Orbit is parabolic. Eccentricity = infinity"
    else:
        a = -mu / (2 * Energy)  # Semi-major Axis
        p = a * (1 - e ** 2)

    print("Semi-major Axis (km) = " + str(a))
    print("Eccentricity = " + str(e))
    print("Inclination (deg)= " + str(i * (180 / np.pi)))
    print("Right Ascension of the Ascending Node (deg) = " + str(RAAN * (180 / np.pi)))
    print("Argument of Periapse (deg)= " + str(w * (180 / np.pi)))
    print("True Anomaly (deg) = " + str(ta * (180 / np.pi)))

    return [a, e, i, RAAN, w, ta]


mu = 398600  # Gravitational Parameter of Earth km^3/s^2
r_vector = np.array([-6045.0, -3490.0, 2500.0])
v_vector = np.array([-3.457, 6.618, 2.533])


COE = RV2COE(r_vector, v_vector)
COE2RV(COE[0], COE[1], COE[2], COE[3], COE[4], COE[5])
```

The second to last line of code converts the position and velocity vectors to its classical orbital elements. The last line of code takes those elements and turns them back into the position and velocity vectors. I was returned the original position and velocity vectors back exactly.

2 Problem 2

To propagate the frozen orbit, from the previous assignment, I wrote my own Runge-Kutta 4th order (RK4) integrator that numerically integrates the unperturbed two-body equations of motion. We can break down the two-body equation of motion

$$\ddot{\mathbf{r}} = \frac{-\mu}{r^3}\mathbf{r} \tag{1}$$

into two differential equations with only one derivative in each:

$$\frac{d\mathbf{r_i}}{dt} = \mathbf{v_i} \tag{2}$$

$$\frac{d\mathbf{v_i}}{dt} = \frac{-\mu}{r^3}\mathbf{r_i} \tag{3}$$

where $\mathbf{r_i}$ is the position vector of body i (i = 1, 2), $\mathbf{v_i}$ is the velocity vector of body i, and r is the norm of the vectors $(\mathbf{r_2} - \mathbf{r_1})$. Because our satellite is orbiting Earth in a geocentric reference frame, we can set the position and velocity vectors of Earth (the first body) equal to 0.

I created a function named "force" that calculates eq. 2 and eq. 3 for a given $\mathbf{r}$ vector

```
    # Calculate State Vectors r & v using the COE2RV I created above
    r_sat, v_sat = COE2RV(a, e, i, raan, w, ta)  # Position state vector (km) and Velocity state vector (km/s) of satellite in the geocentric equitorial frame (km)

    t = 5 * T #Total time for propagation
    dt = 1 #time-step
    t_array = np.linspace(0, t, t / dt + 1) #make a time array
    X = np.array([r_sat[0], r_sat[1], r_sat[2], v_sat[0], v_sat[1], v_sat[2]]) #position and velocity vectors

def force(X, t): #This function gets velocity and acceleration vectors from position and velocity vector
        r = X[0:3]
        r_norm = np.linalg.norm(r)
        dr = np.zeros((3))
        dv = np.zeros((3))
        dr[:] = X[3:6]
        dv[:] = (-u0 / r_norm ** 3) * r
        X_dot = np.array([dr[0], dr[1], dr[2], dv[0], dv[1], dv[2]])
        return X_dot

```

Next, I use the well known RK4 algorithm of finding k1, k2, k3, and k4 to calculate one step forward in $\mathbf{r}$ by calling the "force" function I created

```

def RK4_algorithm(X, t, dt): #This function calculates 1 time-step forward using Runge-Kutta 4 Integration Scheme

        k1 = force(X, t)
        k2 = force(X + dt * k1 / 2, t + dt / 2)
```

```
        k3 = force(X + dt * k2 / 2, t + dt / 2)
        k4 = force(X + dt * k3, t + dt)

        r_update = X + (dt / 6) * (k1 + 2 * k2 + 2 * k3 + k4)
        t_update = t + dt

        r_mag = np.linalg.norm(r_update)
        return(r_update, t_update, r_mag)
```

Then, I created a function named "RK4" that loops the RK4_algorithm function to calculate all steps from t = 0 to the desired t_final. The integrator uses a fixed-time-step size that is defined as t_final divided by dt which is set by the user.

```

def RK4(X, t_final, dt): #This function loops the RK4_algorithm function to integrate
    over the whole time of the simulation
    steps = t_final / dt
    r_array = np.zeros((int(steps), 5))
    for i in range((int(steps))):
        Mean_anomaly = ((2 * np.pi) * (t_array[i])) / (T)
        Ea = Eccentric_Anomaly(Mean_anomaly)
        theta = 2 * np.arctan(np.sqrt((1 + e) / (1 - e)) * np.tan(Ea / 2))
        r_star, t_update, r_mag = RK4_algorithm(X, i, dt)
        X = r_star
        r_array[i, 0] = X[0]
        r_array[i, 1] = X[1]
        r_array[i, 2] = X[2]
        r_array[i, 3] = theta
        r_array[i, 4] = r_mag
    r_update = r_star
    return r_array
```

To compare how my RK4 integrator compares to the conic trajectory, I created another function that calculates the same orbit using classical orbital elements and the orbit formula

$$r = \frac{h^2}{\mu} \frac{1}{1 + ecos(\theta)} \tag{4}$$

where h is specific angular momentum, μ is the gravitational parameter of Earth, e is eccentricity, and θ is true anomaly.

```

def conic_trajectory(T, t_final, dt): #Calculate conic/theoretical orbit using classical
    orbital elements and the orbit formula
    steps = t_final / dt
    r_conic = np.zeros((int(steps), 5))
    for j in range(int(steps)):
        Mean_anomaly = ((2 * np.pi) * (t_array[j])) / (T)
        Ea = Eccentric_Anomaly(Mean_anomaly)
        theta = 2 * np.arctan(np.sqrt((1 + e) / (1 - e)) * np.tan(Ea / 2))
        r_conic_mag = (h ** 2 / u0) * (1 / (1 + e * np.cos(theta)))
        X = r_conic_mag * (np.cos(raan) * np.cos(w + theta) - np.sin(raan) * np.sin(w
  + theta) * np.cos(i))  # Position X-component
        Y = r_conic_mag * (np.sin(raan) * np.cos(w + theta) + np.cos(raan) * np.sin(w
  + theta) * np.cos(i))  # Position Y-component
```

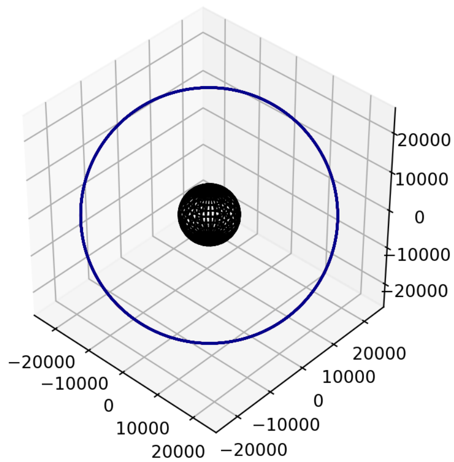
```
            Z = r_conic_mag * (np.sin(i) * np.sin(w + theta))  # Position Z-component
            r_conic[j, 0:5] = X, Y, Z, theta, r_conic_mag
        return r_conic
```

To ensure that both the RK4 integrator and the conic-trajectory simulator use the same fixed-steps (because RK4 uses time-steps while the conic trajectory simulator uses true-anomaly steps), I use mean and eccentric anomalies to "normalize" the time and true anomalies being used. This ensures that each time-step the RK4 integrator uses corresponds to the true anomaly step that the conic trajectory is using. This allows me to more easily compare the accuracy of the two simulations as I can directly compare the calculations made at every step by RK4 to the calculations made by the same step of the conic-trajectory.

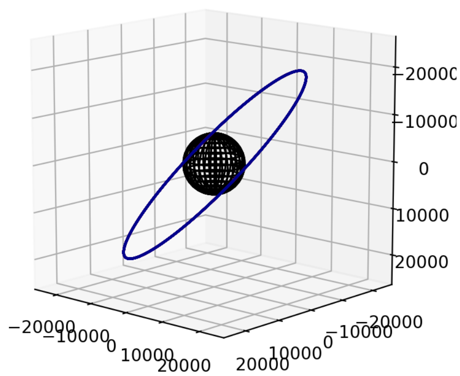
```

def Eccentric_Anomaly(Me): #From HW1
        ratio = 1
        if Me < np.pi:
            E_i = Me + (e / 2.0)
        if Me >= np.pi:
            E_i = Me - (e / 2.0)
    # Step 2) Calculate f(E_i) and f'(E_i) and get ratio f/f'. If |ratio| > tolerance, calculate new E_i+1 = E_i - ratio_i and loop until |ratio| meets tolerance
        while np.abs(ratio) > 1e-12:
            f1 = E_i - e * np.sin(E_i) - Me
            f2 = 1 - e * np.cos(E_i)
            ratio = f1 / f2
            E_i = E_i - ratio
        Ecc_anomaly = E_i
        return Ecc_anomaly
```

Finally, I can run the RK4 integrator and conic-trajectory simulation and compare their results. For the simulation results shown, the initial position and velocity vectors used classical orbital elements $a = 26610.2km$ (which produces an orbital period, T, equal to 12 hrs), $i = 45^o$, $\Omega = 45^o$, $w = 270^o$, $\theta = 0^o$, and $e = \frac{-J_3 R_E}{2J_2 a} sin(i)$ (the eccentricity value was taken from the "Frozen orbit solution from HW2 discussion on Canvas") where J3 is the perturbation due to Earth's 3rd zonal harmonic and is equal to $-2.53243x10^{-6}$, J2 is the perturbation due to Earth's 2nd zonal harmonic and is equal to $1.08263x10^{-3}$, and RE is the average radius of the Earth = $6378km$. Total propagation time is 5 times the orbital period $t = 5 * T$ and time-step $dt = 1$ (and θ corresponding to each time step) [See first chunk of code above the definition of the force function].

```
    orbit = RK4(X, t, dt) #Simulate orbit using RK4 integration
    conic_orbit = conic_trajectory(T, t, dt) #Simulate orbit using orbit formula
    print(np.max(np.abs(orbit[:, 4] - conic_orbit[:, 4]))) #Find the largest deviation in position between RK4 simulation and Conic/Orbit formula simulation
```



(a) View from above the orbit; No perturbations included. Both orbits calculated by RK4 (blue) and Conic-Trajectory (red) are plotted


(b) View from the side the orbit; No perturbations included. Both orbits calculated by RK4 (blue) and Conic-Trajectory (red) are plotted

On the above plots, both the RK4 and Conic-Trajectory orbits are plotted, however they are nearly identical, so they overlap and only the RK4 orbit is seen. You can check this by running the code yourself. The output of the "print(np.max(np.abs(orbit[:, 4] - conic_orbit[:, 4])))" line takes the directly compares the orbital position calculated by RK4 and Conic-Trajectory by taking the difference. Instead of printing the entire difference array, I only print the maximum difference calculated which will give us an idea as to how accurate the RK4 integrator is to the theoretical conic-trajectory. The code returns the value "0.0010166317733819596" which means the greatest deviation in position between the two simulated orbits is just 1.01 meters.

So, to answer the question: "Does the error ever grow over 500km?" The answer is most likely "no" even if one were to propagate the orbit (without perturbations) for a much longer time.

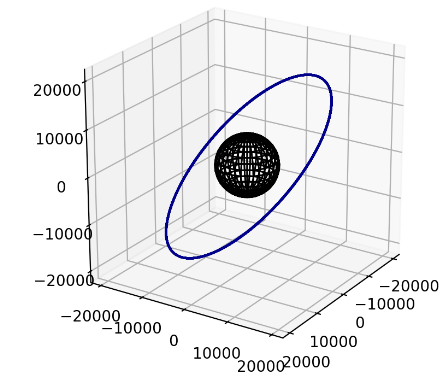
3 Problem 3

Problem 3 is the same as Problem 2, though we now introduce J2 and J3 perturbations to the orbit calculations.

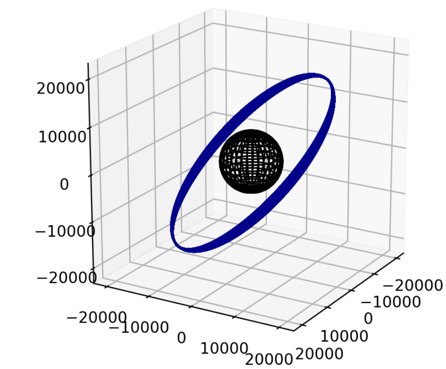
I basically recycle the code from problem 2 making small edits so the functions and variables do not get mixed up with the non-perturbed calculations. The J2 and J3 perturbing accelerations are calculated in the "force_perturbed" function that plays the same roll that "force" function does in problem 2. I calculate the X, Y, and Z components of the J2 and J3 perturbing accelerations, then take their magnitudes and add them to the acceleration vector felt by the satellite on the orbit:

```python
def force_perturbed(X, t):  # Calculates acceleration vector, including J2 and J3
        r = X[0:3]
        r_norm = np.linalg.norm(r)

        # J2 Perturbing Acceleration
        aj2_coefficient = (-3.0 / 2.0) * J2 * (u0 / r_norm ** 2) * (Re / r_norm) ** 2
        aj2x = aj2_coefficient * (1 - 5 * ((r[2] / r_norm) ** 2)) * (r[0] / r_norm)
        aj2y = aj2_coefficient * (1 - 5 * ((r[2] / r_norm) ** 2)) * (r[1] / r_norm)
        aj2z = aj2_coefficient * (3 - 5 * ((r[2] / r_norm) ** 2)) * (r[2] / r_norm)
        aj2 = np.array([aj2x, aj2y, aj2z])

        # J3 Perturbing Acceleration (From Schaub H, Junkins, JL. 2009. "Analytical Mechanics of Space Systems" pg. 380)
        aj3_coefficient = (-1.0 / 2.0) * J3 * (u0 / r_norm ** 2) * (Re / r_norm) ** 3
        aj3x = aj3_coefficient * (5 * (7 * (r[2] / r_norm) ** 3 - 3 * (r[2] / r_norm)) * (r[0] / r_norm))
        aj3y = aj3_coefficient * (5 * (7 * (r[2] / r_norm) ** 3 - 3 * (r[2] / r_norm)) * (r[1] / r_norm))
        aj3z = aj3_coefficient * (3 * (10 * (r[2] / r_norm) ** 2 - (35.0 / 3.0) * (r[2] / r_norm) ** 4 - 1))
        aj3 = np.array([aj3x, aj3y, aj3z])

        dr = np.zeros((3))
        dv = np.zeros((3))
        dr[:] = X[3:6]
        dv[:] = (-u0 / r_norm ** 3) * r + aj2 + aj3  # ACCOUNT FOR J2 AND J3 PERTURBING ACCELERATIONS HERE

        X_dot = np.array([dr[0], dr[1], dr[2], dv[0], dv[1], dv[2]])
        return X_dot
```

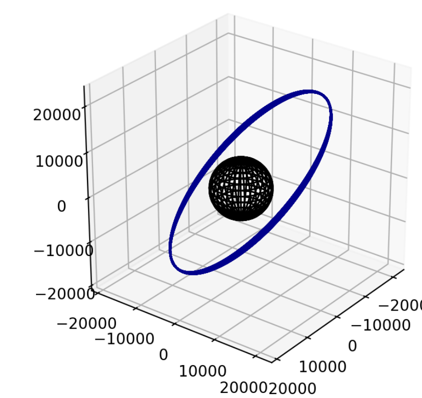
The same sequence for numerically integrating the orbit with RK4 is followed as in problem 2. Some results of the perturbed RK4 integration are shown below.



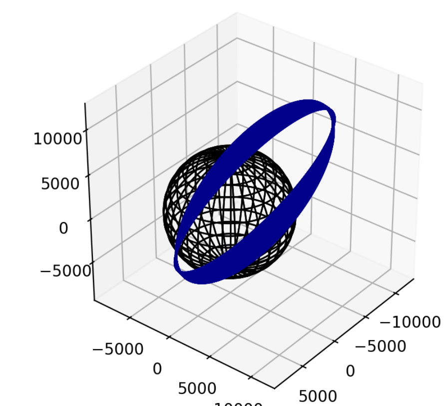
(a) Simulation of 10 orbits with J2 and J3 perturbations present, using same COE as problem 2



(b) Simulation of 200 orbits with J2 and J3 perturbations present, using same COE as problem 2



(c) Simulation of 100 orbits with $e = 0.12$ and the other COE unchanged. J2 and J3 perturbations included.



(d) Simulation of 100 orbits with $e = 0.3$ and $a = 10610.2km$ keeping the remaining COE unchanged. J2 and J3 perturbations included.

Figures (a) and (b) show the short-term and long-term variation, respectively, in the "frozen orbit" due to J2 and J3 perturbations. The short-term variation in the orbit is minimal and is not distinguishable from the non-perturbed orbit simulation from problem 2 [1]. However, after 200 orbits (though short term variations are negligible) variations in the classical orbital elements begin to manifest as the orbit slightly widens around the equator. Because this orbit has a period of 12 hours, this long-term variation is achieved over ≈ 100 days.

Figures (c) and (d) are provided for comparison reasons. Orbit (c) is prop-

[1]It should be noted that "short term" is typically defined as on the order of a single orbit. 10 orbits could be considered beyond this range and no longer "short-term," however, if no meaningful variations are seen after 10 orbits, then a time span of less than 10 orbits will also provide no significant variations to the COE

agated for 100 orbits with an eccentricity value slightly larger than (a) and (b). Some variation in the orbit (around the equator) is present, though not dramatic. Orbit (d) has a much smaller semi-major axis at 10610.2 km and an eccentricity value of 0.3. Because this orbit is much closer to the Earth than the others, the effect of J2 and J3 perturbations are much stronger, resulting in major variations in the classical orbital elements after only 100 orbits.

The results of problem 2 and problem 3 follow the basic idea of general perturbation theory. Problem 2 models an unperturbed orbit which can be numerically integrated, but can also be accurately modeled by a algebraic formula - eq. 4. This equation can be used to plot an ellipse (or circle, parabola, and hyperbola in special cases) around a central body. However, it cannot model an orbit subject to perturbing forces. This is where numerical integration comes in handy. By using a fixed-step integration method such as Runge-Kutta 4, Euler Forward, or Newton's method we can include the perturbing forces felt by an orbiting body at each time step and accurately model the orbit with perturbations. One of the downsides of numerical integration is its accuracy and stability dependent on the chosen step-size and can also be computationally intensive and time consuming to simulate.

Some comments: While completing this assignment and thinking about the short-term and long-term variations, I realized that it is a situation of small-things add up over time and can become impactful on the long-term. A common example is if one were to put a few coins or dollars into their piggy bank each day, they wouldn't have much money saved after a few days - but after many years, one could accumulate a (relatively) large chunk of money. So, just because short-term variations are negligible, like for our "frozen" orbit, that does not mean that it will remain "frozen" on the long-term.

Also, on a side note: The RK4 method is much slower than scipy's odeint function. It gets the job done, though.

4 Problem 4

In Problem 4, we add atmospheric drag as a perturbation (and remove J2 J3) into the orbit simulation and must find the time it takes a 200km altitude equatorial circular orbit to decay to 50 km [2].

To add atmospheric drag into our orbit simulation, we must calculate the perturbing force caused by the atmospheric drag on our spacecraft. The magnitude of this perturbing acceleration is dependent on factors such as atmospheric density, the mass, ballistic coefficient, coefficient of drag, and area normal to velocity of the spacecraft.

First, we define the atmospheric density as:

$$\rho = \rho_0 \exp(-\frac{h - h_0}{H}) \tag{5}$$

[2]For Problem 4, I used Example_12_01.m (located on page e137 here: `https://booksite.elsevier.com/9780080977478/downloads/Appendix_D.pdf`) for some guidance

where, from the assignment: "h is the instantaneous spacecraft altitude, ρ_0 is a reference density (use $5.464x10^{-10} km/m^3$) at a reference altitude h_0 (use 180 km) and H is a scale height (use 29.740 km)."

Next, we define the spacecraft properties: Mass (M) = 250 kg, Area (A) = $4m^2$, and Coefficient of Drag (Cd) = 2.2 which is the standard for low Earth satellites. With these parameters, we can calculate the satellite's Ballistic Coefficient (Bc):

$$B_c = \frac{M}{C_d A} = 28.4091 kg/m^2 \tag{6}$$

The perturbing acceleration due to atmospheric drag is

$$a_{AD} = -\frac{1}{2}\rho V^2 \frac{C_d A}{M} \tag{7}$$

$$= -\frac{1}{2}\frac{\rho \mathbf{V^2}}{B_c} \tag{8}$$

is negative because atmospheric drag is opposes the satellite's velocity vector. V = the relative velocity of the satellite with respect to the Earth's angular velocity vector crossed with the satellite's position vector:

$$\mathbf{V} = \mathbf{V_{satellite}} - \omega_{\mathbf{Earth}} \times \mathbf{r_{satellite}} \tag{9}$$

The numerical integration process is nearly identical to the previous methods, with these parameters calculated in the "force_drag" function. Once the perturbing acceleration is calculated, it is added to eq. 1 which is then propagated through the RK4 algorithm as done previously:

```
def force_drag(X, t):  # Calculates acceleration due to atmospheric drag

    wE = np.array([0, 0, 7.2921159e-5])  # Angular Velocity vector of Earth
    r = X[0:3]
    r_norm = np.linalg.norm(r)
    v = X[3:6]
    vrel = v - np.cross(wE, r)  # Relative velocity vector of satellite with respect to rotating Earth
    vrel_norm = np.linalg.norm(vrel)
    uv = vrel / vrel_norm  # Relative velocity unit vectors

    # Drag Perturbation
    alt = r_norm - Re  # instantaneous altitude
    # print(alt)
    h0 = 180  # km reference altitude
    H = 29.740  # km scale height
    p = p0 * np.exp(- (alt - h0) / H)

    # Satellite parameters
    M = 250  # kg Mass of starlink satellite
    A = 4.0  # m^2
    Cd = 2.2  # Drag Coefficient ***This is the standard value for low earth satellites ***
    Bc = M / (Cd * A)  # Ballistic Coefficient
```

```
        
    Fd = -Cd * A / M * p * (1000 * vrel) ** 2 / 2 * uv  # Acceleration (yes, acceleration not force --> a = f/m. Don't mind the variable naming convention) due to the atmospheric drag force. 1000 * vrel to get m/s^2
    ad = Fd / 1000  # go back to km/s^2

    dr = np.zeros((3))
    dv = np.zeros((3))
    dr[:] = X[3:6]
    dv[:] = ((-u0 / r_norm ** 3) * r) + ad  # ACCOUNT FOR ATMOSPHERIC DRAG ACCELERATION HERE#

    X_dot = np.array([dr[0], dr[1], dr[2], dv[0], dv[1], dv[2]])
    return X_dot
```

With the parameters defined above, and dt = 1 the code calculates the time to decay down to 50 km ≈ **89973 seconds**.

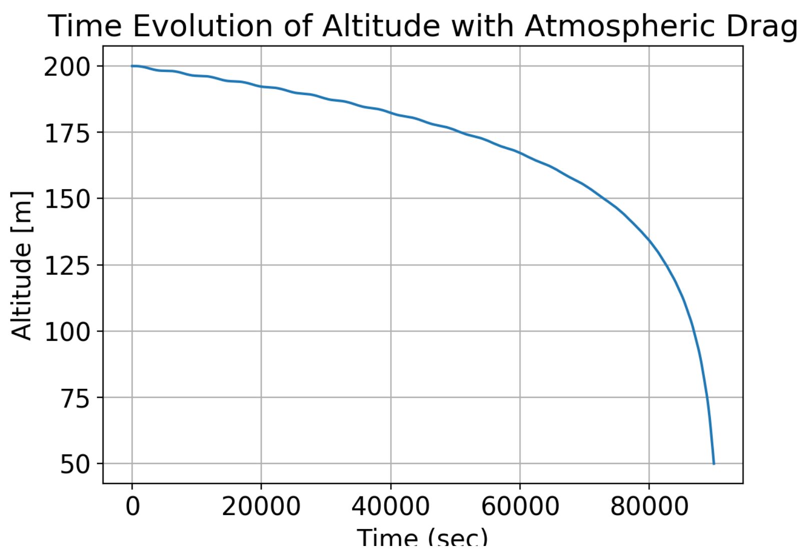


Figure 3: Plot displaying the decrease in altitude from a 200km equatorial circular orbit down to 50km due to atmospheric drag

Finally, we investigate how a 10% uncertainty in out reference density value ρ_0 impacts the decay time with a Monte Carlo Analysis.

```
def Monte_Carlo_Decay(p0_original):
    uncertainty = 0.1 #10 %
    num_of_runs = 40
    decay_times = np.zeros((num_of_runs, 3))
    for n in range(num_of_runs):
        p0 = np.random.normal(p0_original, p0_original * uncertainty)
        decay_time = Orbit_Decay(p0)
        decay_times[n, 0] = p0
        decay_times[n, 1] = decay_time[-1]
```

```
        decay_times[n, 2] = n + 1
        #print("p0 = " + str(decay_times[n, 0]))
        #print(decay_times[n, 0], decay_times[n, 1], decay_times[n, 2])

    #Calculate Uncertainty in Decay Time#
    avg_decay_time = np.average(decay_times[:, 1]) #Take average of the decay time
    deviations = np.zeros((num_of_runs))
    for j in range(num_of_runs): #Get the deviation of each decay time with respect to
     the average value
        deviations[j] = np.abs(decay_times[j, 1] - avg_decay_time)
    uncertainty = np.average(deviations[:]) #Take average of deviations
    print("The uncertainty in time to decay with 10'%' uncertainty in p0 is: " + str(
     uncertainty))

    #Plot Monte Carlo Results#
    ax1 = plt.subplot(2, 1, 1)
    ax1.plot(decay_times[:, 0], decay_times[:, 1])
    plt.xlabel("Time to Decay (sec)")
    plt.ylabel("p0")
    plt.title("Monte Carlo Simulation for p0 and Impact on Decay Time")

    ax2 = plt.subplot(2, 1, 2)
    ax2.plot(decay_times[:, 2], decay_times[:, 1])
    plt.ylabel("Time to Decay (sec)")
    plt.xlabel("Simulations Ran")
    plt.title("Monte Carlo Decay Time vs. Simulation Runs")

    plt.grid()
    plt.show()

    return decay_times, uncertainty
```

Running the Monte Carlo simulation 40 times with a 10% uncertainty in ρ_0 and dt = 0.5 results in an uncertainty in decay time (down to 50km) of $\approx$ **4736 seconds**.

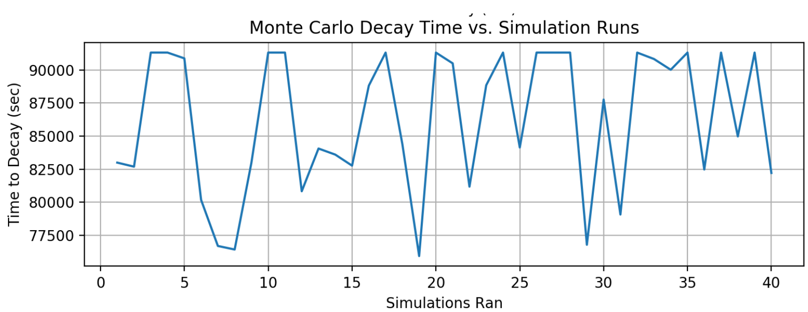


Figure 4: Variation in Decay time for each Monte Carlo simulation ran

```
Time to decay to 50 km with p0 = (5.110195843565214e-10) is: [91323.0] seconds
Time to decay to 50 km with p0 = (5.829992807952333e-10) is: [84384.0] seconds
Time to decay to 50 km with p0 = (6.494050050939299e-10) is: [75916.0] seconds
Time to decay to 50 km with p0 = (5.065688617189978e-10) is: [91323.0] seconds
Time to decay to 50 km with p0 = (5.432120318002591e-10) is: [90507.5] seconds
Time to decay to 50 km with p0 = (6.066964977891941e-10) is: [81172.0] seconds
Time to decay to 50 km with p0 = (5.534738643705791e-10) is: [88854.0] seconds
Time to decay to 50 km with p0 = (5.204534010987316e-10) is: [91323.0] seconds
Time to decay to 50 km with p0 = (5.847943322780693e-10) is: [84140.0] seconds
Time to decay to 50 km with p0 = (4.756628649092569e-10) is: [91323.0] seconds
Time to decay to 50 km with p0 = (4.889232402054714e-10) is: [91323.0] seconds
Time to decay to 50 km with p0 = (4.1845879304052393e-10) is: [91323.0] seconds
Time to decay to 50 km with p0 = (6.418914417789473e-10) is: [76770.5] seconds
Time to decay to 50 km with p0 = (5.604553051063479e-10) is: [87765.5] seconds
Time to decay to 50 km with p0 = (6.228551597161905e-10) is: [79062.5] seconds
Time to decay to 50 km with p0 = (5.144122116056865e-10) is: [91323.0] seconds
Time to decay to 50 km with p0 = (5.410612176373871e-10) is: [90837.5] seconds
Time to decay to 50 km with p0 = (5.46122573160638e-10) is: [90022.5] seconds
Time to decay to 50 km with p0 = (4.459585852292013e-10) is: [91323.0] seconds
Time to decay to 50 km with p0 = (5.969952779739625e-10) is: [82470.5] seconds
Time to decay to 50 km with p0 = (5.044058518606537e-10) is: [91323.0] seconds
Time to decay to 50 km with p0 = (5.791344018339641e-10) is: [84975.5] seconds
Time to decay to 50 km with p0 = (4.986120038086048e-10) is: [91323.0] seconds
Time to decay to 50 km with p0 = (5.988889320218816e-10) is: [82207.0] seconds
The uncertainty in time to decay with 10'%' uncertainty in p0 is: 4735.8324999999995
[Finished in 3211.3s]
```

Figure 5: Output of Monte Carlo simulation ran 40 times with an uncertainty in the reference atmospheric density ρ_0 of 0.1 and time step = 0.5. Note the run time...

A Full Code

```
# Pierce Jackson
# Applied Orbital Mechanics, Davide Guzzetti, September 30, 2020
# HW 3
import numpy as np
import scipy as sci
import scipy.stats as stats
import scipy.integrate  # ode solver solve_ivp(function, t_span, y0) : tspan is interval
    of integration
from matplotlib import pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
## Problem 1 - Create algorithms to convert from Classical Orbital Elements -->
    Position and Velocity Vectors (COE2RV) and vice versa (RV2COE) ##


def COE2RV(a, e, i, RAAN, w, ta):
    # Used https://web.archive.org/web/20160418175843/https://ccar.colorado.edu/
    asen5070/handouts/cart2kep2002.pdf as a reference

    r = (a * (1 - e ** 2)) / (1 + e * np.cos(ta))  # Get position from orbit formula
    h = np.sqrt(mu * a * (1 - e ** 2))  # Magnitude of specific angular momentum
    X = r * (np.cos(RAAN) * np.cos(w + ta) - np.sin(RAAN) * np.sin(w + ta) * np.cos(i)
    )  # Position X-component
    Y = r * (np.sin(RAAN) * np.cos(w + ta) + np.cos(RAAN) * np.sin(w + ta) * np.cos(i
    ))  # Position Y-component
```

```
    Z = r * (np.sin(i) * np.sin(w + ta))  # Position Z-component
    p = a * (1 - e ** 2)  # Semilatus Rectum
    X_dot = ((X * h * e) / (r * p)) * np.sin(ta) - ((h / r) * (np.cos(RAAN) * np.sin(w + ta) + np.sin(RAAN) * np.cos(w + ta) * np.cos(i)))  # Velocity X-component
    Y_dot = ((Y * h * e) / (r * p)) * np.sin(ta) - ((h / r) * (np.sin(RAAN) * np.sin(w + ta) - np.cos(RAAN) * np.cos(w + ta) * np.cos(i)))  # Velocity Y-component
    Z_dot = ((Z * h * e) / (r * p)) * np.sin(ta) + ((h / r) * (np.sin(i) * np.cos(w + ta)))  # Velocity Z-component

    r_vector_from_COE2RV = np.array([X, Y, Z])  # Put X,Y,Z into an array to create a vector
    v_vector_from_COE2RV = np.array([X_dot, Y_dot, Z_dot])  # Put X_dot, Y_dot, Z_dot into an array to create a vector

    #print("Position Vector (km)= " + str(r_vector_from_COE2RV))
    #print("Velocity Vector (km/s) = " + str(v_vector_from_COE2RV))
    return(r_vector_from_COE2RV, v_vector_from_COE2RV)


def RV2COE(r_vector, v_vector):
    # Used "Orbital Mechanics for Engineering Students" 3rd Edition by Howard D. Curtis; Also used our class notes

    r = np.linalg.norm(r_vector)  # Distance
    v = np.linalg.norm(v_vector)  # MAgnitude of Velocity or Speed
    v_r = np.dot(v_vector, r_vector) / r  # radial velocity
    h_vector = np.cross(r_vector, v_vector)  # specific angular momentum vector
    h = np.linalg.norm(h_vector)  # magnitude of specific angular momentum
    i = np.arccos(h_vector[2] / h)  # inclination
    n_vector = np.cross([0, 0, 1], h_vector)  # vector pointing to ascending node
    n = np.linalg.norm(n_vector)  # magnitude of n
    if n_vector[1] > 0:
        RAAN = np.arccos(n_vector[0] / n)  # Right Ascension of the Ascending node
    if n_vector[1] < 0:
        RAAN = 2 * np.pi - np.arccos(n_vector[0] / n)
    e_vector = (1 / mu) * ((((v ** 2) - (mu / r)) * (r_vector)) - ((r * v_r) * v_vector))  # eccentricity vector
    e = np.linalg.norm(e_vector)  # eccentricity
    if e_vector[2] > 0:
        w = np.arccos(np.dot(n_vector, e_vector) / (n * e))  # Argument of periapse
    if e_vector[2] < 0:
        w = 2 * np.pi - np.arccos(np.dot(n_vector, e_vector) / (n * e))
    if v_r > 0:
        ta = np.arccos((np.dot(e_vector, r_vector) / (e * r)))  # True anomaly
    if v_r < 0:
        ta = 2 * pi - np.arccos((r_vector / r) * (e_vector / e))
    Energy = (v ** 2 / 2) - (mu / r)
    if e == 1:
        p = h ** 2 / mu  # Semilatus Rectum
        return "Orbit is parabolic. Eccentricity = infinity"
    else:
        a = -mu / (2 * Energy)  # Semi-major Axis
        p = a * (1 - e ** 2)

    print("Semi-major Axis (km) = " + str(a))
    print("Eccentricity = " + str(e))
    print("Inclination (deg)= " + str(i * (180 / np.pi)))
```

```
    print("Right Ascension of the Ascending Node (deg) = " + str(RAAN * (180 / np.pi)))
    print("Argument of Periapse (deg)= " + str(w * (180 / np.pi)))
    print("True Anomaly (deg) = " + str(ta * (180 / np.pi)))

    return [a, e, i, RAAN, w, ta]


mu = 398600  # Gravitational Parameter of Earth km^3/s^2
r_vector = np.array([-6045.0, -3490.0, 2500.0])
v_vector = np.array([-3.457, 6.618, 2.533])


#COE = RV2COE(r_vector, v_vector)
#COE2RV(COE[0], COE[1], COE[2], COE[3], COE[4], COE[5])

#--------------------------------------------------#
#--------------------------------------------------#

## Problem 2 - Numerically propagate orbit from HW2 ##

# Gravitational Parameters
G = 6.67408e-20  # Gravitational Constant in km^3/(kg*s^2)
Me = 5.9722e+24  # Mass of Earth in kg
Ms = 11110  # Mass of Satellite --> I have chosen the mass of the Hubble Space Telescope
J2 = 1.08263e-3
J3 = -2.53243e-6
theta_E = np.deg2rad(15.04 / 3600)  # rotation rate of earth in rads/s
Re = 6378  # radius of Earth (km)
u0 = 398600  # Standard Gravitational Parameter in km^3/s^2


a = 22610.2  # Semi-major Axis
i = np.deg2rad(45)  # Inclination in radians (value in function in degrees)
raan = np.deg2rad(45)  # Right-Ascension of the Ascending Node in radians (value in function in degrees)
w = np.deg2rad(270)  # Argument of Perigee in radians (value in function in degrees)
#ta = np.deg2rad(0)
ta = np.deg2rad(0.00)  # True Anomaly in radians (value in function in degrees)
e = ((-J3 * Re) / (2 * J2 * a)) * np.sin(i)  # Eccentricity of orbit in degrees TAKEN FROM DISCUSSION
# e = 0.3
apogee = a * (1 + e)  # Apogee radius
perigee = a * (1 - e)  # Perigee radius
v_perigee = np.sqrt(2 * ((u0 / perigee) - (u0 / (2 * a))))  # Velcoity of spacecraft at perigee in km/s; from conservation of energy equation
h = perigee * v_perigee  # specific angular momentum of spacecraft in km^2/s
T = ((2 * np.pi) / np.sqrt(u0)) * a ** (3.0 / 2.0)  # period in seconds
T_hrs = T / 3600.0

#--------------------------------------------------

def Orbit_Integrator_RK4_no_Perturbations(a, e, i, raan, w, ta):

    # Calculate State Vectors r & v using the COE2RV I created above
    r_sat, v_sat = COE2RV(a, e, i, raan, w, ta)  # Position state vector (km) and Velocity state vector (km/s) of satellite in the geocentric equitorial frame (km)
```

```

t = 5 * T  # Total time for propagation
dt = 1  # time-step
t_array = np.linspace(0, t, t / dt + 1)  # make a time array
X = np.array([r_sat[0], r_sat[1], r_sat[2], v_sat[0], v_sat[1], v_sat[2]])  # position and velocity vectors

def force(X, t):  # This function gets velocity and acceleration vectors from position and velocity vector
    r = X[0:3]
    r_norm = np.linalg.norm(r)
    dr = np.zeros((3))
    dv = np.zeros((3))
    dr[:] = X[3:6]
    dv[:] = (-u0 / r_norm ** 3) * r
    X_dot = np.array([dr[0], dr[1], dr[2], dv[0], dv[1], dv[2]])
    return X_dot

def RK4_algorithm(X, t, dt):  # This function calculates 1 time-step forward using Runge-Kutta 4 Integration Scheme

    k1 = force(X, t)
    k2 = force(X + dt * k1 / 2, t + dt / 2)
    k3 = force(X + dt * k2 / 2, t + dt / 2)
    k4 = force(X + dt * k3, t + dt)

    r_update = X + (dt / 6) * (k1 + 2 * k2 + 2 * k3 + k4)
    t_update = t + dt

    r_mag = np.linalg.norm(r_update)
    return(r_update, t_update, r_mag)

def RK4(X, t_final, dt):  # This function loops the RK4_algorithm function to integrate over the whole time of the simulation
    steps = t_final / dt
    r_array = np.zeros((int(steps), 5))
    for i in range((int(steps))):
        Mean_anomaly = ((2 * np.pi) * (t_array[i])) / (T)
        Ea = Eccentric_Anomaly(Mean_anomaly)
        theta = 2 * np.arctan(np.sqrt((1 + e) / (1 - e)) * np.tan(Ea / 2))
        r_star, t_update, r_mag = RK4_algorithm(X, i, dt)
        X = r_star
        r_array[i, 0] = X[0]
        r_array[i, 1] = X[1]
        r_array[i, 2] = X[2]
        r_array[i, 3] = theta
        r_array[i, 4] = r_mag
    r_update = r_star
    return r_array

def Eccentric_Anomaly(Me):  # From HW1
    ratio = 1
    if Me < np.pi:
        E_i = Me + (e / 2.0)
    if Me >= np.pi:
        E_i = Me - (e / 2.0)
# Step 2) Calculate f(E_i) and f'(E_i) and get ratio f/f'. If |ratio| > tolerance,
```

```python
        calculate new E_i+1 = E_i - ratio_i and loop until |ratio| meets tolerance
        while np.abs(ratio) > 1e-12:
            f1 = E_i - e * np.sin(E_i) - Me
            f2 = 1 - e * np.cos(E_i)
            ratio = f1 / f2
            E_i = E_i - ratio
        Ecc_anomaly = E_i
        return Ecc_anomaly

    def conic_trajectory(T, t_final, dt):  # Calculate conic/theoretical orbit using classical orbital elements and the orbit formula
        steps = t_final / dt
        r_conic = np.zeros((int(steps), 5))
        for j in range(int(steps)):
            Mean_anomaly = ((2 * np.pi) * (t_array[j])) / (T)
            Ea = Eccentric_Anomaly(Mean_anomaly)
            theta = 2 * np.arctan(np.sqrt((1 + e) / (1 - e)) * np.tan(Ea / 2))
            r_conic_mag = (h ** 2 / u0) * (1 / (1 + e * np.cos(theta)))
            X = r_conic_mag * (np.cos(raan) * np.cos(w + theta) - np.sin(raan) * np.sin(w + theta) * np.cos(i))  # Position X-component
            Y = r_conic_mag * (np.sin(raan) * np.cos(w + theta) + np.cos(raan) * np.sin(w + theta) * np.cos(i))  # Position Y-component
            Z = r_conic_mag * (np.sin(i) * np.sin(w + theta))  # Position Z-component
            r_conic[j, 0:5] = X, Y, Z, theta, r_conic_mag
        return r_conic

    orbit = RK4(X, t, dt)  # Simulate orbit using RK4 integration
    conic_orbit = conic_trajectory(T, t, dt)  # Simulate orbit using orbit formula

    #------PLOT STUFF-------#
    u, v = np.mgrid[0: 2 * np.pi: 100j, 0: np.pi: 50j]
    x_sphere = Re * np.cos(u) * np.sin(v)
    y_sphere = Re * np.sin(u) * np.sin(v)
    z_sphere = Re * np.cos(v)

    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    ax.set_aspect("equal")
    ax.plot(conic_orbit[:, 0], conic_orbit[:, 1], conic_orbit[:, 2], color="red")
    ax.plot(orbit[:, 0], orbit[:, 1], orbit[:, 2], color="darkblue")
    ax.plot_surface(x_sphere, y_sphere, z_sphere, rstride=3, cstride=3, color='none', edgecolor='k', shade=0)
    set_axes_equal(ax)
    plt.show()

    #print((orbit[:30, 4]), (conic_orbit[:30, 4]))
    print(np.max(np.abs(orbit[:, 4] - conic_orbit[:, 4])))  # Find the largest deviation in position between RK4 simulation and Conic/Orbit formula simulation


def Orbit_Integrator_RK4_J2_J3(a, e, i, raan, w, ta):

    # Calculate State Vectors r & v using the COE2RV I created above
    r_sat, v_sat = COE2RV(a, e, i, raan, w, ta)  # Position state vector (km) and Velocity state vector (km/s) of satellite in the geocentric equitorial frame (km)

    t = 100 * T
```

```
dt = 10
X = np.array([r_sat[0], r_sat[1], r_sat[2], v_sat[0], v_sat[1], v_sat[2]])

def force_perturbed(X, t):  # Calculates acceleration vector, including J2 and J3
    r = X[0:3]
    r_norm = np.linalg.norm(r)

    # J2 Perturbing Acceleration
     aj2_coefficient = (-3.0 / 2.0) * J2 * (u0 / r_norm ** 2) * (Re / r_norm) ** 2
    aj2x = aj2_coefficient * (1 - 5 * ((r[2] / r_norm) ** 2)) * (r[0] / r_norm)
    aj2y = aj2_coefficient * (1 - 5 * ((r[2] / r_norm) ** 2)) * (r[1] / r_norm)
    aj2z = aj2_coefficient * (3 - 5 * ((r[2] / r_norm) ** 2)) * (r[2] / r_norm)
    aj2 = np.array([aj2x, aj2y, aj2z])

    # J3 Perturbing Acceleration (From Schaub H, Junkins, JL. 2009. "Analytical Mechanics of Space Systems" pg. 380)
     aj3_coefficient = (-1.0 / 2.0) * J3 * (u0 / r_norm ** 2) * (Re / r_norm) ** 3
    aj3x = aj3_coefficient * (5 * (7 * (r[2] / r_norm) ** 3 - 3 * (r[2] / r_norm)) * (r[0] / r_norm))
    aj3y = aj3_coefficient * (5 * (7 * (r[2] / r_norm) ** 3 - 3 * (r[2] / r_norm)) * (r[1] / r_norm))
    aj3z = aj3_coefficient * (3 * (10 * (r[2] / r_norm) ** 2 - (35.0 / 3.0) * (r[2] / r_norm) ** 4 - 1))
    aj3 = np.array([aj3x, aj3y, aj3z])

    dr = np.zeros((3))
    dv = np.zeros((3))
    dr[:] = X[3:6]
    dv[:] = (-u0 / r_norm ** 3) * r + aj2 + aj3  # ACCOUNT FOR J2 AND J3 PERTURBING ACCELERATIONS HERE

    X_dot = np.array([dr[0], dr[1], dr[2], dv[0], dv[1], dv[2]])
    return X_dot

def RK4_algorithm_perturbed(X, t, dt):  # This function calculates 1 time-step forward using Runge-Kutta 4 Integration Scheme

    k1 = force_perturbed(X, t)
    k2 = force_perturbed(X + dt * k1 / 2, t + dt / 2)
    k3 = force_perturbed(X + dt * k2 / 2, t + dt / 2)
    k4 = force_perturbed(X + dt * k3, t + dt)

    r_update = X + (dt / 6) * (k1 + 2 * k2 + 2 * k3 + k4)
    t_update = t + dt
    return(r_update, t_update)

def RK4_perturbed(X, t_final, dt):  # This function loops the RK4_algorithm function to integrate over the whole time of the simulation
    steps = t_final / dt
    r_array = np.zeros((int(steps), 4))
    t = 0
    for i in range(int(steps)):
        r_star, t_update, = RK4_algorithm_perturbed(X, t, dt)
        X = r_star
        t = t_update
        r_array[i, 0] = X[0]
        r_array[i, 1] = X[1]
```

```
            r_array[i, 2] = X[2]
            r_array[i, 3] = t
        r_update = r_star
        return r_array

    orbit_perturbed = RK4_perturbed(X, t, dt)  # Simulate perturbed orbit

    #------PLOT STUFF-------#
    u, v = np.mgrid[0: 2 * np.pi: 100j, 0: np.pi: 50j]
    x_sphere = Re * np.cos(u) * np.sin(v)
    y_sphere = Re * np.sin(u) * np.sin(v)
    z_sphere = Re * np.cos(v)

    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    ax.set_aspect("equal")
    ax.plot(orbit_perturbed[:, 0], orbit_perturbed[:, 1], orbit_perturbed[:, 2], color="darkblue")
    ax.plot_surface(x_sphere, y_sphere, z_sphere, rstride=3, cstride=3, color='none', edgecolor='k', shade=0)
    set_axes_equal(ax)
    plt.show()


#p0 = 5.464e-10  # km/m^3 reference density; Must be outside of function for monte carlo to work


def Orbit_Decay(p0):
    a_d = Re + 200
    e_d = np.deg2rad(0)
    i_d = np.deg2rad(0)
    raan_d = np.deg2rad(50)
    w_d = np.deg2rad(90)
    ta_d = np.deg2rad(10)
    # Calculate State Vectors r & v using the COE2RV I created above
    r_sat, v_sat = COE2RV(a_d, e_d, i_d, raan_d, w_d, ta_d)  # Position state vector (km) and Velocity state vector (km/s) of satellite in the geocentric equitorial frame (km)
    T = ((2 * np.pi) / np.sqrt(mu)) * a_d ** (3.0 / 2.0)
    t = 17.2 * T
    dt = 0.5
    X = np.array([r_sat[0], r_sat[1], r_sat[2], v_sat[0], v_sat[1], v_sat[2]])

    def force_drag(X, t):  # Calculates acceleration due to atmospheric drag

        wE = np.array([0, 0, 7.2921159e-5])  # Angular Velocity vector of Earth
        r = X[0:3]
        r_norm = np.linalg.norm(r)
        v = X[3:6]
        vrel = v - np.cross(wE, r)  # Relative velocity vector of satellite with respect to rotating Earth
        vrel_norm = np.linalg.norm(vrel)
        uv = vrel / vrel_norm  # Relative velocity unit vectors

        # Drag Perturbation
        alt = r_norm - Re  # instantaneous altitude
```

```
    # print(alt)
    h0 = 180  # km reference altitude
    H = 29.740  # km scale height
    p = p0 * np.exp(- (alt - h0) / H)

    # Satellite parameters
    M = 250  # kg Mass of starlink satellite
    A = 4.0  # m^2
    Cd = 2.2  # Drag Coefficient ***This is the standard value for low earth satellites ***
    Bc = M / (Cd * A)  # Ballistic Coefficient

    Fd = -Cd * A / M * p * (1000 * vrel) ** 2 / 2 * uv  # Acceleration (yes, acceleration not force --> a = f/m. Don't mind the variable naming convention) due to the atmospheric drag force. 1000 * vrel to get m/s^2
    ad = Fd / 1000  # go back to km/s^2

    dr = np.zeros((3))
    dv = np.zeros((3))
    dr[:] = X[3:6]
    dv[:] = ((-u0 / r_norm ** 3) * r) + ad  # ACCOUNT FOR ATMOSPHERIC DRAG ACCELERATION HERE#

    X_dot = np.array([dr[0], dr[1], dr[2], dv[0], dv[1], dv[2]])
    return X_dot

def RK4_algorithm_drag(X, t, dt):  # This function calculates 1 time-step forward using Runge-Kutta 4 Integration Scheme

    k1 = force_drag(X, t)
    k2 = force_drag(X + dt * k1 / 2, t + dt / 2)
    k3 = force_drag(X + dt * k2 / 2, t + dt / 2)
    k4 = force_drag(X + dt * k3, t + dt)

    r_update = X + (dt / 6) * (k1 + 2 * k2 + 2 * k3 + k4)
    t_update = t + dt
    return(r_update, t_update)

def RK4_drag(X, t_final, dt):   # This function loops the RK4_algorithm function to integrate over the whole time of the simulation
    steps = t_final / dt
    r_array = np.zeros((int(steps), 5))
    t = 0
    for i in range(int(steps)):
        r_star, t_update, = RK4_algorithm_drag(X, t, dt)

        X = r_star
        t = t_update
        r_array[i, 0] = X[0]
        r_array[i, 1] = X[1]
        r_array[i, 2] = X[2]
        r_array[i, 3] = t
        r_array[i, 4] = np.sqrt(X[0] ** 2 + X[1] ** 2 + X[2] ** 2) - Re
        if 49.95 <= r_array[i, 4] <= 50.05:  # We want to know the time when altitude = 50km, so stop loop once altitude = 50km
            return r_array
            break
```

```
            #print(r_array[i, 4])
        r_update = r_star
        return r_array

    orbit_decay = RK4_drag(X, t, dt)  # Simulate perturbed orbit

    decay_time = orbit_decay[:, 3]
    decay_time = np.trim_zeros(decay_time)

    #plt.rcParams.update({'font.size': 15})
    #plt.plot(decay_time, orbit_decay[:len(decay_time), 4])
    #plt.xlabel('Time (sec)')
    #plt.ylabel('Altitude [m]')
    #plt.title('Time Evolution of Altitude with Atmospheric Drag')
    #plt.grid()
    #plt.show()

    print("Time to decay to 50 km with p0 = (" + str(p0) + ") is: [" + str(decay_time
        [-1]) + "] seconds")

    return decay_time
    #------PLOT STUFF-------#
    u, v = np.mgrid[0:2*np.pi:100j, 0:np.pi:50j]
    x_sphere = Re * np.cos(u) * np.sin(v)
    y_sphere = Re * np.sin(u) * np.sin(v)
    z_sphere = Re * np.cos(v)

    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    ax.set_aspect("equal")
    ax.plot(orbit_decay[:, 0], orbit_decay[:, 1], orbit_decay[:, 2], color="darkblue")
    ax.plot_surface(x_sphere, y_sphere, z_sphere, rstride=3, cstride=3, color='none',
        edgecolor='k', shade=0)
    set_axes_equal(ax)
    # plt.show()


def Monte_Carlo_Decay(p0_original):
    uncertainty = 0.1 #0.6827 #1 sigma variation
    num_of_runs = 1
    decay_times = np.zeros((num_of_runs, 3))
    for n in range(num_of_runs):
        p0 = np.random.normal(p0_original, p0_original * uncertainty)
        decay_time = Orbit_Decay(p0)
        decay_times[n, 0] = p0
        decay_times[n, 1] = decay_time[-1]
        decay_times[n, 2] = n + 1
        #print("p0 = " + str(decay_times[n, 0]))
        #print(decay_times[n, 0], decay_times[n, 1], decay_times[n, 2])

    #Calculate Uncertainty in Decay Time#
    avg_decay_time = np.average(decay_times[:, 1]) #Take average of the decay time
    deviations = np.zeros((num_of_runs))
    for j in range(num_of_runs): #Get the deviation of each decay time with respect to
        the average value
        deviations[j] = np.abs(decay_times[j, 1] - avg_decay_time)
    uncertainty = np.average(deviations[:]) #Take average of deviations
```

```python
    print("The uncertainty in time to decay with 10'%' uncertainty in p0 is: " + str(
     uncertainty))

    #Plot Monte Carlo Results#
    ax1 = plt.subplot(2, 1, 1)
    ax1.plot(decay_times[:, 0], decay_times[:, 1])
    plt.xlabel("Time to Decay (sec)")
    plt.ylabel("p0")
    plt.title("Monte Carlo Simulation for p0 and Impact on Decay Time")

    ax2 = plt.subplot(2, 1, 2)
    ax2.plot(decay_times[:, 2], decay_times[:, 1])
    plt.ylabel("Time to Decay (sec)")
    plt.xlabel("Simulations Ran")
    plt.title("Monte Carlo Decay Time vs. Simulation Runs")

    plt.grid()
    plt.show()

    return decay_times, uncertainty

#------PLOT STUFF-------#


def set_axes_equal(ax):
    ''' Make axes of 3D plot have equal scale so that spheres appear as spheres,
    cubes as cubes, etc.. This is one possible solution to Matplotlib's
    ax.set_aspect('equal') and ax.axis('equal') not working for 3D.

    Input
      ax: a matplotlib axis, e.g., as output from plt.gca().
    '''

    x_limits = ax.get_xlim3d()
    y_limits = ax.get_ylim3d()
    z_limits = ax.get_zlim3d()

    x_range = abs(x_limits[1] - x_limits[0])
    x_middle = np.mean(x_limits)
    y_range = abs(y_limits[1] - y_limits[0])
    y_middle = np.mean(y_limits)
    z_range = abs(z_limits[1] - z_limits[0])
    z_middle = np.mean(z_limits)

    # The plot bounding box is a sphere in the sense of the infinity
    # norm, hence I call half the max range the plot radius.
    plot_radius = 0.5 * max([x_range, y_range, z_range])

    ax.set_xlim3d([x_middle - plot_radius, x_middle + plot_radius])
    ax.set_ylim3d([y_middle - plot_radius, y_middle + plot_radius])
    ax.set_zlim3d([z_middle - plot_radius, z_middle + plot_radius])


#Orbit_Integrator_RK4_no_Perturbations(a, e, i, raan, w, ta)
#Orbit_Integrator_RK4_J2_J3(a, e, i, raan, w, ta)
#Orbit_Decay(5.464e-10)
Monte_Carlo_Decay(5.464e-10)
```