

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:

```
1 def COE2RV(a, e, i, RAAN, w, ta):
2     # Used https://web.archive.org/web/20160418175843/https://ccar.colorado.edu/
3     # asen5070/handouts/cart2kep2002.pdf as a reference
4     r = (a * (1 - e ** 2)) / (1 + e * np.cos(ta)) # Get position from orbit formula
5     h = np.sqrt(mu * a * (1 - e ** 2)) # Magnitude of specific angular momentum
6     X = r * (np.cos(RAAN) * np.cos(w + ta) - np.sin(RAAN) * np.sin(w + ta) * np.cos(i)
7     ) # Position X-component
8     Y = r * (np.sin(RAAN) * np.cos(w + ta) + np.cos(RAAN) * np.sin(w + ta) * np.cos(i)
9     ) # Position Y-component
10    Z = r * (np.sin(i) * np.sin(w + ta)) # Position Z-component
11    p = a * (1 - e ** 2) # Semilatus Rectum
12    X_dot = ((X * h * e) / (r * p)) * np.sin(ta) - ((h / r) * (np.cos(RAAN) * np.sin(w
13    + ta) + np.sin(RAAN) * np.cos(w + ta) * np.cos(i))) # Velocity X-component
14    Y_dot = ((Y * h * e) / (r * p)) * np.sin(ta) - ((h / r) * (np.sin(RAAN) * np.sin(w
15    + ta) - np.cos(RAAN) * np.cos(w + ta) * np.cos(i))) # Velocity Y-component
16    Z_dot = ((Z * h * e) / (r * p)) * np.sin(ta) + ((h / r) * (np.sin(i) * np.cos(w + ta)
17    )) # Velocity Z-component
18
19    r_vector_from_COE2RV = np.array([X, Y, Z]) # Put X,Y,Z into an array to create a
20    vector
21    v_vector_from_COE2RV = np.array([X_dot, Y_dot, Z_dot]) # Put X_dot, Y_dot, Z_dot
22    into an array to create a vector
23
24    #print("Position Vector (km)= " + str(r_vector_from_COE2RV))
25    #print("Velocity Vector (km/s) = " + str(v_vector_from_COE2RV))
26    return(r_vector_from_COE2RV, v_vector_from_COE2RV)
27
28 def RV2COE(r_vector, v_vector):
29     # Used "Orbital Mechanics for Engineering Students" 3rd Edition by Howard D.
30     Curtis; Also used our class notes
```

```

25 r = np.linalg.norm(r_vector) # Distance
26 v = np.linalg.norm(v_vector) # MAgnitude of Velocity or Speed
27 v_r = np.dot(v_vector, r_vector) / r # radial velocity
28 h_vector = np.cross(r_vector, v_vector) # specific angular momentum vector
29 h = np.linalg.norm(h_vector) # magnitude of specific angular momentum
30 i = np.arccos(h_vector[2] / h) # inclination
31 n_vector = np.cross([0, 0, 1], h_vector) # vector pointing to ascending node
32 n = np.linalg.norm(n_vector) # magnitude of n
33 if n_vector[1] > 0:
34     RAAN = np.arccos(n_vector[0] / n) # Right Ascension of the Ascending node
35 if n_vector[1] < 0:
36     RAAN = 2 * np.pi - np.arccos(n_vector[0] / n)
37 e_vector = (1 / mu) * (((v ** 2) - (mu / r)) * (r_vector)) - ((r * v_r) * v_vector)
38 # eccentricity vector
39 e = np.linalg.norm(e_vector) # eccentricity
40 if e_vector[2] > 0:
41     w = np.arccos(np.dot(n_vector, e_vector) / (n * e)) # Argument of periapse
42 if e_vector[2] < 0:
43     w = 2 * np.pi - np.arccos(np.dot(n_vector, e_vector) / (n * e))
44 if v_r > 0:
45     ta = np.arccos((np.dot(e_vector, r_vector) / (e * r))) # True anomaly
46 if v_r < 0:
47     ta = 2 * pi - np.arccos((r_vector / r) * (e_vector / e))
48 Energy = (v ** 2 / 2) - (mu / r)
49 if e == 1:
50     p = h ** 2 / mu # Semilatus Rectum
51     return "Orbit is parabolic. Eccentricity = infinity"
52 else:
53     a = -mu / (2 * Energy) # Semi-major Axis
54     p = a * (1 - e ** 2)
55
56 print("Semi-major Axis (km) = " + str(a))
57 print("Eccentricity = " + str(e))
58 print("Inclination (deg) = " + str(i * (180 / np.pi)))
59 print("Right Ascension of the Ascending Node (deg) = " + str(RAAN * (180 / np.pi)))
60 print("Argument of Periapse (deg) = " + str(w * (180 / np.pi)))
61 print("True Anomaly (deg) = " + str(ta * (180 / np.pi)))
62
63 return [a, e, i, RAAN, w, ta]
64
65 mu = 398600 # Gravitational Parameter of Earth km^3/s^2
66 r_vector = np.array([-6045.0, -3490.0, 2500.0])
67 v_vector = np.array([-3.457, 6.618, 2.533])
68
69
70 COE = RV2COE(r_vector, v_vector)
71 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

$$\ddot{\mathbf{r}} = \frac{-\mu}{r^3} \mathbf{r} \quad (1)$$

into two differential equations with only one derivative in each:

$$\frac{d\mathbf{r}_i}{dt} = \mathbf{v}_i \quad (2)$$

$$\frac{d\mathbf{v}_i}{dt} = \frac{-\mu}{r^3} \mathbf{r}_i \quad (3)$$

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

```
1 # Calculate State Vectors r & v using the COE2RV I created above
2 r_sat, v_sat = COE2RV(a, e, i, raan, w, ta) # Position state vector (km) and
3 Velocity state vector (km/s) of satellite in the geocentric equatorial frame (km)
4
5 t = 5 * T #Total time for propagation
6 dt = 1 #time-step
7 t_array = np.linspace(0, t, t / dt + 1) #make a time array
8 X = np.array([r_sat [0], r_sat [1], r_sat [2], v_sat [0], v_sat [1], v_sat [2]]) #position
9 and velocity vectors
10
11 def force(X, t): #This function gets velocity and acceleration vectors from position and
12 velocity vector
13 r = X[0:3]
14 r_norm = np.linalg.norm(r)
15 dr = np.zeros((3))
16 dv = np.zeros((3))
17 dr[:] = X[3:6]
18 dv[:] = (-u0 / r_norm ** 3) * r
19 X_dot = np.array([dr[0], dr [1], dr [2], dv [0], dv [1], dv [2]])
20 return X_dot
```

Next, I use the well known RK4 algorithm of finding k_1 , k_2 , k_3 , and k_4 to calculate one step forward in \mathbf{r} by calling the "force" function I created

```
1 def RK4_algorithm(X, t, dt): #This function calculates 1 time-step forward using Runge-
2 Kutta 4 Integration Scheme
3
4 k1 = force(X, t)
5 k2 = force(X + dt * k1 / 2, t + dt / 2)
```

```

6 k3 = force(X + dt * k2 / 2, t + dt / 2)
7 k4 = force(X + dt * k3, t + dt)
8
9 r_update = X + (dt / 6) * (k1 + 2 * k2 + 2 * k3 + k4)
10 t_update = t + dt
11
12 r_mag = np.linalg.norm(r_update)
13 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.

```

1
2 def RK4(X, t_final, dt): #This function loops the RK4_algorithm function to integrate
   over the whole time of the simulation
3     steps = t_final / dt
4     r_array = np.zeros((int(steps), 5))
5     for i in range(int(steps)):
6         Mean_anomaly = ((2 * np.pi) * (t_array[i])) / (T)
7         Ea = Eccentric_Anomaly(Mean_anomaly)
8         theta = 2 * np.arctan(np.sqrt((1 + e) / (1 - e)) * np.tan(Ea / 2))
9         r_star, t_update, r_mag = RK4_algorithm(X, i, dt)
10        X = r_star
11        r_array[i, 0] = X[0]
12        r_array[i, 1] = X[1]
13        r_array[i, 2] = X[2]
14        r_array[i, 3] = theta
15        r_array[i, 4] = r_mag
16    r_update = r_star
17    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

$$r = \frac{h^2}{\mu} \frac{1}{1 + e \cos(\theta)} \quad (4)$$

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

```

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

```

```

12     Z = r_conic_mag * (np.sin(i) * np.sin(w + theta)) # Position Z-component
13     r_conic[j, 0:5] = X, Y, Z, theta, r_conic_mag
14     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.

```

1 def Eccentric_Anomaly(Me): #From HW1
2     ratio = 1
3     if Me < np.pi:
4         E_i = Me + (e / 2.0)
5     if Me >= np.pi:
6         E_i = Me - (e / 2.0)
7     # Step 2) Calculate f(E_i) and f'(E_i) and get ratio f/f'. If |ratio| > tolerance,
8     calculate new E_i+1 = E_i - ratio_i and loop until |ratio| meets tolerance
9     while np.abs(ratio) > 1e-12:
10        f1 = E_i - e * np.sin(E_i) - Me
11        f2 = 1 - e * np.cos(E_i)
12        ratio = f1 / f2
13        E_i = E_i - ratio
14    Ecc_anomaly = E_i
15    return Ecc_anomaly

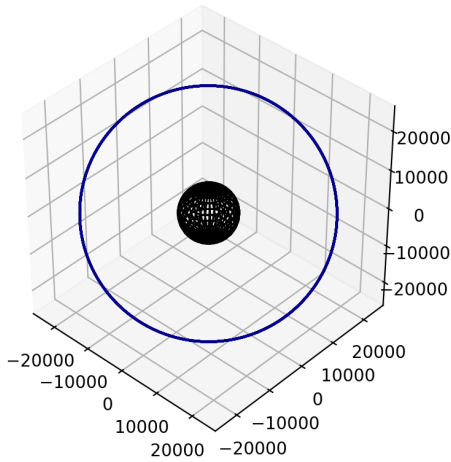
```

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), $i = 45^\circ$, $\Omega = 45^\circ$, $w = 270^\circ$, $\theta = 0^\circ$, 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 J_3 is the perturbation due to Earth's 3rd zonal harmonic and is equal to -2.53243×10^{-6} , J_2 is the perturbation due to Earth's 2nd zonal harmonic and is equal to 1.08263×10^{-3} , and R_E 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].

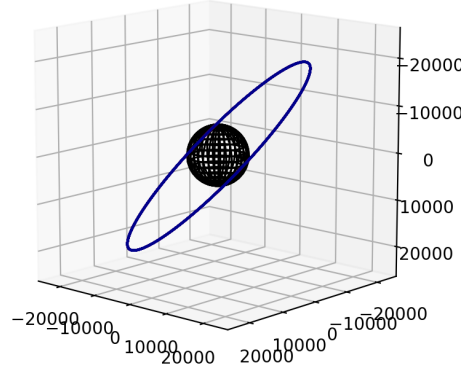
```

1 orbit = RK4(X, t, dt) #Simulate orbit using RK4 integration
2 conic_orbit = conic_trajectory(T, t, dt) #Simulate orbit using orbit formula
3 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.

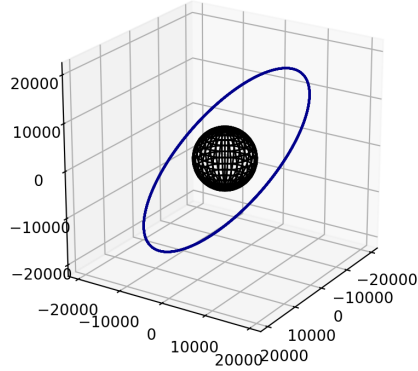
I basically recycle the code from problem 2 making small edits so the functions and variables do not get mixed up with the non-perturbed calculations. The J2 and J3 perturbing accelerations are calculated in the `force_perturbed` function that plays the same roll that `force` function does in problem 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:

```

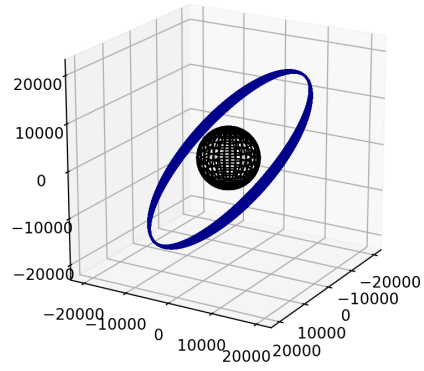
1 def force_perturbed(X, t): # Calculates acceleration vector, including J2 and J3
2     r = X[0:3]
3     r_norm = np.linalg.norm(r)
4
5     # J2 Perturbing Acceleration
6     aj2_coefficient = (-3.0 / 2.0) * J2 * (u0 / r_norm ** 2) * (Re / r_norm) ** 2
7     aj2x = aj2_coefficient * (1 - 5 * ((r[2] / r_norm) ** 2)) * (r[0] / r_norm)
8     aj2y = aj2_coefficient * (1 - 5 * ((r[2] / r_norm) ** 2)) * (r[1] / r_norm)
9     aj2z = aj2_coefficient * (3 - 5 * ((r[2] / r_norm) ** 2)) * (r[2] / r_norm)
10    aj2 = np.array([aj2x, aj2y, aj2z])
11
12    # J3 Perturbing Acceleration (From Schaub H, Junkins, JL. 2009. "Analytical
13    Mechanics of Space Systems" pg. 380)
14    aj3_coefficient = (-1.0 / 2.0) * J3 * (u0 / r_norm ** 2) * (Re / r_norm) ** 3
15    aj3x = aj3_coefficient * (5 * (7 * (r[2] / r_norm) ** 3 - 3 * (r[2] / r_norm)) * (
16    r[0] / r_norm))
17    aj3y = aj3_coefficient * (5 * (7 * (r[2] / r_norm) ** 3 - 3 * (r[2] / r_norm)) * (
18    r[1] / r_norm))
19    aj3z = aj3_coefficient * (3 * (10 * (r[2] / r_norm) ** 2 - (35.0 / 3.0) * (r[2] /
20    r_norm) ** 4 - 1))
21    aj3 = np.array([aj3x, aj3y, aj3z])
22
23    dr = np.zeros((3))
24    dv = np.zeros((3))
25    dr[:] = X[3:6]
26    dv[:] = (-u0 / r_norm ** 3) * r + aj2 + aj3 # ACCOUNT FOR J2 AND J3
27    PERTURBING ACCELERATIONS HERE
28
29    X_dot = np.array([dr[0], dr[1], dr[2], dv[0], dv[1], dv[2]])
30    return X_dot

```

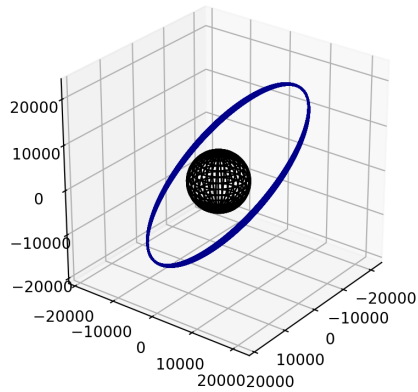
The same sequence for numerically integrating the orbit with RK4 is followed as in problem 2. Some results of the perturbed RK4 integration are shown below.



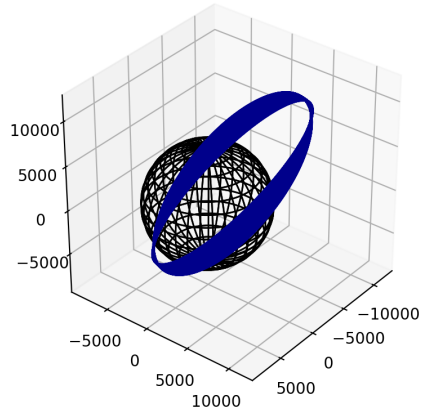
(a) Simulation of 10 orbits with J2 and J3 perturbations present, using same COE as problem 2



(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 2¹. 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-

¹It 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 an 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:

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

²For 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 $5.464 \times 10^{-10} \text{ km/m}^3$) 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) = 4 m^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):

$$B_c = \frac{M}{C_d A} = 28.4091 \text{ kg/m}^2 \quad (6)$$

The perturbing acceleration due to atmospheric drag is

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

$$= -\frac{1}{2} \frac{\rho \mathbf{V}^2}{B_c} \quad (8)$$

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

$$\mathbf{V} = \mathbf{V}_{\text{satellite}} - \omega_{\text{Earth}} \times \mathbf{r}_{\text{satellite}} \quad (9)$$

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:

```

1 def force_drag(X, t): # Calculates acceleration due to atmospheric drag
2
3     wE = np.array([0, 0, 7.2921159e-5]) # Angular Velocity vector of Earth
4     r = X[0:3]
5     r_norm = np.linalg.norm(r)
6     v = X[3:6]
7     vrel = v - np.cross(wE, r) # Relative velocity vector of satellite with respect
8     to rotating Earth
9     vrel_norm = np.linalg.norm(vrel)
10    uv = vrel / vrel_norm # Relative velocity unit vectors
11
12    # Drag Perturbation
13    alt = r_norm - Re # instantaneous altitude
14    # print(alt)
15    h0 = 180 # km reference altitude
16    H = 29.740 # km scale height
17    p = p0 * np.exp(-(alt - h0) / H)
18
19    # Satellite parameters
20    M = 250 # kg Mass of starlink satellite
21    A = 4.0 # m^2
22    Cd = 2.2 # Drag Coefficient ***This is the standard value for low earth
    satellites ***
    Bc = M / (Cd * A) # Ballistic Coefficient

```

```

23
24     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
25     ad = Fd / 1000 # go back to km/s^2
26
27     dr = np.zeros((3))
28     dv = np.zeros((3))
29     dr[:] = X[3:6]
30     dv[:] = ((-u0 / r_norm ** 3) * r) + ad # ACCOUNT FOR ATMOSPHERIC
DRAG ACCELERATION HERE#
31
32     X_dot = np.array([dr[0], dr[1], dr[2], dv[0], dv[1], dv[2]])
33     return X_dot

```

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

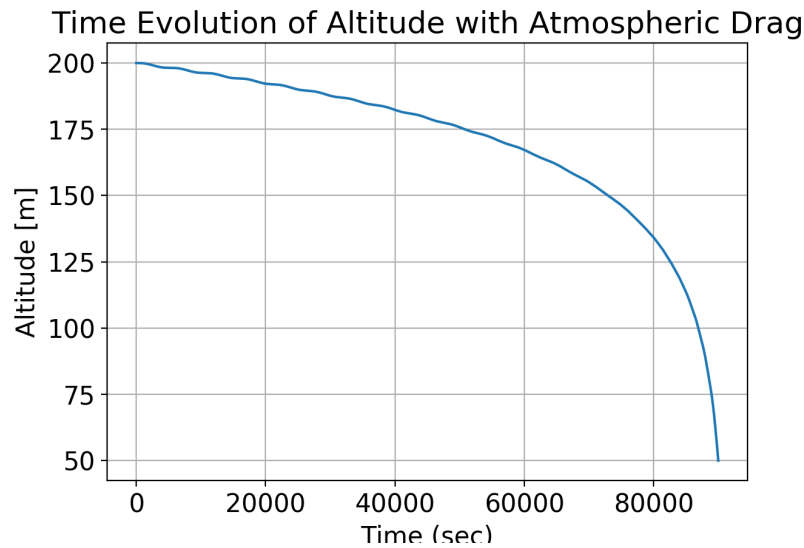


Figure 3: Plot displaying the decrease in altitude from a 200km equatorial circular orbit down to 50km due to atmospheric drag

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

```

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

```

```

10 decay_times[n, 2] = n + 1
11 #print("p0 = " + str(decay_times[n, 0]))
12 #print(decay_times[n, 0], decay_times[n, 1], decay_times[n, 2])
13
14 #Calculate Uncertainty in Decay Time#
15 avg_decay_time = np.average(decay_times[:, 1]) #Take average of the decay time
16 deviations = np.zeros((num_of_runs))
17 for j in range(num_of_runs): #Get the deviation of each decay time with respect to
    the average value
18     deviations[j] = np.abs(decay_times[j, 1] - avg_decay_time)
19 uncertainty = np.average(deviations[:]) #Take average of deviations
20 print("The uncertainty in time to decay with 10%' uncertainty in p0 is: " + str(
    uncertainty))
21
22 #Plot Monte Carlo Results#
23 ax1 = plt.subplot(2, 1, 1)
24 ax1.plot(decay_times[:, 0], decay_times[:, 1])
25 plt.xlabel("Time to Decay (sec)")
26 plt.ylabel("p0")
27 plt.title("Monte Carlo Simulation for p0 and Impact on Decay Time")
28
29 ax2 = plt.subplot(2, 1, 2)
30 ax2.plot(decay_times[:, 2], decay_times[:, 1])
31 plt.ylabel("Time to Decay (sec)")
32 plt.xlabel("Simulations Ran")
33 plt.title("Monte Carlo Decay Time vs. Simulation Runs")
34
35 plt.grid()
36 plt.show()
37
38 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**.

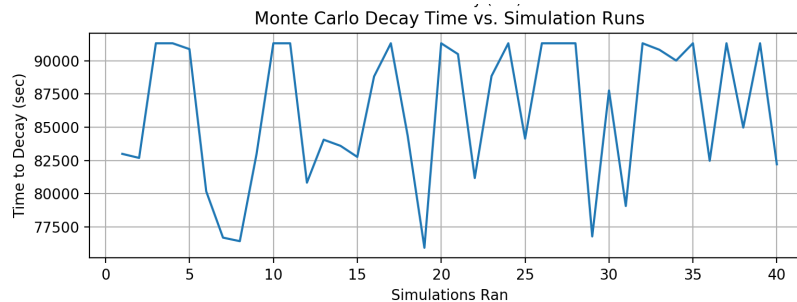


Figure 4: Variation in Decay time for each Monte Carlo simulation ran

```

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: 4735.832499999995
[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

```

1 # Pierce Jackson
2 # Applied Orbital Mechanics, Davide Guzzetti, September 30, 2020
3 # HW 3
4 import numpy as np
5 import scipy as sci
6 import scipy.stats as stats
7 import scipy.integrate # ode solver solve_ivp(function, t_span, y0) : tspan is interval
  of integration
8 from matplotlib import pyplot as plt
9 from mpl_toolkits.mplot3d import Axes3D
10 ## Problem 1 – Create algorithms to convert from Classical Orbital Elements -->
  Position and Velocity Vectors (COE2RV) and vice versa (RV2COE) ##
11
12
13 def COE2RV(a, e, i, RAAN, w, ta):
14     # Used https://web.archive.org/web/20160418175843/https://ccar.colorado.edu/
  asen5070/handouts/cart2kep2002.pdf as a reference
15
16     r = (a * (1 - e ** 2)) / (1 + e * np.cos(ta)) # Get position from orbit formula
17     h = np.sqrt(mu * a * (1 - e ** 2)) # Magnitude of specific angular momentum
18     X = r * (np.cos(RAAN) * np.cos(w + ta) - np.sin(RAAN) * np.sin(w + ta) * np.cos(i)
  ) # Position X-component
19     Y = r * (np.sin(RAAN) * np.cos(w + ta) + np.cos(RAAN) * np.sin(w + ta) * np.cos(i)
  ) # Position Y-component

```

```

20 Z = r * (np.sin(i) * np.sin(w + ta)) # Position Z-component
21 p = a * (1 - e ** 2) # Semilatus Rectum
22 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
23 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
24 Z_dot = ((Z * h * e) / (r * p)) * np.sin(ta) + ((h / r) * (np.sin(i) * np.cos(w + ta)
)) # Velocity Z-component
25
26 r_vector_from_COE2RV = np.array([X, Y, Z]) # Put X,Y,Z into an array to create a
vector
27 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
28
29 #print("Position Vector (km)= " + str(r_vector_from_COE2RV))
30 #print("Velocity Vector (km/s) = " + str(v_vector_from_COE2RV))
31 return(r_vector_from_COE2RV, v_vector_from_COE2RV)
32
33
34 def RV2COE(r_vector, v_vector):
35     # Used "Orbital Mechanics for Engineering Students" 3rd Edition by Howard D.
Curtis; Also used our class notes
36
37     r = np.linalg.norm(r_vector) # Distance
38     v = np.linalg.norm(v_vector) # MAgnitude of Velocity or Speed
39     v_r = np.dot(v_vector, r_vector) / r # radial velocity
40     h_vector = np.cross(r_vector, v_vector) # specific angular momentum vector
41     h = np.linalg.norm(h_vector) # magnitude of specific angular momentum
42     i = np.arccos(h_vector[2] / h) # inclination
43     n_vector = np.cross([0, 0, 1], h_vector) # vector pointing to ascending node
44     n = np.linalg.norm(n_vector) # magnitude of n
45     if n_vector[1] > 0:
46         RAAN = np.arccos(n_vector[0] / n) # Right Ascension of the Ascending node
47     if n_vector[1] < 0:
48         RAAN = 2 * np.pi - np.arccos(n_vector[0] / n)
49     e_vector = (1 / mu) * (((v ** 2) - (mu / r)) * (r_vector)) - ((r * v_r) * v_vector)
# eccentricity vector
50     e = np.linalg.norm(e_vector) # eccentricity
51     if e_vector[2] > 0:
52         w = np.arccos(np.dot(n_vector, e_vector) / (n * e)) # Argument of periapse
53     if e_vector[2] < 0:
54         w = 2 * np.pi - np.arccos(np.dot(n_vector, e_vector) / (n * e))
55     if v_r > 0:
56         ta = np.arccos((np.dot(e_vector, r_vector) / (e * r))) # True anomaly
57     if v_r < 0:
58         ta = 2 * np.pi - np.arccos((r_vector / r) * (e_vector / e))
59     Energy = (v ** 2 / 2) - (mu / r)
60     if e == 1:
61         p = h ** 2 / mu # Semilatus Rectum
62         return "Orbit is parabolic. Eccentricity = infinity"
63     else:
64         a = -mu / (2 * Energy) # Semi-major Axis
65         p = a * (1 - e ** 2)
66
67     print("Semi-major Axis (km) = " + str(a))
68     print("Eccentricity = " + str(e))
69     print("Inclination (deg)= " + str(i * (180 / np.pi)))

```

```

70     print("Right Ascension of the Ascending Node (deg) = " + str(RAAN * (180 / np.pi)))
71     print("Argument of Periapse (deg)= " + str(w * (180 / np.pi)))
72     print("True Anomaly (deg) = " + str(ta * (180 / np.pi)))
73
74     return [a, e, i, RAAN, w, ta]
75
76
77 mu = 398600 # Gravitational Parameter of Earth km^3/s^2
78 r_vector = np.array([-6045.0, -3490.0, 2500.0])
79 v_vector = np.array([-3.457, 6.618, 2.533])
80
81
82 #COE = RV2COE(r_vector, v_vector)
83 #COE2RV(COE[0], COE[1], COE[2], COE[3], COE[4], COE[5])
84
85 #-----#
86 #-----#
87
88 ## Problem 2 – Numerically propagate orbit from HW2 ##
89
90 # Gravitational Parameters
91 G = 6.67408e-20 # Gravitational Constant in km^3/(kg*s^2)
92 Me = 5.9722e+24 # Mass of Earth in kg
93 Ms = 11110 # Mass of Satellite --> I have chosen the mass of the Hubble Space
      Telescope
94 J2 = 1.08263e-3
95 J3 = -2.53243e-6
96 theta_E = np.deg2rad(15.04 / 3600) # rotation rate of earth in rads/s
97 Re = 6378 # radius of Earth (km)
98 u0 = 398600 # Standard Gravitational Parameter in km^3/s^2
99
100
101 a = 22610.2 # Semi-major Axis
102 i = np.deg2rad(45) # Inclination in radians (value in function in degrees)
103 raan = np.deg2rad(45) # Right-Ascension of the Ascending Node in radians (value in
      function in degrees)
104 w = np.deg2rad(270) # Argument of Perigee in radians (value in function in degrees)
105 #ta = np.deg2rad(0)
106 ta = np.deg2rad(0.00) # True Anomaly in radians (value in function in degrees)
107 e = ((-J3 * Re) / (2 * J2 * a)) * np.sin(i) # Eccentricity of orbit in degrees TAKEN
      FROM DISCUSSION
108 # e = 0.3
109 apogee = a * (1 + e) # Apogee radius
110 perigee = a * (1 - e) # Perigee radius
111 v_perigee = np.sqrt(2 * ((u0 / perigee) - (u0 / (2 * a)))) # Velocity of spacecraft at
      perigee in km/s; from conservation of energy equation
112 h = perigee * v_perigee # specific angular momentum of spacecraft in km^2/s
113 T = ((2 * np.pi) / np.sqrt(u0)) * a ** (3.0 / 2.0) # period in seconds
114 T_hrs = T / 3600.0
115
116 #-----#
117
118 def Orbit_Integrator_RK4_no_Perturbations(a, e, i, raan, w, ta):
119
120     # Calculate State Vectors r & v using the COE2RV I created above
121     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 equatorial frame (km)

```

```

122
123 t = 5 * T # Total time for propagation
124 dt = 1 # time-step
125 t_array = np.linspace(0, t, t / dt + 1) # make a time array
126 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
127
128 def force(X, t): # This function gets velocity and acceleration vectors from position
    and velocity vector
129     r = X[0:3]
130     r_norm = np.linalg.norm(r)
131     dr = np.zeros((3))
132     dv = np.zeros((3))
133     dr [:] = X[3:6]
134     dv [:] = (-u0 / r_norm ** 3) * r
135     X_dot = np.array([dr[0], dr [1], dr [2], dv [0], dv [1], dv [2]])
136     return X_dot
137
138 def RK4_algorithm(X, t, dt): # This function calculates 1 time-step forward using
    Runge-Kutta 4 Integration Scheme
139
140     k1 = force(X, t)
141     k2 = force(X + dt * k1 / 2, t + dt / 2)
142     k3 = force(X + dt * k2 / 2, t + dt / 2)
143     k4 = force(X + dt * k3, t + dt)
144
145     r_update = X + (dt / 6) * (k1 + 2 * k2 + 2 * k3 + k4)
146     t_update = t + dt
147
148     r_mag = np.linalg.norm(r_update)
149     return(r_update, t_update, r_mag)
150
151 def RK4(X, t_final, dt): # This function loops the RK4_algorithm function to
    integrate over the whole time of the simulation
152     steps = t_final / dt
153     r_array = np.zeros((int(steps), 5))
154     for i in range((int(steps))):
155         Mean_anomaly = ((2 * np.pi) * (t_array[i])) / (T)
156         Ea = Eccentric_Anomaly(Mean_anomaly)
157         theta = 2 * np.arctan(np.sqrt((1 + e) / (1 - e)) * np.tan(Ea / 2))
158         r_star, t_update, r_mag = RK4_algorithm(X, i, dt)
159         X = r_star
160         r_array [i, 0] = X[0]
161         r_array [i, 1] = X[1]
162         r_array [i, 2] = X[2]
163         r_array [i, 3] = theta
164         r_array [i, 4] = r_mag
165     r_update = r_star
166     return r_array
167
168 def Eccentric_Anomaly(Me): # From HW1
169     ratio = 1
170     if Me < np.pi:
171         E_i = Me + (e / 2.0)
172     if Me >= np.pi:
173         E_i = Me - (e / 2.0)
174     # Step 2) Calculate f(E_i) and f'(E_i) and get ratio f/f'. If |ratio| > tolerance,

```



```

175     calculate new  $E_{i+1} = E_i - \text{ratio}_i$  and loop until  $|\text{ratio}|$  meets tolerance
176     while np.abs(ratio) > 1e-12:
177         f1 =  $E_i - e * \text{np.sin}(E_i) - M_e$ 
178         f2 =  $1 - e * \text{np.cos}(E_i)$ 
179         ratio = f1 / f2
180          $E_i = E_i - \text{ratio}$ 
181     Ecc_anomaly =  $E_i$ 
182     return Ecc_anomaly
183
184 def conic_trajectory(T, t_final, dt): # Calculate conic/theoretical orbit using
185     classical orbital elements and the orbit formula
186     steps = t_final / dt
187     r_conic = np.zeros((int(steps), 5))
188     for j in range(int(steps)):
189         Mean_anomaly = ((2 * np.pi) * (t_array[j])) / (T)
190         Ea = Eccentric_Anomaly(Mean_anomaly)
191         theta = 2 * np.arctan(np.sqrt((1 + e) / (1 - e)) * np.tan(Ea / 2))
192         r_conic_mag = (h ** 2 / u0) * (1 / (1 + e * np.cos(theta)))
193         X = r_conic_mag * (np.cos(raan) * np.cos(w + theta) - np.sin(raan) * np.sin(w
194         + theta) * np.cos(i)) # Position X-component
195         Y = r_conic_mag * (np.sin(raan) * np.cos(w + theta) + np.cos(raan) * np.sin(w
196         + theta) * np.cos(i)) # Position Y-component
197         Z = r_conic_mag * (np.sin(i) * np.sin(w + theta)) # Position Z-component
198         r_conic[j, 0:5] = X, Y, Z, theta, r_conic_mag
199     return r_conic
200
201 orbit = RK4(X, t, dt) # Simulate orbit using RK4 integration
202 conic_orbit = conic_trajectory(T, t, dt) # Simulate orbit using orbit formula
203
204 #-----PLOT STUFF-----#
205 u, v = np.mgrid[0:2 * np.pi: 100j, 0: np.pi: 50j]
206 x_sphere =  $R_e * \text{np.cos}(u) * \text{np.sin}(v)$ 
207 y_sphere =  $R_e * \text{np.sin}(u) * \text{np.sin}(v)$ 
208 z_sphere =  $R_e * \text{np.cos}(v)$ 
209
210 fig = plt.figure()
211 ax = fig.add_subplot(111, projection='3d')
212 ax.set_aspect("equal")
213 ax.plot(conic_orbit[:, 0], conic_orbit[:, 1], conic_orbit[:, 2], color="red")
214 ax.plot(orbit[:, 0], orbit[:, 1], orbit[:, 2], color="darkblue")
215 ax.plot_surface(x_sphere, y_sphere, z_sphere, rstride=3, cstride=3, color='none',
216               edgcolor='k', shade=0)
217 set_axes_equal(ax)
218 plt.show()
219
220 #print((orbit[:30, 4]), (conic_orbit[:30, 4]))
221 print(np.max(np.abs(orbit[:30, 4] - conic_orbit[:30, 4]))) # Find the largest deviation
222     in position between RK4 simulation and Conic/Orbit formula simulation
223
224 def Orbit_Integrator_RK4_J2_J3(a, e, i, raan, w, ta):
225     # Calculate State Vectors r & v using the COE2RV I created above
226     r_sat, v_sat = COE2RV(a, e, i, raan, w, ta) # Position state vector (km) and
227         Velocity state vector (km/s) of satellite in the geocentric equatorial frame (km)
228
229     t = 100 * T

```

```

225 dt = 10
226 X = np.array([r_sat [0], r_sat [1], r_sat [2], v_sat [0], v_sat [1], v_sat [2]])
227
228 def force_perturbed(X, t): # Calculates acceleration vector, including J2 and J3
229     r = X[0:3]
230     r_norm = np.linalg.norm(r)
231
232     # J2 Perturbing Acceleration
233     aj2_coefficient = (-3.0 / 2.0) * J2 * (u0 / r_norm ** 2) * (Re / r_norm) ** 2
234     aj2x = aj2_coefficient * (1 - 5 * ((r [2] / r_norm) ** 2)) * (r [0] / r_norm)
235     aj2y = aj2_coefficient * (1 - 5 * ((r [2] / r_norm) ** 2)) * (r [1] / r_norm)
236     aj2z = aj2_coefficient * (3 - 5 * ((r [2] / r_norm) ** 2)) * (r [2] / r_norm)
237     aj2 = np.array([aj2x, aj2y, aj2z])
238
239     # J3 Perturbing Acceleration (From Schaub H, Junkins, JL. 2009. "Analytical
240     Mechanics of Space Systems" pg. 380)
241     aj3_coefficient = (-1.0 / 2.0) * J3 * (u0 / r_norm ** 2) * (Re / r_norm) ** 3
242     aj3x = aj3_coefficient * (5 * (7 * (r [2] / r_norm) ** 3 - 3 * (r [2] / r_norm))) * (
243     r [0] / r_norm)
244     aj3y = aj3_coefficient * (5 * (7 * (r [2] / r_norm) ** 3 - 3 * (r [2] / r_norm))) * (
245     r [1] / r_norm)
246     aj3z = aj3_coefficient * (3 * (10 * (r [2] / r_norm) ** 2 - (35.0 / 3.0) * (r [2] /
247     r_norm) ** 4 - 1))
248     aj3 = np.array([aj3x, aj3y, aj3z])
249
250     dr = np.zeros((3))
251     dv = np.zeros((3))
252     dr [:] = X[3:6]
253     dv [:] = (-u0 / r_norm ** 3) * r + aj2 + aj3 # ACCOUNT FOR J2 AND J3
254     PERTURBING ACCELERATIONS HERE
255
256     X_dot = np.array([dr [0], dr [1], dr [2], dv [0], dv [1], dv [2]])
257     return X_dot
258
259 def RK4_algorithm_perturbed(X, t, dt): # This function calculates 1 time-step
260     forward using Runge-Kutta 4 Integration Scheme
261
262     k1 = force_perturbed(X, t)
263     k2 = force_perturbed(X + dt * k1 / 2, t + dt / 2)
264     k3 = force_perturbed(X + dt * k2 / 2, t + dt / 2)
265     k4 = force_perturbed(X + dt * k3, t + dt)
266
267     r_update = X + (dt / 6) * (k1 + 2 * k2 + 2 * k3 + k4)
268     t_update = t + dt
269     return(r_update, t_update)
270
271 def RK4_perturbed(X, t_final, dt): # This function loops the RK4_algorithm function
272     to integrate over the whole time of the simulation
273     steps = t_final / dt
274     r_array = np.zeros((int(steps), 4))
275     t = 0
276     for i in range(int(steps)):
277         r_star, t_update, = RK4_algorithm_perturbed(X, t, dt)
278         X = r_star
279         t = t_update
280         r_array [i, 0] = X[0]
281         r_array [i, 1] = X[1]

```

```

275         r_array[i, 2] = X[2]
276         r_array[i, 3] = t
277         r_update = r_star
278         return r_array
279
280 orbit_perturbed = RK4_perturbed(X, t, dt) # Simulate perturbed orbit
281
282 #-----PLOT STUFF-----#
283 u, v = np.mgrid[0: 2 * np.pi: 100j, 0: np.pi: 50j]
284 x_sphere = Re * np.cos(u) * np.sin(v)
285 y_sphere = Re * np.sin(u) * np.sin(v)
286 z_sphere = Re * np.cos(v)
287
288 fig = plt.figure()
289 ax = fig.add_subplot(111, projection='3d')
290 ax.set_aspect("equal")
291 ax.plot(orbit_perturbed[:, 0], orbit_perturbed[:, 1], orbit_perturbed[:, 2], color="
    darkblue")
292 ax.plot_surface(x_sphere, y_sphere, z_sphere, rstride=3, cstride=3, color='none',
    edgcolor='k', shade=0)
293 set_axes_equal(ax)
294 plt.show()
295
296
297 #p0 = 5.464e-10 # km/m^3 reference density; Must be outside of function for monte
    carlo to work
298
299
300 def Orbit_Decay(p0):
301     a_d = Re + 200
302     e_d = np.deg2rad(0)
303     i_d = np.deg2rad(0)
304     raan_d = np.deg2rad(50)
305     w_d = np.deg2rad(90)
306     ta_d = np.deg2rad(10)
307     # Calculate State Vectors r & v using the COE2RV I created above
308     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 equatorial frame (
        km)
309     T = ((2 * np.pi) / np.sqrt(mu)) * a_d ** (3.0 / 2.0)
310     t = 17.2 * T
311     dt = 0.5
312     X = np.array([r_sat[0], r_sat[1], r_sat[2], v_sat[0], v_sat[1], v_sat[2]])
313
314     def force_drag(X, t): # Calculates acceleration due to atmospheric drag
315
316         wE = np.array([0, 0, 7.2921159e-5]) # Angular Velocity vector of Earth
317         r = X[0:3]
318         r_norm = np.linalg.norm(r)
319         v = X[3:6]
320         vrel = v - np.cross(wE, r) # Relative velocity vector of satellite with respect
        to rotating Earth
321         vrel_norm = np.linalg.norm(vrel)
322         uv = vrel / vrel_norm # Relative velocity unit vectors
323
324         # Drag Perturbation
325         alt = r_norm - Re # instantaneous altitude

```

```

326 # print(alt)
327 h0 = 180 # km reference altitude
328 H = 29.740 # km scale height
329 p = p0 * np.exp(- (alt - h0) / H)
330
331 # Satellite parameters
332 M = 250 # kg Mass of starlink satellite
333 A = 4.0 # m^2
334 Cd = 2.2 # Drag Coefficient ***This is the standard value for low earth
satellites ***
335 Bc = M / (Cd * A) # Ballistic Coefficient
336
337 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
338 ad = Fd / 1000 # go back to km/s^2
339
340 dr = np.zeros((3))
341 dv = np.zeros((3))
342 dr[:] = X[3:6]
343 dv[:] = ((-u0 / r_norm ** 3) * r) + ad # ACCOUNT FOR ATMOSPHERIC
DRAG ACCELERATION HERE#
344
345 X_dot = np.array([dr[0], dr[1], dr[2], dv[0], dv[1], dv[2]])
346 return X_dot
347
348 def RK4_algorithm_drag(X, t, dt): # This function calculates 1 time-step forward
using Runge-Kutta 4 Integration Scheme
349
350 k1 = force_drag(X, t)
351 k2 = force_drag(X + dt * k1 / 2, t + dt / 2)
352 k3 = force_drag(X + dt * k2 / 2, t + dt / 2)
353 k4 = force_drag(X + dt * k3, t + dt)
354
355 r_update = X + (dt / 6) * (k1 + 2 * k2 + 2 * k3 + k4)
356 t_update = t + dt
357 return(r_update, t_update)
358
359 def RK4_drag(X, t_final, dt): # This function loops the RK4_algorithm function to
integrate over the whole time of the simulation
360 steps = t_final / dt
361 r_array = np.zeros((int(steps), 5))
362 t = 0
363 for i in range(int(steps)):
364     r_star, t_update, = RK4_algorithm_drag(X, t, dt)
365
366     X = r_star
367     t = t_update
368     r_array[i, 0] = X[0]
369     r_array[i, 1] = X[1]
370     r_array[i, 2] = X[2]
371     r_array[i, 3] = t
372     r_array[i, 4] = np.sqrt(X[0]**2 + X[1]**2 + X[2]**2) - Re
373     if 49.95 <= r_array[i, 4] <= 50.05: # We want to know the time when
altitude = 50km, so stop loop once altitude = 50km
374         return r_array
375     break

```

```

376         #print(r_array[i, 4])
377         r_update = r_star
378         return r_array
379
380     orbit_decay = RK4_drag(X, t, dt) # Simulate perturbed orbit
381
382     decay_time = orbit_decay[:, 3]
383     decay_time = np.trim_zeros(decay_time)
384
385     #plt.rcParams.update({'font.size': 15})
386     #plt.plot(decay_time, orbit_decay[:, len(decay_time), 4])
387     #plt.xlabel('Time (sec)')
388     #plt.ylabel('Altitude [m]')
389     #plt.title('Time Evolution of Altitude with Atmospheric Drag')
390     #plt.grid()
391     #plt.show()
392
393     print("Time to decay to 50 km with p0 = (" + str(p0) + ") is: [" + str(decay_time
394           [-1]) + "] seconds")
395
396     return decay_time
397     #-----PLOT STUFF-----#
398     u, v = np.mgrid[0: 2 * np.pi: 100j, 0: np.pi: 50j]
399     x_sphere = Re * np.cos(u) * np.sin(v)
400     y_sphere = Re * np.sin(u) * np.sin(v)
401     z_sphere = Re * np.cos(v)
402
403     fig = plt.figure()
404     ax = fig.add_subplot(111, projection='3d')
405     ax.set_aspect("equal")
406     ax.plot(orbit_decay[:, 0], orbit_decay[:, 1], orbit_decay[:, 2], color="darkblue")
407     ax.plot_surface(x_sphere, y_sphere, z_sphere, rstride=3, cstride=3, color='none',
408                   edgecolor='k', shade=0)
409     set_axes_equal(ax)
410     # plt.show()
411
412 def Monte_Carlo_Decay(p0_original):
413     uncertainty = 0.1 #0.6827 #1 sigma variation
414     num_of_runs = 1
415     decay_times = np.zeros((num_of_runs, 3))
416     for n in range(num_of_runs):
417         p0 = np.random.normal(p0_original, p0_original * uncertainty)
418         decay_time = Orbit_Decay(p0)
419         decay_times[n, 0] = p0
420         decay_times[n, 1] = decay_time[-1]
421         decay_times[n, 2] = n + 1
422         #print("p0 = " + str(decay_times[n, 0]))
423         #print(decay_times[n, 0], decay_times[n, 1], decay_times[n, 2])
424
425     #Calculate Uncertainty in Decay Time#
426     avg_decay_time = np.average(decay_times[:, 1]) #Take average of the decay time
427     deviations = np.zeros((num_of_runs))
428     for j in range(num_of_runs): #Get the deviation of each decay time with respect to
429         the average value
430         deviations[j] = np.abs(decay_times[j, 1] - avg_decay_time)
431     uncertainty = np.average(deviations[:]) #Take average of deviations

```

```

430 print("The uncertainty in time to decay with 10%' uncertainty in p0 is: " + str(
      uncertainty))
431
432 #Plot Monte Carlo Results#
433 ax1 = plt.subplot(2, 1, 1)
434 ax1.plot(decay_times[:, 0], decay_times[:, 1])
435 plt.xlabel("Time to Decay (sec)")
436 plt.ylabel("p0")
437 plt.title("Monte Carlo Simulation for p0 and Impact on Decay Time")
438
439 ax2 = plt.subplot(2, 1, 2)
440 ax2.plot(decay_times[:, 2], decay_times[:, 1])
441 plt.ylabel("Time to Decay (sec)")
442 plt.xlabel("Simulations Ran")
443 plt.title("Monte Carlo Decay Time vs. Simulation Runs")
444
445 plt.grid()
446 plt.show()
447
448 return decay_times, uncertainty
449
450 #-----PLOT STUFF-----#
451
452 def set_axes_equal(ax):
453     """ Make axes of 3D plot have equal scale so that spheres appear as spheres,
454         cubes as cubes, etc..  This is one possible solution to Matplotlib's
455         ax.set_aspect('equal') and ax.axis('equal') not working for 3D.
456
457         Input
458         ax: a matplotlib axis, e.g., as output from plt.gca().
459     """
460
461     x_limits = ax.get_xlim3d()
462     y_limits = ax.get_ylim3d()
463     z_limits = ax.get_zlim3d()
464
465     x_range = abs(x_limits[1] - x_limits[0])
466     x_middle = np.mean(x_limits)
467     y_range = abs(y_limits[1] - y_limits[0])
468     y_middle = np.mean(y_limits)
469     z_range = abs(z_limits[1] - z_limits[0])
470     z_middle = np.mean(z_limits)
471
472     # The plot bounding box is a sphere in the sense of the infinity
473     # norm, hence I call half the max range the plot radius.
474     plot_radius = 0.5 * max([x_range, y_range, z_range])
475
476     ax.set_xlim3d([x_middle - plot_radius, x_middle + plot_radius])
477     ax.set_ylim3d([y_middle - plot_radius, y_middle + plot_radius])
478     ax.set_zlim3d([z_middle - plot_radius, z_middle + plot_radius])
479
480
481
482 #Orbit_Integrator_RK4_no_Perturbations(a, e, i, raan, w, ta)
483 #Orbit_Integrator_RK4_J2_J3(a, e, i, raan, w, ta)
484 #Orbit_Decay(5.464e-10)
485 Monte_Carlo_Decay(5.464e-10)

```