Orbit Determination Final Project

Tory Smith, tsmith2@mit.edu (collaborated with Jackie Smith)

10 May 2023

Background

For this final project, we put together everything we had learned over the semester towards determining the orbit of a satellite given its initial state and observation data. In the final phase, we processed six days of apparent range and range-rate data, and we propagated the satellite to a seventh day. I used an Extended Kalman Filter (EKF) to update predictions based on the provided measurement data and propagated the satellite with perturbations from a 20x20 EGM-96 gravity field, a facet-based model for the drag, a cannonball model for solar radiation pressure, and luni-solar third-body effects. Figure 1, 2, and 3 describe the starting initial conditions, configuration and materials for the satellite. The ECEF ground station locations are provided in Figure 4

	Position (km)	Velocity (km/s)
i	6984.45711518852	-1.67667852227336
j	1612.2547582643	7.26143715396544
k	13.0925904314402	0.259889857225218

Figure 1: Project initial conditions for satellite propagation.

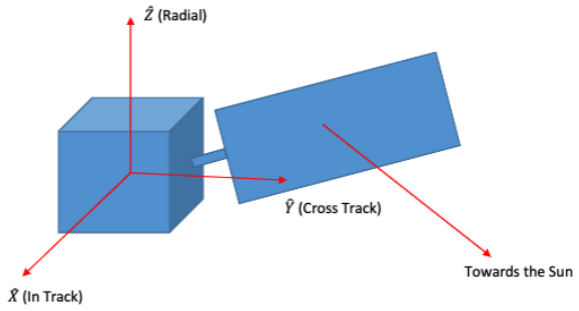

Figure 2: Satellite configuration graphic.

Component	Area	Coating
+X/-X Face	$6m^2$	MLI Kapton
+Y/-Y Face	$8m^2$	MLI Kapton
+Z/-Z Face	$12m^2$	White Paint/Germanium Kapton
Solar Panel	$15m^2$	Solar Cells

Material	C_d	C_s
Bulk S/C MLI Kapton	0.04	0.59
White Paint	0.80	0.04
Germanium Kapton	0.28	0.18
Solar Cells	0.04	0.04

Figure 3: Satellite component properties.

Number	Description	X_s (m)	Y_s (m)	Z_s (m)
1	Kwajalein	-6143584	1364250	1033743
2	Diego Garcia	1907295	6030810	-817119
3	Arecibo	2390310	-5564341	1994578

Figure 4: Ground station coordinates.

Seven cases were examined during this project. Each explores a different amount of data available for the Kalman Filter, either through variation of station, type of data, or length of time. Each of these is defined in Figure 5.

(a) fit range only for all sensors
(b) fit range-rate only for all sensors
(c) fit Kwajalein only for all data types
(d) fit Diego Garcia only for all data types
(e) fit Arecibo only for all data types
(f) fit the long-arc (all data and all sensors)
(g) fit the short arc (only the last day of data for all sensors)

Figure 5: Case descriptions for project deliverables.

Assumptions

Table 1 shows the assumptions made when designing my propagator and Kalman Filter. The range and range-rate data measurement weights were provided in the project description so I did not perform further analysis. Station range-rate biases were also provided at 0 m/s. To estimate the range-biases I removed all process noise and then ran a case with all data and sensors available to the filter and observed the first 86400 seconds of filtering. From this, it was obvious that Arecibo had a bias in range, while there didn't seem to be a significant

bias from Kwajelien or Diego Garcia. This is shown in Figure 6. A suggested starting position covariance of 10 km and 10 m/s was used in the initial analysis and was varied between 1-100km for position and 1-100 m/s for velocity with the most accurate estimation remaining at 10 km and 10 m/s respectively. Using a state noise compensation model, large σ values of 1E-9 were used to compensate for unmodeled dynamics when using simple dynamics models, such as J2 only and cannon ball for both drag and SRP, and then were subsequently decreased as higher fidelity models were added until there were diminishing returns in accuracy.

I only used the position and velocity of the satellite when estimating its state. Doing so was sufficient enough to achieve a relatively accurate result. These parameters were the most visible due to there direct correlation with the range and range-rate measurements. I also considered including less observable parameters such as the biases of the station and the coefficient of drag, but the additional computational cost and debugging time needed to do so compared to the slight increase in accuracy made the pursuit less tenable.

Error Source	Baseline Analysis
Data Weights	
Kwajalein Range	10 m
Kwajalein Range-Rate	0.5 mm/s
Diego Garcia Range	5 m
Diego Garcia Range-Rate	1 mm/s
Arecibo Range	10 m
Arecibo Range-Rate	0.5 mm/s
Data Biases	
All Stations Range-Rate	0 m/s
Kwajalein Range	0 m
Diego Garcia Range	0 m
Arecibo Range	20 m
Estimated Parameters	
Position	10 km
Velocity	10 m/s
Considered Parameters	
Coefficient of Drag	
Station Biases	
Process Noise	
σ for Unmodeled Dynamics	1E-11

Table 1: Error Assumption

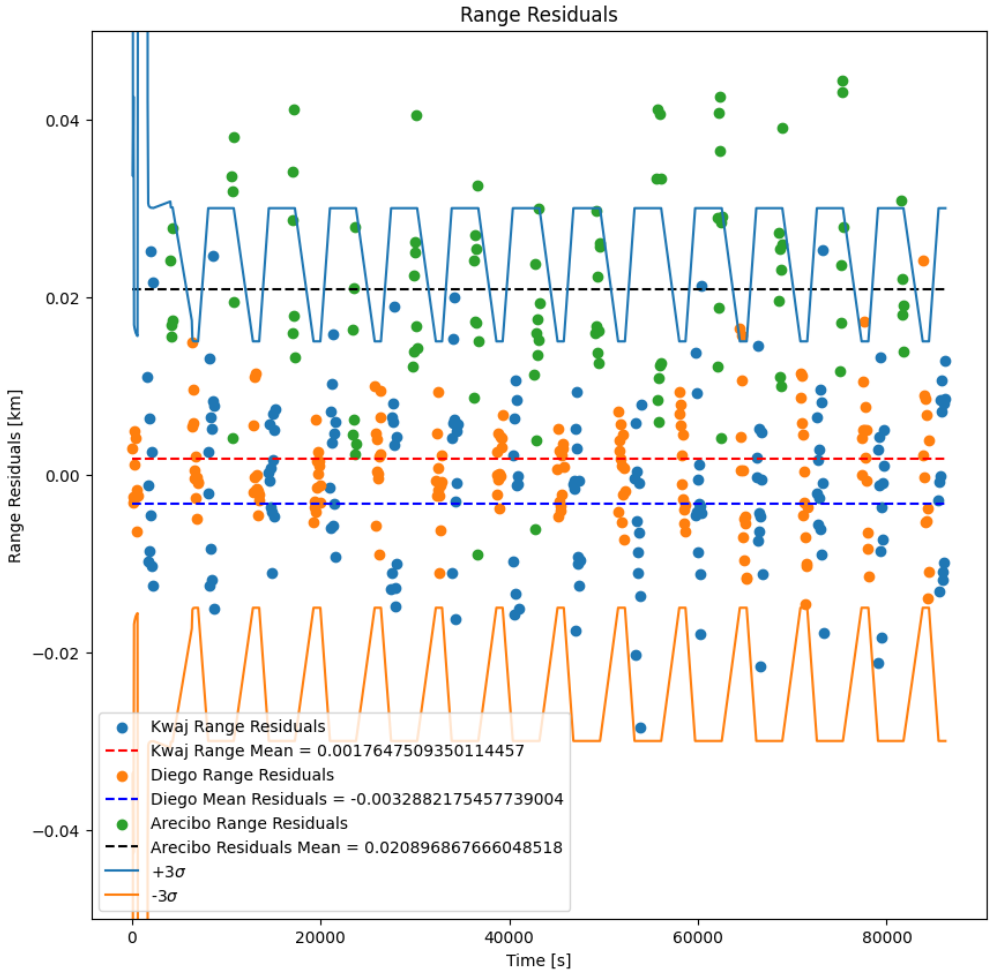

Figure 6: Range Residuals with Mean Bias for Each Station After 1 Day

Discussion

This project required significant iteration and debugging to reach a point of satisfactory accuracy. I began by building relatively simple models (J2, Cannon-ball drag and SRP, third-body effects) for propagation and testing these against a 6 hour truth value provided in HW5. I used these models as the basis for my A matrix needed to propagate the state transition matrix Φ. Once simple models and the process for deriving Φ were in place, I built functions to calculate the predicted range and range-rate from each station. From these functions, I constructed the $\tilde{H}$ matrix needed for the Innovations Covariance and Kalman Gain.

With the dynamics and measurement functions and the A and H matrices built, I moved on to the Kalman Filter. I decided to pursue an Extended Kalman Filter as this seemed the best choice with the Gaussian nature of the noise. I didn't employ a smoother since we had enough data between the beginning and the end of the process that the Kalman filter would be able to compensate for errors in the initial state.

After getting the foundations of the project set-up, I moved on to estimating the different biases and process noise needed to compensate for the unmodeled dynamics as discussed in the assumptions section. From here, I began to iterate on my dynamics models to improve their fidelity and reduce the process noise compensation. For my final iteration, I used a 20x20 EGM96 gravity field, facet based drag, a cannon-ball SRP, and third-body effects. The dynamics model seemed to be most sensitive to the changes in the acceleration due to gravity as the addition of the EGM96 gravity field had the largest effect on the accuracy over the seven day propagation, followed by the drag, SRP and the third-body effects. The Kalman filter itself seemed to be most susceptible to changes in the process noise. A larger process noise allowed the filter to capture everything within the 3σ bounds, but it would also add to inaccuracies and hide issues with my dynamics models, especially as they improved.

Exploring the different cases described in figure 5, I was able gain a better grasp of what real-world data may be like, how those different scenarios may affect results and how they could be used to identify errors and biases in other data. By limiting the Kalman Filter's access to only one type of data like in case (a) and (b) I could infer how each type of data contributes to the performance of the filter. When using only range data as in figure 8, the Kalman filter performed poorly when estimating the range-rate, however, when limited to only range-rate data as shown in figure 8 the Kalman filter was able to successfully capture the range. This implies that the range-rate contributes more information when estimating the overall state of the satellite.

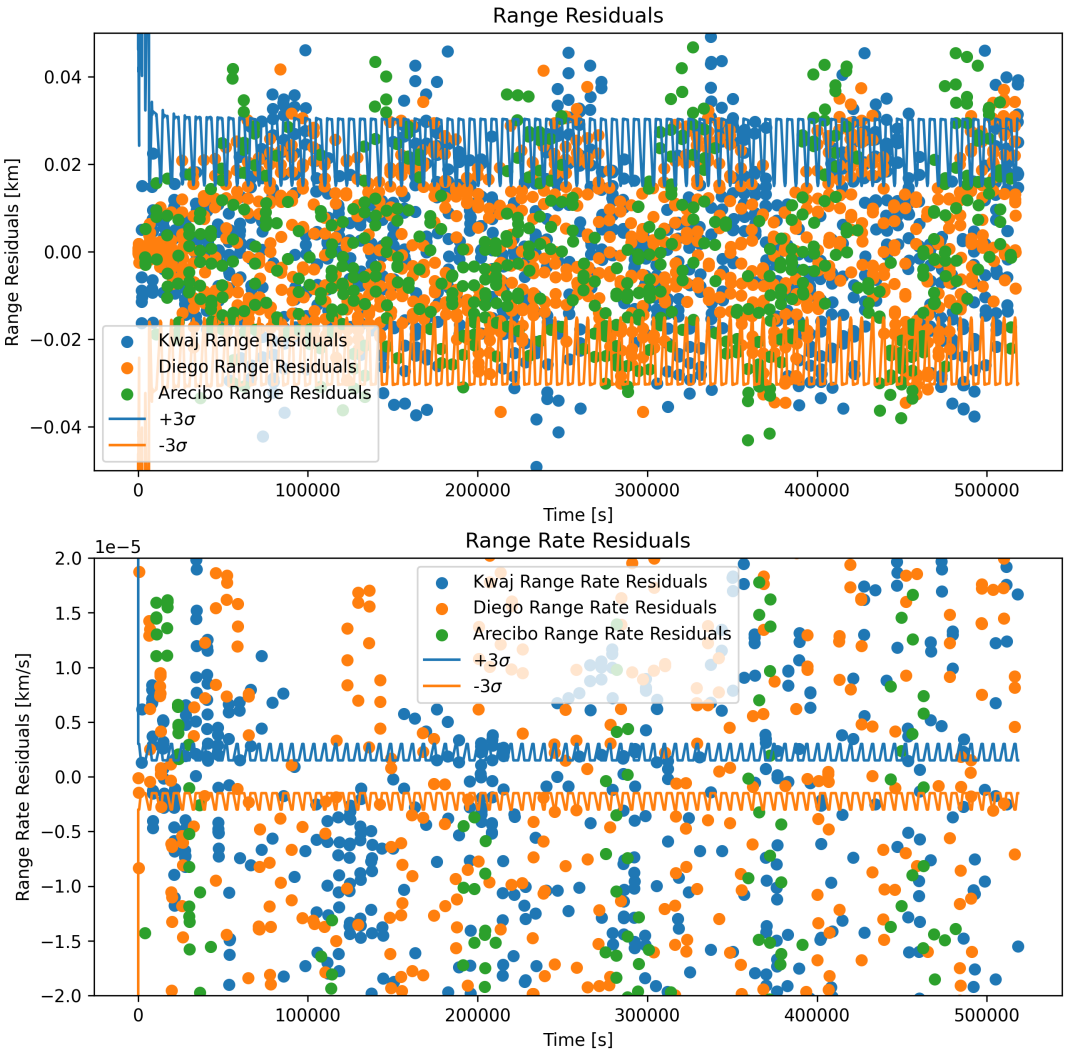

Figure 7: Case (a) Post-fit range and range-rate residuals over 7 days.

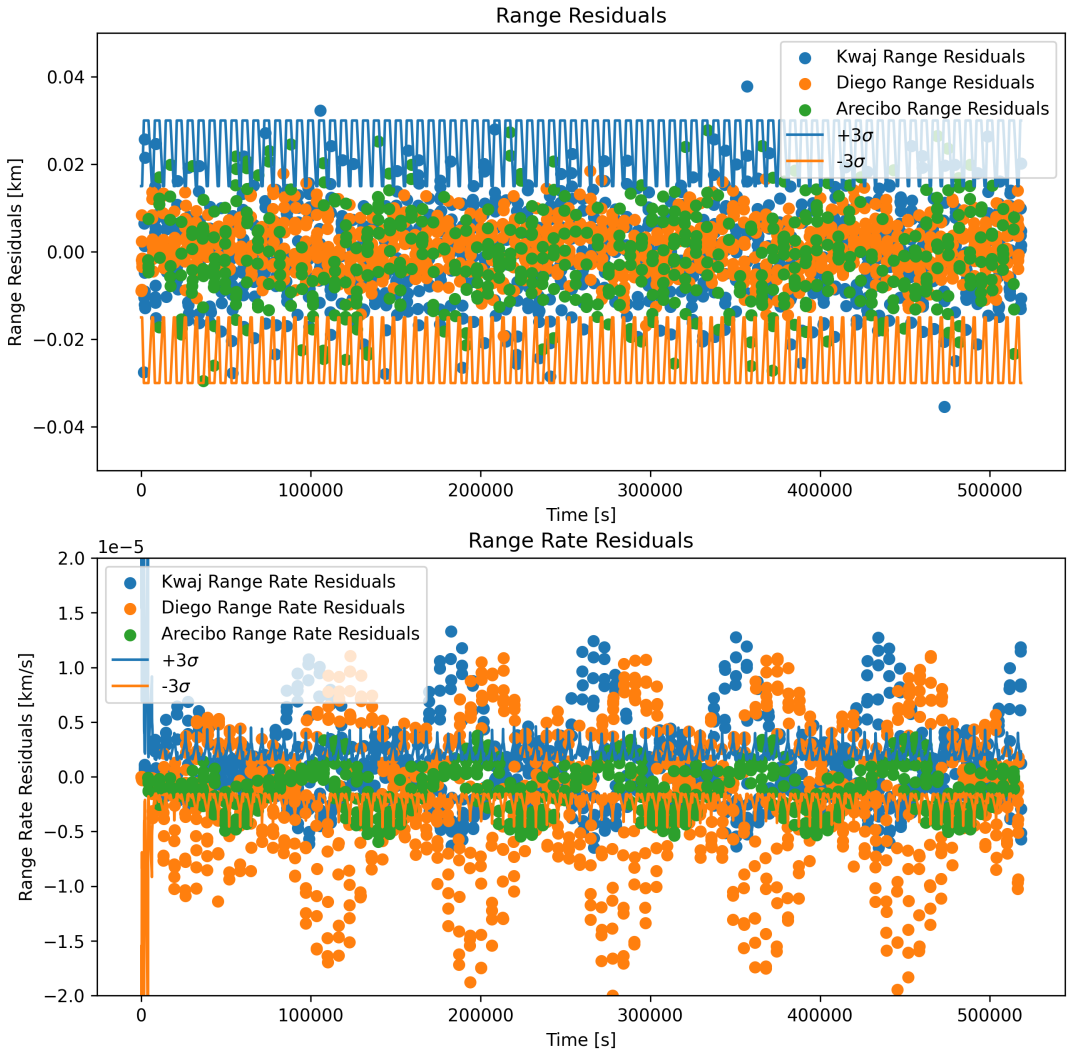

Figure 8: Case (b) Post-fit range and range-rate residuals over 7 days.

Case (c)-(e) could represent scenarios where as a satellite operator, I only have access to one sensor and must compensate for any issues that only having one data source may cause. They also assist in identifying the biases of other stations as it wouldn't be possible to identify if there were no other sources to compare against. Case (f) should provide the best estimate for a non-maneuvering space craft that is only subject to the dynamics. It makes intuitive sense that more data is better. However, if the space craft were to maneuver, the *apriori* confidence of the filter may be too high to capture it properly if multiple days of data had proceeded it. This is where case (g) would be helpful, as it wouldn't have as much confidence and could more easily capture any unpredicted changes.

Results

I calculated Range and Range-Rate residuals along with Root Mean Square (RMS) error for both the pre-fit and post-fit predicted range and range-rate associated with each station. The RMS results are presented in Table 2 and Table 3. It's notable how the pre-fit to post-fit correction is significant for Diego Garcia. This is due to it being the first station to take in observation and thus the largest initial error. Specifically, the first predicted measurement has a range-residual of 3 km, thus significantly affecting the overall RMS. We can see the filter improving its pre-fit predictions as it transitions to the other stations with Kwajalein being 2nd and still containing some of the initial error, and then the initial error nearly disappearing as it reaches Arecibo.

Station	Range RMS	Range Rate RMS
Kwajalein	0.06566 km	0.00013km/s
Diego Garcia	0.1215 km	0.0007km/s
Arecibo	0.0107 km	9.7628E-6 km/s

Table 2: Pre-fit Residual RMS for Case (f)

Station	Range RMS	Range Rate RMS
Kwajalein	0.0104 km	4.0299E-6 km/s
Diego Garcia	0.00607 km	7.0213E-6 km/s
Arecibo	0.0105 km	2.115E-6 km/s

Table 3: Post-fit Residual RMS for Case (f)

The pre-fit range and range-rate residuals for case (f) are shown in Figure 9 and the post-fit residuals are shown in Figure 10. As can be seen, the pre-fit and post fit range residuals were captured well within the 3σ bounds when using a 1E-11 process noise, however, the range-rate residuals were not fully captured. The sinusoidal structure of the range-rate residuals is most likely due to the rotation pattern of the earth not being fully captured in my ECEF to ECI transformations. The post fit residuals do show some improvement in range-rate, but not enough to completely capture them within the bounds. I could increased the process noise to better capture the range-rate residuals, but this seemed to interfere with the accuracy of my model and produce less favorable estimations.

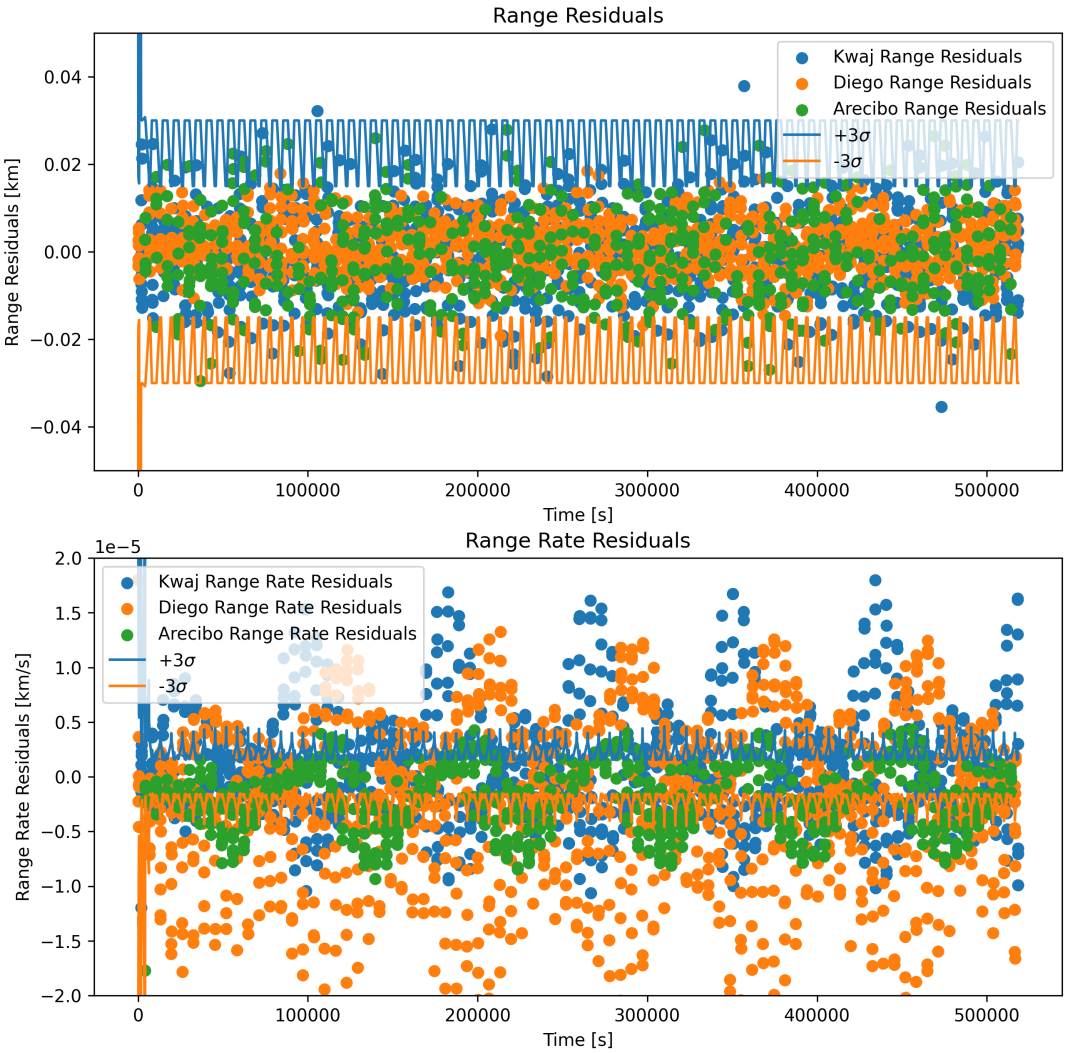

Figure 9: Case (f) Pre-fit range and range-rate residuals over 7 days.

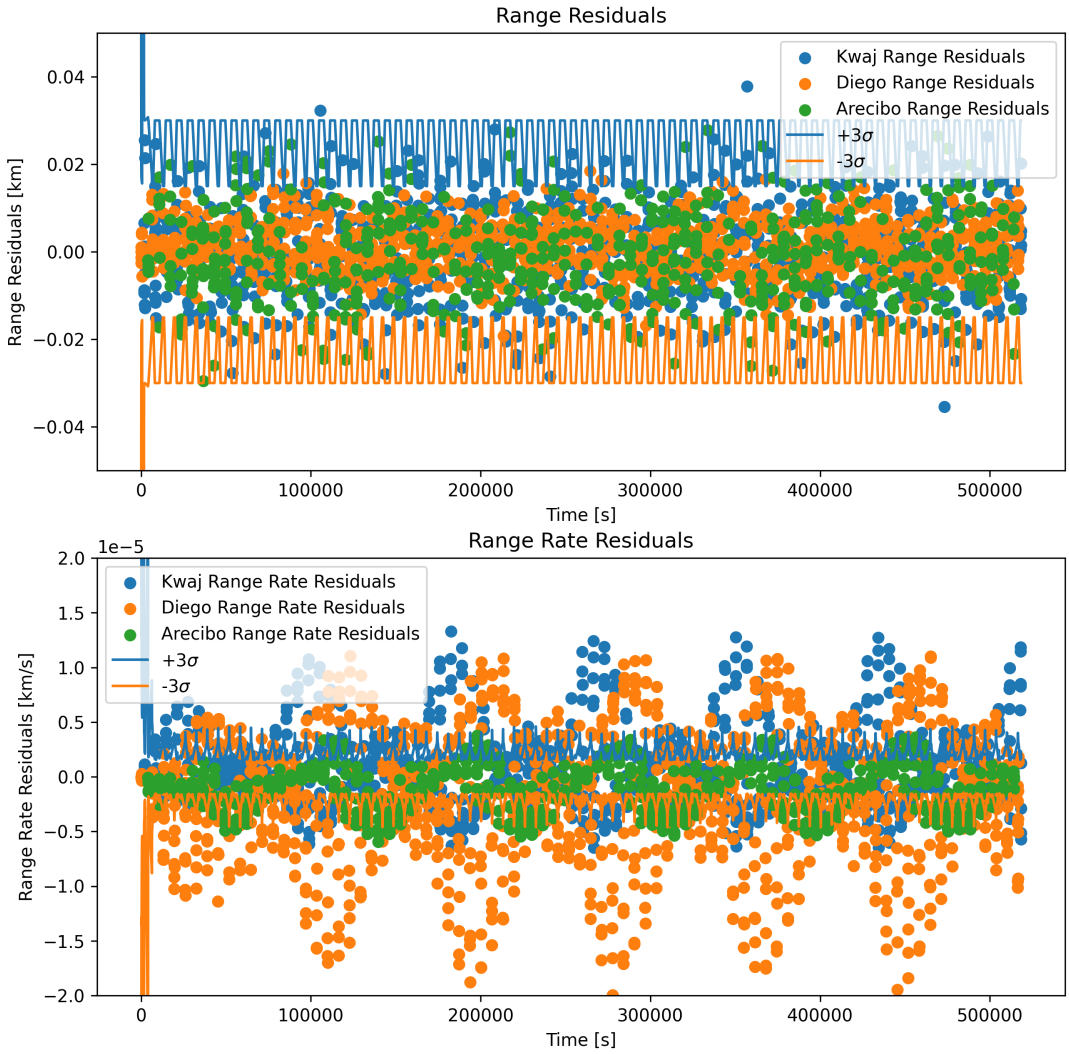

Figure 10: Case (f) Post-fit range and range-rate residuals over 7 days.

Figure 11 showcases the final predicted positions and their error ellipses for Radial vs. In Track, Radial Vs. Cross Track, and In-Track vs. Cross-Track when compared to case (f) which was my best case with an overall error of 21 m. The dynamics models performed the best in the Radial and Cross-Track directions with a larger spread in the In-Track direction. This may have been improved with the inclusion of the coefficient of drag as an estimated parameter, or using different process noise along the x, y, and z directions of the satellite. Overall, an 1E-11 process noise and the EKF with my higher fidelity gravity and drag models performed well and were able to achieve a high degree of accuracy. Table 4 shows the final position and covariance matrix for case (f).

Position	$[445.841203945569, -7100.14804466148, -183.858674932258]$	km
Covariance Matrix	$\begin{bmatrix} 0.0014 & 1.774-06 & 6.0081e-07 \\ 1.774-06 & 0.0013 & 4.3233e-08 \\ 6.008e-07 & 4.3233e-08 & 0.0014 \end{bmatrix}$	km

Table 4: Case (f) final position and covariance

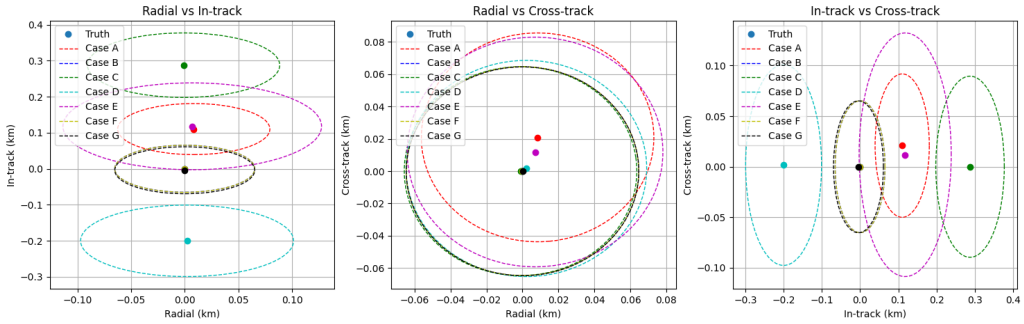

Figure 11: Radial, Cross-track, In-track error when compared to case (f)

Conclusion

This final project was the culmination of a significant amount of knowledge and understanding gained over the course of this semester. I have an immense respect for the orbit determination field and those who developed the techniques necessary to be successful in this domain. There is no doubt in my mind that what I learned in this course will be applicable for much of my career.

APPENDIX: Python Code

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint, solve_ivp
from scipy.special import lpmn
from scipy.io import savemat, loadmat
import pandas as pd
from datetime import datetime
import sympy as sym
import csv
from numba import jit

leap_sec = 37 #s
x_ps = np.array([20.816, 22.156, 23.439, 24.368, 25.676,
   26.952, 28.108, 29.400, 31.034])/1000 #arcsec
y_ps = np.array([381.008, 382.613, 384.264, 385.509, 386.420,
   387.394, 388.997, 390.776, 392.669])/1000 #arcsec
del_UT1s = np.array([144.0585, 143.1048, 142.2335, 141.3570,
   140.4078, 139.3324, 138.1510, 136.8455, 135.4165])/1000 #s
LODs = np.array([1.0293,0.9174, 0.8401, 0.8810, 1.0141, 1.1555,
    1.2568, 1.3360, 1.3925])/1000 #s


def gregorian_to_jd(year, month, day, hour, minute, second):
    '''Convert Gregorian calendar date to Julian date time.
    Input
        year : int
        month : int
        day : int
        hour : int
        minute : int
        second : float
    Returns
        jd : float '''

    a = int((14 - month)/12)
    y = year + 4800 - a
    m = month + 12*a - 3
    jd = day + int((153*m + 2)/5) + 365*y + int(y/4) - int(y
       /100) + int(y/400) - 32045
    jd = jd + (hour - 12)/24 + minute/1440 + second/86400
    return jd
```

```
JD_UTC_st = gregorian_to_jd(2018, 3, 23, 8, 55, 3)
JD_UTC_day = gregorian_to_jd(2018, 3, 23, 0, 0, 0)
JD_days = np.arange(JD_UTC_day, JD_UTC_day+9, 1)

def interp_EOP(JD_UTC):
    '''Linear interpolation of Earth Orientation Parameters (
       EOP) data

    Inputs:
    JD_UTC: time in Julian Centuries since J2000

    Outputs:
    interp_x_p: interpolated x polar motion in arcseconds
    interp_y_p: interpolated y polar motion in arcseconds
    interp_del_UT1: interpolated UT1-UTC in seconds
    interp_LOD: interpolated length of day in seconds
    '''

    interp_x_p = np.interp(JD_UTC, JD_days, x_ps)
    interp_y_p = np.interp(JD_UTC, JD_days, y_ps)
    interp_del_UT1 = np.interp(JD_UTC, JD_days, del_UT1s)
    interp_LOD = np.interp(JD_UTC, JD_days, LODs)


    return interp_x_p, interp_y_p, interp_del_UT1, interp_LOD


def orthodcm(DCM):
    '''Orthogonalize a direction cosine matrix (DCM) using the
       Rodriguez formula
    Inputs:
    DCM: 3x3 direction cosine matrix
    Outputs:
    orthoDCM: 3x3 orthogonalized direction cosine matrix
    '''
    delT = DCM @ DCM.T - np.eye(3)
    orthoDCM = DCM @ (np.eye(3) - (1/2)*delT + (1/2)*(3/4)*delT
       **2 - (1/2)*(3/4)*(5/6)*delT**3 + (1/2)*(3/4)*(5/6)
       *(7/8)*delT**4 -
                      (1/2)*(3/4)*(5/6)*(7/8)*(9/10)*delT**5 +
                         (1/2)*(3/4)*(5/6)*(7/8)*(9/10)*(11/12)*
                         delT**6 -
                      (1/2)*(3/4)*(5/6)*(7/8)*(9/10)*(11/12)
                         *(13/14)*delT**7 + (1/2)*(3/4)*(5/6)
```

```
                        *(7/8)*(9/10)*(11/12)*(13/14)*(15/16)*
                        delT**8 -
                    (1/2)*(3/4)*(5/6)*(7/8)*(9/10)*(11/12)
                        *(13/14)*(15/16)*(17/18)*delT**9 +
                        (1/2)*(3/4)*(5/6)*(7/8)*(9/10)*(11/12)
                        *(13/14)*(15/16)*(17/18)*(19/20)*delT
                        **10)
    return orthoDCM


#Rotation Martices

def Rot1(theta: float) -> np.ndarray:
    return np.array([[1, 0, 0], \
                     [0, np.cos(theta), np.sin(theta)], \
                     [0, -np.sin(theta), np.cos(theta)]])

def Rot2(theta: float) -> np.ndarray:
    return np.array([[np.cos(theta), 0, -np.sin(theta)],\
                    [0, 1, 0], \
                    [np.sin(theta), 0, np.cos(theta)]])

def Rot3(theta: float) -> np.ndarray:
    return np.array([[np.cos(theta), np.sin(theta), 0], \
                     [-np.sin(theta), np.cos(theta), 0], \
                    [0, 0, 1]])

#read in nutation data file
def read_nut80():
    # IAU1980 Theory of Nutation model
    dat_file = "nut80.dat"

    #nutaton model column names
    column_names = ['ki1', 'ki2', 'ki3', 'ki4', 'ki5', 'Aj', '
       Bj', 'Cj', 'Dj', 'j']

    #nutation dataframe
    df_nut80 = pd.read_csv(dat_file, sep="\s+", names=
       column_names)
    return df_nut80

df_nut80 = read_nut80()
Aj = df_nut80['Aj'].values
Bj = df_nut80['Bj'].values
Cj = df_nut80['Cj'].values
```

```
Dj = df_nut80['Dj'].values
k = df_nut80[df_nut80.columns[0:5]].values




def PrecessionMatrix(T_TT):
    '''
    Calculates the precession matrix

    Inputs:
    T_TT: Julian Centuries since J2000

    returns: precession matrix
    '''

    arc_sec_to_rad = np.pi/(180*3600)

     #precession angles
    C_a = (2306.2181*T_TT + 0.30188*T_TT**2 + 0.017998*T_TT**3)
       *arc_sec_to_rad
    theta_a = (2004.3109*T_TT - 0.42665*T_TT**2 - 0.041833*T_TT
       **3)*arc_sec_to_rad
    z_a = (2306.2181*T_TT + 1.09468*T_TT**2 + 0.018203*T_TT**3)
       *arc_sec_to_rad

    #precession matrix
    P = Rot3(C_a) @Rot2(-theta_a) @ Rot3(z_a)

    return P

def PolarMotionMatrix(x_p, y_p):
    '''
    Calculates the polar motion matrix

    Inputs:
    x_p: x polar motion in arcseconds
    y_p: y polar motion in arcseconds

    returns: polar motion matrix
    '''

    arc_sec_to_rad = np.pi/(180*3600)
```

```
    #radians conversions
    x_p = x_p*arc_sec_to_rad
    y_p = y_p*arc_sec_to_rad

    # Polar Motion Matrix
    W = Rot1(y_p) @ Rot2(x_p)

    return W

def SiderealTimeMatrix(T_UT1, T_TT):
    '''
    Calculates the sidereal time matrix

    Inputs:
    T_UT1: Julian Centuries since J2000
    T_TT: Julian Centuries since J2000

    returns: sidereal time matrix
    '''
    w = 7.2921151467E-5*180/np.pi #earth rotation rate in deg/s
    deg2rad = np.pi/180
    arc_sec_to_rad = np.pi/(180*3600)

    #Greenwich Mean Sidereal Time
    GMST = 67310.54841 + (876600*3600 + 8640184.812866)*T_UT1 +
        0.093104*T_UT1**2 - 6.2E-6*T_UT1**3

    #convert GMST to radians
    GMST = (GMST%86400)/240*deg2rad

    # print('GMST: ', GMST*180/np.pi)

    #anamolies
    r = 360
    Mmoon = (134.96298139 + (1325*r + 198.8673981)*T_TT +
       0.0086972*T_TT**2 + 1.78E-5*T_TT**3)
    Mdot = (357.52772333 + (99*r + 359.0503400)*T_TT -
       0.0001603*T_TT**2 - 3.3E-6*T_TT**3)
    uMoon = (93.27191028 + (1342*r + 82.0175381)*T_TT -
       0.0036825*T_TT**2 + 3.1E-6*T_TT**3)
    Ddot = (297.85036306 + (1236*r + 307.1114800)*T_TT -
       0.0019142*T_TT**2 + 5.3E-6*T_TT**3)
    lamMoon = (125.04452222 - (5*r + 134.1362608)*T_TT +
       0.0020708*T_TT**2 + 2.2E-6*T_TT**3)
```

```
    alpha = np.array([Mmoon, Mdot, uMoon, Ddot, lamMoon])*
       deg2rad

    #nutation in lam
    del_psi = np.dot((Aj*10**-4 + Bj*10**-4*T_TT)*
       arc_sec_to_rad, np.sin(np.dot(k, alpha)))

    #mean obliquity of the ecliptic
    epsilon_m = 84381.448 - 46.8150*T_TT - 0.00059*T_TT**2 +
       0.001813*T_TT**3

    #conversion to radians
    epsilon_m = epsilon_m*arc_sec_to_rad

    #equation of the equinoxes
    Eq_eq = del_psi*np.cos(epsilon_m) + 0.000063*arc_sec_to_rad
       *np.sin(2*alpha[4]) + 0.00264*arc_sec_to_rad*np.sin(
       alpha[4])

    #greenwich apparent sidereal time
    GAST = GMST + Eq_eq


    #sidereal rotation matrix
    R_mat = Rot3(-GAST)

    return R_mat


def NutationMatrix(T_TT):
    '''
    Calculates the nutation matrix

    Inputs:
    T_TT: Julian Centuries since J2000

    returns: nutation matrix
    '''

    deg2rad = np.pi/180
    arc_sec_to_rad = np.pi/(180*3600)

    #anamolies
    r = 360
```

```
    Mmoon = (134.96298139 + (1325*r + 198.8673981)*T_TT +
       0.0086972*T_TT**2 + 1.78E-5*T_TT**3)
    Mdot = (357.52772333 + (99*r + 359.0503400)*T_TT -
       0.0001603*T_TT**2 - 3.3E-6*T_TT**3)
    uMoon = (93.27191028 + (1342*r + 82.0175381)*T_TT -
       0.0036825*T_TT**2 + 3.1E-6*T_TT**3)
    Ddot = (297.85036306 + (1236*r + 307.1114800)*T_TT -
       0.0019142*T_TT**2 + 5.3E-6*T_TT**3)
    lamMoon = (125.04452222 - (5*r + 134.1362608)*T_TT +
       0.0020708*T_TT**2 + 2.2E-6*T_TT**3)
    alpha = np.array([Mmoon, Mdot, uMoon, Ddot, lamMoon])*
       deg2rad


    #nutation in lam
    del_psi = np.dot((Aj*10**-4 + Bj*10**-4*T_TT)*
       arc_sec_to_rad, np.sin(np.dot(k, alpha)))

    #nutation in obliquity
    del_epsilon = np.dot((Cj*10**-4 + Dj*10**-4*T_TT)*
       arc_sec_to_rad, np.cos(np.dot(k, alpha)))

    #mean obliquity of the ecliptic
    epsilon_m = 84381.448 - 46.8150*T_TT - 0.00059*T_TT**2 +
       0.001813*T_TT**3

    #conversion to radians
    epsilon_m = epsilon_m*arc_sec_to_rad

    #true obliquity of the ecliptic
    epsilon = epsilon_m + del_epsilon


    #nutation matrix Rot1, Rot3, Rot1
    N = Rot1(-epsilon_m) @ Rot3(del_psi) @ Rot1(epsilon)


    return N

def ECI2ECEF(r_ECI, v_ECI, JD_UTC, x_p, y_p, leap_sec, del_UT1,
    LOD):

    '''
    Converts ECI to ECEF using IAU-76/FK5
```

```
Inputs:
r_ECI: ECI position vector
JD_UTC: Julian Date in UTC
x_p: x polar motion in arc seconds
y_p: y polar motion in arc seconds
leap_sec: leap seconds
del_UT1: UT1-UTC in seconds

returns: ECI position vector
'''

# time constants
JD2000 = 2451545.0

#T_UT1
JD_UT1 = JD_UTC + del_UT1/86400
T_UT1 = (JD_UT1-JD2000)/36525

#T_TT
TAI = JD_UTC + leap_sec/86400
JD_TT = TAI + 32.184/86400
T_TT = (JD_TT-JD2000)/36525

# Polar Motion Matrix
W = PolarMotionMatrix(x_p, y_p)

# #sidereal rotation matrix
R = SiderealTimeMatrix(T_UT1, T_TT)
# # r_TOD = np.matmul(R, r_PEF)

#nutation matrix Rot1, Rot3, Rot1
N = NutationMatrix(T_TT)
# r_mod = np.matmul(N, r_TOD)

#precession matrix
P = PrecessionMatrix(T_TT)


r_ECEF = None
v_ECEF = None

if r_ECI is not None:
    r_ECEF = W.T @ R.T @ N.T @ P.T @ r_ECI
if v_ECI is not None:
    w = np.array([0, 0, 7.2921158553E-5]) #rad/s
```

```
        w_rate = w*(1-LOD/86400)

        v_ECEF = np.matmul(W.T, np.matmul(R.T, np.matmul(N.T,
           np.matmul(P.T, v_ECI) - np.cross(w_rate, np.matmul(W
           , r_ECEF)))))
    return r_ECEF, v_ECEF


def ECEF2ECI(r_ECEF, v_ECEF, a_ECEF, JD_UTC, x_p, y_p, leap_sec
   , del_UT1, LOD):

    '''
    Converts ECEF to ECI using IAU-76/FK5

    Inputs:
    r_ECEF: ECEF position vector
    JD_UTC: Julian Date in UTC
    x_p: x polar motion in arc seconds
    y_p: y polar motion in arc seconds
    leap_sec: leap seconds
    del_UT1: UT1-UTC in seconds

    returns: ECI position vector
    '''

    # time constants
    JD2000 = 2451545.0

    #T_UT1
    JD_UT1 = JD_UTC + del_UT1/86400
    T_UT1 = (JD_UT1-JD2000)/36525

    #T_UT1
    TAI = JD_UTC + leap_sec/86400
    JD_TT = TAI + 32.184/86400
    T_TT = (JD_TT-JD2000)/36525
    w = np.array([0, 0, 7.2921158553E-5]) #rad/s
    w_rate = w*(1-LOD/86400)
    # print(np.cross(w_rate, np.matmul(W, r_ECEF)))

     # Polar Motion Matrix
    W = PolarMotionMatrix(x_p, y_p)
```

```
    #sidereal rotation matrix
    R = SiderealTimeMatrix(T_UT1, T_TT)



    #nutation matrix Rot1, Rot3, Rot1
    N = NutationMatrix(T_TT)



    #precession matrix
    P = PrecessionMatrix(T_TT)


    r_ECI = None
    a_ECI = None
    v_ECI = None
    if r_ECEF is not None:
        r_ECI = np.matmul(np.matmul(np.matmul(np.matmul(P, N),
           R), W), r_ECEF)
    if v_ECEF is not None:
        # v_ECI = np.matmul(np.matmul(np.matmul(P, N), R), (np.
           matmul(W, v_ECEF) + np.cross(w_rate, np.matmul(W,
           r_ECEF))))
        v_ECI = P @ N @ R @ (W @ v_ECEF + np.cross(w_rate, W @
           r_ECEF))
    if a_ECEF is not None:
        a_ECI = P @ N @ R @ (W @ a_ECEF)

    return r_ECI, v_ECI, a_ECI

def sun_position_vector(JD_UTC, leap_sec, del_UT1):
    '''
    Returns the position vector of the sun in ECI coordinates

    Inputs:
        JD_UTC: Julian Date in UTC
        del_UT1: difference between UT1 and UTC in milliseconds
        leap_sec: number of leap seconds
    Returns:
        r_ECI: position vector of the sun in ECI coordinates
    '''

    # Constants
    deg2rad = np.pi / 180.0
```

```
au = 149597870.7 # Astronomical unit [km]
# Time variables

JD2000 = 2451545.0

#T_TT
TAI = JD_UTC + leap_sec/86400
JD_TT = TAI + 32.184/86400
T_TT = (JD_TT-JD2000)/36525

# Mean lam of the Sun
l = (280.460 + 36000.771285 * T_TT) %360

# Mean anomaly of the Sun
M = (357.528 + 35999.050957 * T_TT) %360

# Ecliptic lam of the Sun
lambda_sun = l + 1.915 * np.sin(M * deg2rad) + 0.020 * np.
   sin(2 * M * deg2rad)

# Obliquity of the ecliptic
epsilon = 23.439291 - 0.01461 * T_TT

#magnitude of the sun
R = 1.00014 - 0.01671 * np.cos(M * deg2rad) - 0.00014 * np.
   cos(2 * M * deg2rad)

#sun position vector in ecliptic coordinates
r_ecliptic = np.array([R * np.cos(lambda_sun * deg2rad),
                       R * np.cos(epsilon * deg2rad) * np.
                          sin(lambda_sun * deg2rad),
                       R * np.sin(epsilon * deg2rad) * np.
                          sin(lambda_sun * deg2rad)])

#rotation from TOD to ECI

#nutation matrix Rot1, Rot3, Rot1
N = NutationMatrix(T_TT)

# r_mod = np.matmul(N, r_TOD)

#precession matrix
P = PrecessionMatrix(T_TT)

#sun position vector in ECI
```

```
    r_ECI = np.matmul(P, np.matmul(N, r_ecliptic))*au

    return r_ECI


def moon_position_vector(JD_UTC, leap_sec, del_UT1):
    '''
    Calculates the position vector of the moon in ECI
       coordinates

    Inputs:
    JD_UTC - Julian date in UTC
    del_UT1 - UT1-UTC in ms
    leap_sec - leap seconds

    Returns:
    r_ECI - position vector of the moon in ECI coordinates
    '''

    # Constants
    deg2rad = np.pi / 180.0
    # Time variables

    JD2000 = 2451545.0

    #T_UT1
    JD_UT1 = JD_UTC + del_UT1/86400
    T_UT1 = (JD_UT1-JD2000)/36525

    #T_TT
    TAI = JD_UTC + leap_sec/86400

    JD_TT = TAI + 32.184/86400
    T_TT = (JD_TT-JD2000)/36525

    # Mean lam of the Moon
    l = (218.32 + 481267.8813*T_TT + 6.29*np.sin((134.9 +
       477198.85*T_TT)*deg2rad) \
        - 1.27*np.sin((259.2 - 413335.38*T_TT)*deg2rad) + 0.66*
           np.sin((235.7 + 890534.23*T_TT)*deg2rad) \
        + 0.21*np.sin((269.9 + 954397.70*T_TT)*deg2rad) - 0.19*
           np.sin((357.5 + 35999.05*T_TT)*deg2rad) \
        - 0.11*np.sin((186.6 + 966404.05*T_TT)*deg2rad)) % 360

    #ecliptic lattitude of the Moon
```

```
phi = (5.13*np.sin((93.3 + 483202.03*T_TT)*deg2rad) + 0.28*
   np.sin((228.2 + 960400.87*T_TT)*deg2rad) \
    - 0.28*np.sin((318.3 + 6003.18*T_TT)*deg2rad) - 0.17*np
       .sin((217.6 - 407332.20*T_TT)*deg2rad)) %360

# Horizontal parallax of the Moon
O = (0.9508 + 0.0518*np.cos((134.9 + 477198.85*T_TT)*
   deg2rad) \
    + 0.0095*np.cos((259.2 - 413335.38*T_TT)*deg2rad) +
       0.0078*np.cos((235.7 + 890534.23*T_TT)*deg2rad) \
    + 0.0028*np.cos((269.9 + 954397.70*T_TT)*deg2rad)) %
       360

#oblquity of the ecliptic
epsilon = (23.439291 - 0.0130042*T_TT - 1.64E-7*T_TT**2 +
   5.04E-7*T_TT**3) % 360
#magnitude of the vector from the Earth to the Moon
R_earth = 6378.1363 #km
r_moon = R_earth/np.sin(O*deg2rad)
#moon position vector in ecliptic coordinates
r_ecliptic = np.array([r_moon*np.cos(phi*deg2rad)*np.cos(l*
   deg2rad), \
                       r_moon*(np.cos(epsilon*deg2rad)*np.
                          cos(phi*deg2rad)*np.sin(l*deg2rad
                          ) - np.sin(epsilon*deg2rad)*np.
                          sin(phi*deg2rad)), \
                        r_moon*(np.sin(epsilon*deg2rad)*np.
                           cos(phi*deg2rad)*np.sin(l*
                           deg2rad) + np.cos(epsilon*
                           deg2rad)*np.sin(phi*deg2rad))])

#rotation from TOD to ECI

#nutation matrix Rot1, Rot3, Rot1
N = NutationMatrix(T_TT)
# r_mod = np.matmul(N, r_TOD)

#precession matrix
P = PrecessionMatrix(T_TT)

#sun position vector in ECI
r_ECI = np.matmul(P, np.matmul(N, r_ecliptic))


return r_ECI
```

```
def satellite_motion_phi(t, R, A, JD):
    '''Calculates the derivative of the state vector for the
       satellite motion
    Inputs:
        t: time in seconds
        R: state vector
        A: matrix of the linearized dynamics
        JD: Julian date in UTC

    Outputs:
        r_ddot: derivative of the position vector
        phi: derivative of the state vector

    '''

    phi = R[6:].reshape(6, 6)
    r = R[0:3]
    r_dot = R[3:6]
    x, y, z = R[0:3]
    x_dot, y_dot, z_dot = R[3:6]
    leap_sec = 37
    JD += t/86400
    # print('JD', JD)

    x_p, y_p, del_UT1, LOD = interp_EOP(JD)

    #gravity
    # r_ddot_grav = a_gravity_J2(r)
    r_ECEF, _ = ECI2ECEF(r, None, JD, x_p, y_p, leap_sec,
       del_UT1, LOD)

    g_ECEF = grav_odp(r_ECEF)
    _, _, r_ddot_grav = ECEF2ECI(None, None, g_ECEF, JD, x_p,
       y_p, leap_sec, del_UT1, LOD)

    #drag
    leap_sec = 37
    r_sun = np.zeros((1, 3))
    r_sun[0] = sun_position_vector(JD, leap_sec, del_UT1)

    A_Cross = 6E-6 #[km^2]
    C_D = 1.88
    # r_ddot_drag = a_drag(C_D, r, r_dot, A_Cross)
    r_ddot_drag = a_drag_sol(C_D, r, r_dot, r_sun[0])
```

```
    # r_ddot_drag = dragAccel2(C_D, r, r_dot, r_sun[0])


    #solar

    # d_UT1 = 196.5014/1000 #[s]
    C_s = 0.04
    C_d = 0.04
    A_Cross_sol = 6E-6 #[km^2]
    r_ddot_sol = a_solar(r, r_sun[0], C_s, C_d, A_Cross_sol)
    # r_ddot_sol = a_solar_facet(r, r_dot, r_sun[0])




    #third body
    r_moon = np.zeros((1, 3))
    r_moon[0] = moon_position_vector(JD, leap_sec, del_UT1)
    r_ddot_tb = a_third_body(r, r_sun[0], r_moon[0])



    #total acceleration
    r_ddot = r_ddot_grav + r_ddot_drag  + r_ddot_sol +
       r_ddot_tb
    #

    #A matrix
    A_1 = np.array(A(x, y, z, x_dot, y_dot, z_dot, C_D, A_Cross
       , A_Cross_sol, r_sun, r_moon))

    #state transition matrix
    phi_dot = A_1 @ phi

    dydt = np.concatenate((r_dot, r_ddot, phi_dot.ravel()))

    return dydt

def satellite_motion(t, R, JD):
    '''
    Calculates the state vector of a satellite in ECI
       coordinates

    '''
```

```
leap_sec = 37
r = R[0:3]
r_dot = R[3:6]
JD += t/86400


x_p, y_p, del_UT1, LOD = interp_EOP(JD)
#J2
# r_ddot_grav = a_gravity_J2(r)
r_ECEF, _ = ECI2ECEF(r, None, JD, x_p, y_p, leap_sec,
   del_UT1, LOD)

g_ECEF = grav_odp(r_ECEF)
# print('g_ECEF', g_ECEF)
_, _, r_ddot_grav = ECEF2ECI(None, None, g_ECEF, JD, x_p,
   y_p, leap_sec, del_UT1, LOD)


#drag
leap_sec = 37
# del_UT1 = 196.5014 #[s]
r_sun = np.zeros((1, 3))
r_sun[0] = sun_position_vector(JD, leap_sec, del_UT1)
A_Cross = 6E-6 #km^2
C_D = 1.88
# r_ddot_drag = a_drag(C_D, r, r_dot, A_Cross)
r_ddot_drag = a_drag_sol(C_D, r, r_dot, r_sun[0])
# r_ddot_drag = dragAccel2(C_D, r, r_dot, r_sun[0])

#solar

C_s = 0.04
C_d = 0.04
A_Cross_sol = 6E-6 #km^2
r_ddot_sol = a_solar(r, r_sun[0], C_s, C_d, A_Cross_sol)
# r_ddot_sol = a_solar_facet(r, r_dot, r_sun[0])


#third body
r_moon = np.zeros((1, 3))
r_moon[0] = moon_position_vector(JD, leap_sec, del_UT1)
r_ddot_tb = a_third_body(r, r_sun[0], r_moon[0])
```

```
    #total acceleration
    r_ddot = r_ddot_grav + r_ddot_drag  + r_ddot_sol +
       r_ddot_tb

    dydt = np.concatenate((r_dot, r_ddot))

    return dydt


def light_time_correction(JD, r_0, v_0, ECI_station, station):
    '''Light time correction for satellite position and
       velocity
    Inputs:
        JD_UTC: Julian date in UTC
        r_0: satellite position vector in ECI at time t
        v_0: satellite velocity vector in ECI at time t
        station: station position vector in ECEF at time t

    Outputs:
        r_0: satellite position vector in ECI at time t - lt
        v_0: satellite velocity vector in ECI at time t - lt
    '''

    x_p, y_p, del_UT1, LOD = interp_EOP(JD)
    c = 299792.458 #km/s
    rho_station = np.linalg.norm(r_0 - ECI_station)
    lt = rho_station/c
    threshold = 1E-6
    old_X_lt = r_0
    JD_lt = JD - lt/86400
    times = np.linspace(lt, 0, 11)

    y0 = np.concatenate((r_0, v_0))
    sol = solve_ivp(satellite_motion, [lt, 0], t_eval=times, y0
       =y0, args=(JD_lt,), rtol=3E-14, atol=1E-16)
    new_X_lt = sol.y[0:6, -1]
    delta = np.linalg.norm(new_X_lt[0:3] - old_X_lt)

    while delta > threshold:
        old_X_lt = new_X_lt
        x_p, y_p, del_UT1, LOD = interp_EOP(JD_lt)
        # new_station, _, _ = ECEF2ECI(station, np.array
           ([0,0,0]), None, JD_lt, x_p, y_p, leap_sec, del_UT1,
            LOD)
        new_rho = np.linalg.norm(new_X_lt[0:3] - ECI_station)
```

```
        lt = new_rho/c
        JD_lt -= lt/86400
        # print('lighttime pos diff', np.linalg.norm(new_X_lt
           [0:3] - r_0))
        times = np.linspace(lt, 0, 11)
        sol = solve_ivp(satellite_motion, [lt, 0], y0=y0,
           t_eval=times, args=(JD_lt,), rtol=3E-14, atol=1E-16)

        new_X_lt = sol.y[0:6, -1]
        delta = np.linalg.norm(new_X_lt[0:3] - old_X_lt[0:3])

    return new_X_lt

def A_Matrix(drag=True, gravity=True, solar=True, third_body=
   True):
    '''Calculates the A matrix for the equations of motion

    Inputs:
    drag: boolean, if True, drag is included in the equations
       of motion
    gravity: boolean, if True, gravity is included in the
       equations of motion
    solar: boolean, if True, solar radiation pressure is
       included in the equations of motion
    third_body: boolean, if True, third body perturbations are
       included in the equations of motion

    Outputs:
    A: A_Matrix function
    '''

    #base equation of motion
    x = sym.Symbol('x')
    y = sym.Symbol('y')
    z = sym.Symbol('z')
    A_Cross= sym.Symbol('A_Cross')
    A_Cross_Sol = sym.Symbol('A_Cross_Sol')
    x_dot = sym.Symbol('x_dot')
    y_dot = sym.Symbol('y_dot')
    z_dot = sym.Symbol('z_dot')
    C_D = sym.Symbol('C_D')
    r = (x**2 + y**2 + z**2)**(1/2)
    mu = 398600.4415 #km^3/s^2

    #with gravity
```

```
if gravity:
    R_earth = 6378.1363 #[km]
    J_2 = 0.00108248
    phi = z/r
    F_x = sym.diff(mu/r*(1-J_2*(R_earth/r)**2*(3/2*phi
       **2-1/2)), x)
    F_y = sym.diff(mu/r*(1-J_2*(R_earth/r)**2*(3/2*phi
       **2-1/2)), y)
    F_z = sym.diff(mu/r*(1-J_2*(R_earth/r)**2*(3/2*phi
       **2-1/2)), z)


#with atmospheric drag
if drag:

    R_earth = 6378.1363 #[km]
    m = 2000 #[kg]
    theta_dot = 7.292115146706979E-5 #[rad/s]
    rho_0 = 3.614E-4 #[kg/km^3]
    H = 88667.0/1E3 #[km]
    r0 = (700.0 + R_earth) #[km]

    rho_A = rho_0*sym.exp(-(r-r0)/H)

    V_A_bar = sym.Matrix([x_dot+theta_dot*y, y_dot-
       theta_dot*x, z_dot])
    V_A = sym.sqrt((x_dot + theta_dot*y)**2 + (y_dot-
       theta_dot*x)**2 + z_dot**2)

    r_ddot = -1/2*C_D*A_Cross/m*rho_A*V_A*V_A_bar
    F_x += r_ddot[0]
    F_y += r_ddot[1]
    F_z += r_ddot[2]

#with solar radiation pressure
if solar:
    r_sun = sym.MatrixSymbol('r_sun', 1, 3)
    m = 2000 #kg
    AU = 149597870.7 #m
    d = ((r_sun[0] - x)**2 + (r_sun[1] - y)**2 + (r_sun[2]
       - z)**2)**(1/2)/AU

    P = 4.56E-3 #N/km^2
    C_r = 0.63
```

```
        F_x += -P*A_Cross_Sol/m*C_r*(r_sun[0]-x)/d**2
        F_y += -P*A_Cross_Sol/m*C_r*(r_sun[1]-y)/d**2
        F_z += -P*A_Cross_Sol/m*C_r*(r_sun[2]-z)/d**2

    #with thrird body perturbations
    if third_body:
        mu_sun = 132712440018 #[km^3/s^2]
        mu_moon = 4902.800066 #[km^3/s^2]
        r_sun = sym.MatrixSymbol('r_sun', 1, 3)
        r_moon = sym.MatrixSymbol('r_moon', 1, 3)
        r_sun_mag = (r_sun[0]**2 + r_sun[1]**2 + r_sun[2]**2)
           **(1/2)
        r_moon_mag = (r_moon[0]**2 + r_moon[1]**2 + r_moon
           [2]**2)**(1/2)

        del_sun_mag = ((r_sun[0]-x)**2 + (r_sun[1]-y)**2 + (
           r_sun[2]-z)**2)**(1/2)
        del_moon_mag = ((r_moon[0]-x)**2 + (r_moon[1]-y)**2 + (
           r_moon[2]-z)**2)**(1/2)
        F_x += mu_sun*((r_sun[0]-x)/(del_sun_mag)**3 - r_sun
           [0]/r_sun_mag**3) + mu_moon*((r_moon[0]-x)/(
           del_moon_mag**3) - r_moon[0]/r_moon_mag**3)
        F_y += mu_sun*((r_sun[1]-y)/(del_sun_mag)**3 - r_sun
           [1]/r_sun_mag**3) + mu_moon*((r_moon[1]-y)/(
           del_moon_mag**3) - r_moon[1]/r_moon_mag**3)
        F_z += mu_sun*((r_sun[2]-z)/(del_sun_mag)**3 - r_sun
           [2]/r_sun_mag**3) + mu_moon*((r_moon[2]-z)/(
           del_moon_mag**3) - r_moon[2]/r_moon_mag**3)

    #F functions
    F1 = x_dot
    F2 = y_dot
    F3 = z_dot
    F4, F5, F6 = F_x, F_y, F_z
    F7 = 0

    #A matrix
    A = [[sym.diff(F1, x), sym.diff(F1, y), sym.diff(F1, z),
       sym.diff(F1, x_dot), sym.diff(F1, y_dot), sym.diff(F1,
       z_dot)],
        [sym.diff(F2, x), sym.diff(F2, y), sym.diff(F2, z), sym
           .diff(F2, x_dot), sym.diff(F2, y_dot), sym.diff(F2,
           z_dot)],
        [sym.diff(F3, x), sym.diff(F3, y), sym.diff(F3, z), sym
           .diff(F3, x_dot), sym.diff(F3, y_dot), sym.diff(F3,
```

```
            z_dot)],
        [sym.diff(F4, x), sym.diff(F4, y), sym.diff(F4, z), sym
            .diff(F4, x_dot), sym.diff(F4, y_dot), sym.diff(F4,
            z_dot)],
        [sym.diff(F5, x), sym.diff(F5, y), sym.diff(F5, z), sym
            .diff(F5, x_dot), sym.diff(F5, y_dot), sym.diff(F5,
            z_dot)],
        [sym.diff(F6, x), sym.diff(F6, y), sym.diff(F6, z), sym
            .diff(F6, x_dot), sym.diff(F6, y_dot), sym.diff(F6,
            z_dot)]]

    if gravity and drag and solar and third_body:
     A = sym.lambdify([x, y, z, x_dot, y_dot, z_dot, C_D,
        A_Cross, A_Cross_Sol, r_sun, r_moon], A)

    elif gravity:
        A = sym.lambdify([x, y, z], A)

    elif drag:
        A = sym.lambdify([x, y, z, x_dot, y_dot, z_dot, C_D,
            A_Cross], A)

    elif solar:
        A = sym.lambdify([x, y, z, A_Cross_Sol, d, m], A)

    elif third_body:
        A = sym.lambdify([x, y, z, r_sun, r_moon], A)

    else:
        A = sym.lambdify([x, y, z], A)

    return A

#H_tilde
def H_tilde_matrix():
    x = sym.Symbol('x')
    y = sym.Symbol('y')
    z = sym.Symbol('z')
    x_dot = sym.Symbol('x_dot')
    y_dot = sym.Symbol('y_dot')
    z_dot = sym.Symbol('z_dot')
    C_D = sym.Symbol('C_D')

    x_s = sym.Symbol('x_s')
    y_s = sym.Symbol('y_s')
```

```
    z_s = sym.Symbol('z_s')

    x_s_dot = sym.Symbol('x_s_dot')
    y_s_dot = sym.Symbol('y_s_dot')
    z_s_dot = sym.Symbol('z_s_dot')

    bias = sym.Symbol('bias')


    rho = sym.sqrt((x - x_s)**2 + (y - y_s)**2 + (z - z_s)**2)
       - bias
    #for project omega x r ECEF frame
    #vallado chapter 4 ECEF to ECI transformation
    rho_dot = ((x-x_s)*(x_dot-x_s_dot) + (y-y_s)*(y_dot-y_s_dot
       ) + (z-z_s)*(z_dot-z_s_dot))/rho

    H_tilde_sym = [[sym.diff(rho, x), sym.diff(rho, y), sym.
       diff(rho, z), sym.diff(rho, x_dot), sym.diff(rho, y_dot)
       , sym.diff(rho, z_dot)],[
            sym.diff(rho_dot, x), sym.diff(rho_dot, y), sym.diff
               (rho_dot, z), sym.diff(rho_dot, x_dot), sym.diff(
               rho_dot, y_dot), sym.diff(rho_dot, z_dot)]]

    H_tilde_func = sym.lambdify((x, y, z, x_dot, y_dot, z_dot,
       x_s, y_s, z_s, x_s_dot, y_s_dot, z_s_dot, bias),
       H_tilde_sym, 'numpy')

    return H_tilde_func


def a_third_body(r, r_sun, r_moon):
    '''
    Calculates the acceleration due to third body perturbations

    Inputs:
        r: position vector of satellite
        r_sun: position vector of sun
        r_moon: position vector of moon

    Outputs:
        a_x: acceleration in x direction
        a_y: acceleration in y direction
        a_z: acceleration in z direction
    '''
```

```
    mu_sun = 132712440018 #km^3/s^2
    mu_moon = 4902.800066 #km^3/s^2
    r_sun_mag = np.linalg.norm(r_sun)
    r_moon_mag = np.linalg.norm(r_moon)

    del_sun_mag = np.linalg.norm(r_sun-r)
    del_moon_mag = np.linalg.norm(r_moon-r)
    a = mu_sun*((r_sun - r)/(del_sun_mag**3) - r_sun/r_sun_mag
       **3) + mu_moon*((r_moon - r)/(del_moon_mag**3) - r_moon/
       r_moon_mag**3)

    return a


def a_solar(r, s, C_s, C_d, A_Cross_sol):
    '''
    Calculates the acceleration due to solar radiation pressure

    Inputs:
        r: position vector of satellite
        s: position vector of sun
        C_s: solar radiation pressure coefficient
        C_d: solar radiation pressure coefficient
        A_Cross_sol: cross sectional area of satellite

    Outputs:
        r_ddot: acceleration vector of satellite
    '''

    r_ddot = np.zeros(3)
    tau_min = (np.linalg.norm(r)**2 - np.dot(r, s))/(np.linalg.
       norm(r)**2 + np.linalg.norm(s)**2 - 2*np.dot(r, s))
    if tau_min < 0:
        AU = 149597870.7 #km
        d = np.linalg.norm(s-r)/AU
        m = 2000 #kg
        P = 4.56E-3 #N/km^2
        C_r = 0.63
        u = (s-r)/np.linalg.norm(s-r)
        r_ddot = -P*A_Cross_sol/m*C_r/d**2*u

    return r_ddot
```

```
def a_solar_facet(R, V, Rs):
    '''
    Calculates the acceleration due to solar radiation pressure
    '''
    x = R[0]
    y = R[1]
    z = R[2]
    rs_x = Rs[0]
    rs_y = Rs[1]
    rs_z = Rs[2]
    xd = V[0]
    yd = V[1]
    zd = V[2]
    r = np.sqrt(x ** 2 + y ** 2 + z ** 2)
    sat_vec = np.array([x, y, z])
    sun_vec = np.array([rs_x, rs_y, rs_z]).T
    srp = np.array([0, 0, 0])
    tau_min = (r**2 - np.dot(sat_vec, sun_vec))/(r**2 + np.
       linalg.norm(sun_vec)**2 - 2*np.dot(sat_vec, sun_vec))

    Cd_sp = 0.04
    Cs_sp = 0.04
    Cd_xy = 0.04
    Cs_xy = 0.59
    Cd_zpos = 0.80
    Cs_zpos = 0.04
    Cd_zneg = 0.28
    Cs_zneg = 0.18
    Asp = 15E-6
    Axx = 6E-6
    Ayy = 8E-6
    Azz = 12E-6
    theta = 0
    m_sc = 2000 #kg
    au = 149597870.7 #km
    sol_cons = 1367 #W/km^2
    c = 299792.458 #km/s


    r = np.sqrt(x**2 + y**2 + z**2)
    sat_vec = np.array([x, y, z])
    sun_vec = np.array([rs_x, rs_y, rs_z]).T
    srp = np.array([0.0, 0.0, 0.0])
    tau_min = (r**2 - np.dot(sat_vec, sun_vec))/(r**2 + np.
       linalg.norm(sun_vec)**2 - 2*np.dot(sat_vec, sun_vec))
```

```
# only apply srp if sat is in sunlight
if tau_min < 0:
    d = np.sqrt((rs_x - x)**2 + (rs_y - y)**2 + (rs_z - z)
       **2)/au
    nu_sp = Cd_sp/3
    nu_xy = Cd_xy/3
    nu_zpos = Cd_zpos/3
    nu_zneg = Cd_zneg/3
    mu_sp = Cs_sp/2
    mu_xy = Cs_xy/2
    mu_zpos = Cs_zpos/2
    mu_zneg = Cs_zneg/2
    B_sp = 2*nu_sp*np.cos(theta) + 4*mu_sp*np.cos(theta)**2
    A1_sp = Asp/m_sc
    C1 = sol_cons/c
    u = (sun_vec-sat_vec)/np.linalg.norm(sun_vec-sat_vec)

    n_hat = np.eye(3)
    r = sat_vec
    sat2sun = (sun_vec - sat_vec)
    v = np.array([xd, yd, zd]).T

    # Frame Definitions
    R_hat = r/np.linalg.norm(r)
    N_hat = np.cross(r, v)/np.linalg.norm(np.cross(r, v))
    T_hat = np.cross(N_hat, R_hat)

    # 1. define Qbf2
    Qbf2eci = np.vstack([R_hat, T_hat, N_hat]).T
    Qbf2eci = orthodcm(Qbf2eci)

    # 2. convert sat2sun in BF
    sat2sun_bf = np.linalg.inv(Qbf2eci) @ sat2sun/np.linalg
       .norm(sat2sun)

    # 3. take dot with normal to find angle
    theta_x = np.arccos(np.dot(sat2sun_bf, n_hat[:,0]))
    theta_y = np.arccos(np.dot(sat2sun_bf, n_hat[:,1]))
    theta_z = np.arccos(np.dot(sat2sun_bf, n_hat[:,2]))

    # 4. constraints for back side illumination
    if theta_z < 0:
        mu_z = mu_zneg
        nu_z = nu_zneg
```

```
            else:
                mu_z = mu_zpos
                nu_z = nu_zpos

            # 5. compute acceleration
            # for i in range(7):
            #     n_hat_eci[:,i] = Qbf2eci @ n_hat[:,i]
            #     if (np.dot(u, n_hat_eci[:,i]) != 0):
            #         brakt_term = areas[i]
            #     else:
            #         brakt_term = 0
            #     Across = Across + brakt_term
            B_x = 2*nu_xy*np.cos(theta_x) + 4*mu_xy*np.cos(theta_x)
               **2
            A1_x = Axx/m_sc

            B_y = 2*nu_xy*np.cos(theta_y) + 4*mu_xy*np.cos(theta_y)
               **2
            A1_y = Ayy/m_sc

            B_z = 2*nu_z*np.cos(theta_z) + 4*mu_z*np.cos(theta_z)
               **2
            A1_z = Azz/m_sc

            srp_sp = (-C1/d**2)*(B_sp*n_hat[:,1]*-1 + (1-mu_sp)*np.
               cos(theta)**2*sat2sun_bf)*A1_sp
            srp_x = (-C1/d**2)*(B_x*n_hat[:,0] + (1-mu_xy)*np.cos(
               theta_x)**2*sat2sun_bf)*A1_x
            srp_y = (-C1/d**2)*(B_y*n_hat[:,1] + (1-mu_xy)*np.cos(
               theta_y)**2*sat2sun_bf)*A1_y
            srp_z = (-C1/d**2)*(B_z*n_hat[:,2] + (1-mu_z)*np.cos(
               theta_z)**2*sat2sun_bf)*A1_z
            srp_tot = srp_sp + srp_x + srp_y + srp_z

            # 6. convert back to ECI
            srp = Qbf2eci @ srp_tot

    return srp


def a_drag(C_D, r, v, A_Cross):

    '''
    Computes the acceleration due to atmospheric drag
```

```
    Inputs:
    r - position vector in ECI frame [m]
    v - velocity vector in ECI frame [m/s]
    A_Cross - cross sectional area of satellite [m^2]

    Outputs:
    a_drag - acceleration due to atmospheric drag [m/s^2]

    '''

    #drag parameters
    R_earth = 6378.1363#[m]
    m = 2000 #[kg]
    theta_dot = 7.292115146706979E-5 #[rad/s]
    rho_0 = 3.614E-4#[kg/km^3]
    H = 88667.0/1E3 #[km]
    r0 = (700.0 + R_earth) #[m]

    r_mag = np.linalg.norm(r)
    rho_A = rho_0*np.exp(-(r_mag-r0)/H)
    V_A_bar = np.array([v[0]+theta_dot*r[1], v[1]-theta_dot*r
       [0], v[2]])
    V_A = np.sqrt((v[0] + theta_dot*r[1])**2 +(v[1]-theta_dot*r
       [0])**2 + v[2]**2)


    return -1/2*C_D*A_Cross/m*rho_A*V_A*V_A_bar


def a_drag_sol(C_D, r, v, s):

    '''
    Computes the acceleration due to atmospheric drag

    Inputs:
    C_D - drag coefficient
    r - position vector in ECI frame [m]
    v - velocity vector in ECI frame [m/s]
    r_sun - position vector of sun in ECI frame [m]

    Outputs:
    a_drag - acceleration due to atmospheric drag [m/s^2]

    '''
    r_mag = np.linalg.norm(r)
```

```
#drag parameters
R_earth = 6378.1363 #[m]
m = 2000 #[kg]
theta_dot = 7.292115146706979E-5 #[rad/s]


#compute the rotation angles of the satellite so that it is
    nadir pointed
phi_sat = np.arctan2(r[1], r[0])
theta_sat = np.arctan2(r[2], np.sqrt(r[0]**2 + r[1]**2))

#compute Cross sectional area of the satellite body in each
    direction
A_Cross_x = np.abs((np.pi/2 - phi_sat))/(np.pi/2)*6E-6
A_Cross_y = np.abs((phi_sat)/np.pi/2)*8E-6
A_Cross_z = np.abs((theta_sat)/np.pi/2)*12E-6

rho_0 = 3.614E-4 #[kg/km^3]
H = 88667.0/1E3 #[km]
r0 = (700.0 + R_earth) #[km]

#check if the satellite is in the shadow of the earth, if
   it is then don't add the solar panel to drag
tau_min = (np.linalg.norm(r)**2 - np.dot(r, s))/(np.linalg.
   norm(r)**2 + np.linalg.norm(s)**2 - 2*np.dot(r, s))
if tau_min < 0:

    #compute vector between sun and satellite
    r_sat_sun = s-r

    #compute phi and theta of the satellite in relation to
       the sun
    phi_sat_sun = np.arctan2(r_sat_sun[1], r_sat_sun[0])
    theta_sat_sun = np.arctan2(r_sat_sun[2], np.sqrt(
       r_sat_sun[0]**2 + r_sat_sun[1]**2))

    #compute the amount of the solar panel in front of the
       satellite
    solar_theta = (np.pi/2 - np.abs(theta_sat_sun -
       theta_sat))/(np.pi/2)*15E-6

    #compute Cross sectional area of the solar panel in
       each direction
```

```
            A_Cross_x += np.abs(np.pi/2-np.abs(phi_sat_sun -
               phi_sat))/(np.pi/2)*solar_theta
            A_Cross_y += np.abs(phi_sat_sun - phi_sat)/(np.pi/2)*
               solar_theta
            A_Cross_z += np.abs(theta_sat_sun - theta_sat)/(np.pi
               /2)*solar_theta

        # A_Cross = np.array([A_Cross_x, A_Cross_y, A_Cross_z])
        A_Cross = A_Cross_x + A_Cross_y + A_Cross_z
        rho_A = rho_0*np.exp(-(r_mag-r0)/H)
        V_A_bar = np.array([v[0]+theta_dot*r[1], v[1]-theta_dot*r
           [0], v[2]])
        V_A = np.sqrt((v[0] + theta_dot*r[1])**2 +(v[1]-theta_dot*r
           [0])**2 + v[2]**2)


        return -1/2*C_D*rho_A*V_A*A_Cross*V_A_bar/m


def a_drag_sol_2(C_D, r, v, s):

    '''
    Computes the acceleration due to atmospheric drag

    Inputs:
    C_D - drag coefficient
    r - position vector in ECI frame [m]
    v - velocity vector in ECI frame [m/s]
    r_sun - position vector of sun in ECI frame [m]

    Outputs:
    a_drag - acceleration due to atmospheric drag [m/s^2]

    '''
    # Calculate the magnitude of the position vector
    r_mag = np.linalg.norm(r)

    # Define drag parameters
    R_earth = 6378.1363 #[m]
    m = 2000 #[kg]
    theta_dot = 7.292115146706979E-5 #[rad/s]
    C_D = 1.88 # Drag coefficient

    # Calculate velocity in atmosphere
```

```
V_A_bar = np.array([v[0]+theta_dot*r[1], v[1]-theta_dot*r
   [0], v[2]])

areas = np.array([6, 6, 8, 8, 12, 12, 15])*1E-6
A_cross = 0

# Create unit vectors in body frame
n_hat = np.array([[1, 0, 0],
                  [-1, 0, 0],
                  [0, 1, 0],
                  [0, -1, 0],
                  [0, 0, 1],
                  [0, 0, -1],
                  [0, -1, 0]])
n_hat_eci = np.zeros((7, 3))

T_hat = v/np.linalg.norm(v)
W_hat = np.cross(r, v)/np.linalg.norm(np.cross(r, v))
N_hat = np.cross(T_hat, W_hat)

# Transform from body frame to ECI frame
Qbf2eci = np.vstack([T_hat, W_hat, N_hat])

Qbf2eci = orthodcm(Qbf2eci)

# Calculate components of n_hat in ECI frame
for i in range(len(n_hat)):
    n_hat_eci[i] = Qbf2eci @ n_hat[i]
    if not np.isclose(np.dot(T_hat, n_hat_eci[i]), 0, 1E
       -16):
        facet_a = areas[i]
    else:
        facet_a = 0
    A_cross += facet_a
print('A_cross: ', A_cross)



# Calculate cross sectional area for each component

# A_Cross = np.zeros(3)
# A_Cross[0] = (n_hat_eci_x @ T_hat)*(6E-6)
# A_Cross[1] = (n_hat_eci_y @ T_hat)*(8E-6)
# A_Cross[2] = (n_hat_eci_z @ T_hat)*(12E-6)
```

```
    # Define atmospheric parameters
    rho_0 = 3.614E-4 #[kg/km^3]
    H = 88667.0/1E3 #[km]
    r0 = (700.0 + R_earth) #[m]

    # # # Check if satellite is in shadow of earth
    # tau_min = (np.linalg.norm(r)**2 - np.dot(r, s))/(np.
       linalg.norm(r)**2 + np.linalg.norm(s)**2 - 2*np.dot(r, s
       ))
    # if tau_min < 0:
    # Compute vector between sun and satellite
    # r_sat_sun = (s-r)/np.linalg.norm(s-r)

    # Calculate solar panel contribution to drag
    # n_hat_bf_sol = np.cross(n_hat, r_sat_sun)
    # n_hat_eci_sol = (Qbf2eci @ n_hat_bf_sol)

    # A_Cross[0] += (n_hat_eci_sol[0] @ T_hat)*(15E-6)
    # A_Cross[1] += (n_hat_eci_sol[1] @ T_hat)*(15E-6)
    # A_Cross[2] += (n_hat_eci_sol[2] @ T_hat)*(15E-6)

    # Calculate total cross sectional area
    # A_Cross = np.array([A_Cross_x, A_Cross_y, A_Cross_z])


    # Calculate atmospheric density
    rho_A = rho_0*np.exp(-(r_mag-r0)/H)

    # Calculate velocity in atmosphere
    V_A = np.sqrt((v[0] + theta_dot*r[1])**2 +(v[1]-theta_dot*r
       [0])**2 + v[2]**2)

    # Return drag force
    return -1/2*C_D*A_cross/m*rho_A*V_A*V_A_bar

def dragAccel2(Cd, R, V, r_sun):

    x = R[0]
    y = R[1]
    z = R[2]

    xd = V[0]
    yd = V[1]
```

```
zd = V[2]

rs_x = r_sun[0]
rs_y = r_sun[1]
rs_z = r_sun[2]

R_earth = 6378.1363 #[m]
m_sc = 2000 #[kg]
w = 7.292115146706979E-5 #[rad/s]
C_D = 1.88 # Drag coefficient
rho0 = 3.614E-4 #[kg/km^3]
H = 88667.0/1E3 #[km]
r0_drag = (700.0 + R_earth) #[m]
Axx = 6E-6
Ayy = 8E-6
Azz = 12E-6
Asp = 15E-6


r = np.array([x,y,z])
v = np.array([xd, yd, zd])
r_mag = np.sqrt(x**2 + y**2 + z**2)
sun_vec = np.array([rs_x, rs_y, rs_z])
sat2sun = (sun_vec-r)/np.linalg.norm(sun_vec-r)
rho_A = rho0*np.exp(-(r_mag-r0_drag)/H)
V_A_bar = [xd + w*y, yd - w*x, zd]
V_A = np.sqrt((xd + w*y)**2 + (yd - w*x)**2 + zd**2)

n_hat = np.identity(3)

# Frame Definitions
T_hat = v/np.linalg.norm(v)
W_hat = np.cross(r, v)/np.linalg.norm(np.cross(r, v))
N_hat = np.cross(T_hat, W_hat)

# 1. define Qbf2
Qbf2eci = np.transpose(np.vstack((N_hat, T_hat, W_hat)))
Qbf2eci = orthodcm(Qbf2eci)

# 2. convert sat2sun in BF
sat2sun_bf = np.linalg.inv(Qbf2eci) @ sat2sun

# 3. take dot with normal to find angle
theta_x = np.arccos(np.abs(np.dot(T_hat, n_hat[0,:])))
theta_y = np.arccos(np.abs(np.dot(T_hat, n_hat[1,:])))
```

```
    theta_z = np.arccos(np.abs(np.dot(T_hat, n_hat[2,:])))
    theta_srp = np.arccos(np.abs(np.dot(T_hat, sat2sun_bf)))

    # 5. compute acceleration
    Across_x = np.cos(theta_x)*Axx*T_hat
    Across_y = np.cos(theta_y)*Ayy*T_hat
    Across_z = np.cos(theta_z)*Azz*T_hat
    Across_sp = np.cos(theta_srp)*Asp*T_hat
    drag_x = -1/2*Cd*rho_A*V_A*(Across_x*np.transpose(V_A_bar))
       /m_sc
    drag_y = -1/2*Cd*rho_A*V_A*(Across_y*np.transpose(V_A_bar))
       /m_sc
    drag_z = -1/2*Cd*rho_A*V_A*(Across_z*np.transpose(V_A_bar))
       /m_sc
    tau_min = (np.linalg.norm(r)**2 - np.dot(r, r_sun))/(np.
       linalg.norm(r)**2 + np.linalg.norm(r_sun)**2 - 2*np.dot(
       r, r_sun))
    if tau_min < 0:

        drag_sp = -1/2*Cd*rho_A*V_A*(Across_sp*np.transpose(
           V_A_bar))/m_sc

        drag_sp = np.zeros(3)
    drag_tot = drag_x + drag_y + drag_z + drag_sp

    # 6. convert back to ECI
    drag = Qbf2eci @ drag_tot

    return drag

def a_gravity_J2(r):
    '''
    Computes the acceleration due to J2 perturbation

    Inputs:
    r - position vector in ECI frame [m]

    Outputs:
    a_gravity_J2 - acceleration due to J2 perturbation [m/s^2]
    '''
    x = r[0]
    y = r[1]
    z = r[2]
    J2 = 0.00108248
    Re = 6378.1363
```

```
    mu = 398600.4415
        # The equation for calculating dUdx
    dUdx = -(2*mu*x*(x**2 + y**2 + z**2)**2 - 15*J2*Re**2*mu*x*
       z**2 +
            3*J2*Re**2*mu*x*(x**2 + y**2 + z**2))/(2*(x**2 + y
               **2 + z**2)**(7/2))

    # The equation for calculating dUdy
    dUdy = -(2*mu*y*(x**2 + y**2 + z**2)**2 - 15*J2*Re**2*mu*y*
       z**2 +
            3*J2*Re**2*mu*y*(x**2 + y**2 + z**2))/(2*(x**2 + y
               **2 + z**2)**(7/2))

    # The equation for calculating dUdz
    dUdz = -(2*mu*z*(x**2 + y**2 + z**2)**2 - 15*J2*Re**2*mu*z
       **3 +
            9*J2*Re**2*mu*z*(x**2 + y**2 + z**2))/(2*(x**2 + y
               **2 + z**2)**(7/2))

    a_gravity_J2 = np.array([dUdx, dUdy, dUdz])

    return a_gravity_J2

def load_egm96_coefficients():
    '''
    Loads the EGM96 coefficients from the CSV files

    Inputs:
    None

    Outputs:
    C - EGM96 C coefficients
    S - EGM96 S coefficients
    '''

    EGM96_C_file = 'EGM96_C.csv'
    EGM96_S_file = 'EGM96_S.csv'

    # Load EGM96 coefficients from file
    C = []
    S = []
    # Read in the CSV file and populate the matrix
    with open(EGM96_C_file, 'r') as file:
        csv_reader = csv.reader(file)
        for row in csv_reader:
```

```
            C.append(row)
    # S = np.loadtxt(EGM96_S_file)
    with open(EGM96_S_file, 'r') as file:
        csv_reader = csv.reader(file)
        for row in csv_reader:
            S.append(row)
    C = np.array(C).astype(float)
    S = np.array(S).astype(float)
    return C, S

C, S = load_egm96_coefficients()

@jit
def grav_odp(r):

    degree = 20
    C_deg = C[0:degree, 0:degree]
    S_deg = S[0:degree, 0:degree]
    # print(C_deg.shape)

    x, y, z = r
    mu = 398600.4415 #km^3/s^2
    RE = 6378.1363 #km
    nmaxp1, mmaxp1 = C_deg.shape

    nmax = nmaxp1 - 1
    mmax = mmaxp1 - 1
    Anm = np.zeros(nmaxp1+1)
    Anm1 = np.zeros(nmaxp1+1)
    Anm2 = np.zeros(nmaxp1+2)
    R = np.zeros(nmaxp1+1)
    I = np.zeros(nmaxp1+1)
    rb2 = x*x + y*y + z*z
    rb = np.sqrt(rb2)
    mur2 = mu/rb2
    mur3 = mur2/rb

    # direction of spacecraft position
    s = x/rb
    t = y/rb
    u = z/rb

    # Calculate contribution of only Zonals
    Anm1[0] = 0
    Anm1[1] = np.sqrt(3)
```

```
Anm2[1] = 0
Anm2[2] = np.sqrt(3.75)

as_ = 0
at = 0
au = 0
ar = 0
rat1 = 0
rat2 = 0
Dnm = 0
Enm = 0
Fnm = 0
Apor = np.zeros(nmaxp1)
# print(Apor.shape)
Apor[0] = 1
Apor[1] = RE/rb

for n in range(1, nmax +1):
    i = n+1
    an2 = 2*n
    rat1 = np.sqrt((an2+3.0)*(((an2+1.0)/n)/(n+2.0)))
    rat2 = np.sqrt((n+1.0)*(((n-1.0)/(an2-1.0))/(an2+1.0)))
    Anm1[i] = rat1*(u*Anm1[i-1] - rat2*Anm1[i-2])
    Apor[i-1] = Apor[i-2]*Apor[1]
    if n < mmaxp1:
        rat1 = np.sqrt((an2+5.0)*(((an2+3.0)/n)/(n+4.0)))
        rat2 = np.sqrt((n+3.0)*(((n-1.0)/(an2+1.0))/(an2
           +3.0)))
        Anm2[i+1] = rat1*(u*Anm2[i] - rat2*Anm2[i-1])
    if n < nmaxp1:
        rat1 = np.sqrt(0.5*n*(n+1.0))
        au = au - Apor[i-1]*rat1*Anm1[i-1]*(-C_deg[i-1, 0])
        rat2 = np.sqrt(0.5*((an2+1.0)/(an2+3.0))*(n+1.0)*(n
           +2.0))
        ar = ar + Apor[i-1]*rat2*Anm1[i]*(-C_deg[i-1, 0])

# Calculate contribution of Tesserals
R = np.array([1], dtype=np.float64) # Initialize R as a
   numpy array with one element = 1
I = np.array([0], dtype=np.float64) # Initialize I as a
   numpy array with one element = 0

for m in range(1, mmax + 1): # Loop from 1 to mmax (
   inclusive)
```

```
        j = m
        am2 = 2*m
        R = np.append(R, s*R[j-1] - t*I[j-1])
        I = np.append(I, s*I[j-1] + t*R[j-1])

        for l in range(m, mmax + 1): # Loop from the current
           value of m to mmax (inclusive)
            i = l
            Anm[i] = Anm1[i]
            Anm1[i] = Anm2[i]

        Anm1[mmax+1] = Anm2[mmax+1]

        for l in range(m, mmax + 1):
            i = l
            an2 = 2*l
            if l == m:
                Anm2[j+1] = 0.0
                Anm2[j+2] = np.sqrt((am2+5.0)/(am2+4.0))*Anm1[j
                   +1]
            else:
                rat1 = np.sqrt((an2+5.0)*(((an2+3.0)/(l-m))/(l+
                   m+4.0)))
                rat2 = np.sqrt((l+m+3.0)*(((l-m-1.0)/(an2+1.0))
                   /(an2+3.0)))
                Anm2[i+2] = rat1*(u*Anm2[i+1] - rat2*Anm2[i])

            Dnm = C_deg[i,j]*R[j] + S_deg[i,j]*I[j]
            Enm = C_deg[i,j]*R[j-1] + S_deg[i,j]*I[j-1]
            Fnm = S_deg[i,j]*R[j-1] - C_deg[i,j]*I[j-1]

            rat1 = np.sqrt((l+m+1.0)*(l-m))
            rat2 = np.sqrt(((an2+1.0)/(an2+3.0))*(l+m+1.0)*(l+m
               +2.0))

            as_ += Apor[i]*m*Anm[i]*Enm
            at += Apor[i]*m*Anm[i]*Fnm
            au += Apor[i]*rat1*Anm1[i]*Dnm
            ar -= Apor[i]*rat2*Anm1[i+1]*Dnm

    agx_ECEF = -mur3*x + mur2*(as_ + ar*s)
    agy_ECEF = -mur3*y + mur2*(at + ar*t)
    agz_ECEF = -mur3*z + mur2*(au + ar*u)
```

```
    return np.array([agx_ECEF, agy_ECEF, agz_ECEF])


def range_range_rate(r, v, station_ECI, station_dot_ECI):
    '''
    Compute range and range rate from a station to a satellite

    Inputs:
        r - position vector of satellite in ECI frame [m]
        v - velocity vector of satellite in ECI frame [m/s]
        station_ECI - position vector of station in ECI frame [
           m]
        station_dot_ECI - velocity vector of station in ECI
           frame [m/s]

    Outputs:
        rho - range from station to satellite [m]
        rho_dot - range rate from station to satellite [m/s]
    '''

    rho = np.linalg.norm(r - station_ECI)
    rho_dot = ((r - station_ECI) @ (v - station_dot_ECI))/rho

    return rho, rho_dot

def EKF(case, r_ECI, v_ECI, stations, JD_UTC, seconds, Ps, Psv,
    Qs, a_bias):
    '''
    Extended Kalman Filter for orbit determination

    Inputs:
        case - case number
        r_ECI - initial position vector of satellite in ECI
           frame [m]
        v_ECI - initial velocity vector of satellite in ECI
           frame [m/s]
        stations - matrix of station locations
        JD_UTC - Julian date in UTC
        seconds - seconds of propagation
        Ps - initial covariance of position
        Psv - initial covariance velocity
        Qs - process noise sigma

    Outputs:
        y_arr - array of state vectors
```

```
    P_hat - array of covariance matrices
    P_ZZ - array of innovation covariance matrices
    obs_mat - modified data of observations
    Q - process noise matrix

'''
print('Case: ', case)
obs_mat_3d = loadmat('LEO_DATA_Apparent_3Days.mat')
obs_mat_4_6 = loadmat('LEO_DATA_Apparent_Days4-6.mat')
obs_mat_4_6['LEO_DATA_Apparent'][:,1] += 3*86400
obs_mat = np.concatenate((obs_mat_3d['LEO_DATA_Apparent'],
   obs_mat_4_6['LEO_DATA_Apparent']), axis=0)

if case == 'C':
    obs_mat = obs_mat[obs_mat[:, 0] == 1]


elif case == 'D':
    obs_mat = obs_mat[obs_mat[:, 0] == 2]
    # obs_mat = obs_mat[obs_mat[:, 0] != 3]

elif case == 'E':
    obs_mat = obs_mat[obs_mat[:, 0] == 3]

elif case == 'G':
    obs_mat = obs_mat[obs_mat[:, 1] >= 5*86400]



#Initialize Extended Kalman Filter


y0 = np.concatenate((r_ECI, v_ECI))

#initial covariance


sigma_x = Ps
sigma_y = Ps
sigma_z = Ps
sigma_vx = Psv
sigma_vy = Psv
sigma_vz = Psv
```

```
#initial covariance matrix
P_hat_0 = np.diag([sigma_x**2, sigma_y**2, sigma_z**2,
   sigma_vx**2, sigma_vy**2, sigma_vz**2])

#process noise
sigma_x = Qs
sigma_y = Qs
sigma_z = Qs

# sigma_x = 0
# sigma_y = 0
# sigma_z = 0

del_t = sym.Symbol('del_t')
Q_sym = del_t**2*sym.Matrix([[del_t**2/4*sigma_x**2, 0, 0,
   del_t/2*sigma_x**2, 0, 0],
                        [0, del_t**2/4*sigma_y**2, 0, 0,
                           del_t/2*sigma_y**2, 0],
                        [0, 0, del_t**2/4*sigma_z**2, 0, 0,
                            del_t/2*sigma_z**2],
                        [del_t/2*sigma_x**2, 0, 0, sigma_x
                           **2, 0, 0],
                        [0, del_t/2*sigma_y**2, 0, 0,
                           sigma_y**2, 0],
                        [0, 0, del_t/2*sigma_z**2, 0, 0,
                           sigma_z**2]])

Q = sym.lambdify(del_t, Q_sym, 'numpy')

#measurement noise
sigma_rho_kwaj = 10E-3 #km
sigma_rho_dot_kwaj = 0.5E-6 #km/s
sigma_rho_dg = 5E-3 #km
sigma_rho_dot_dg = 1E-6 #km/s
sigma_rho_arecibo = 10E-3 #km
sigma_rho_dot_arecibo = 0.5E-6 #km/s
R = np.array([np.diag([sigma_rho_kwaj**2,
   sigma_rho_dot_kwaj**2]),
             np.diag([sigma_rho_dg**2, sigma_rho_dot_dg**2])
                ,
             np.diag([sigma_rho_arecibo**2,
                sigma_rho_dot_arecibo**2])])
```

```
#measurement bias
# bias_kwaj = 40 #m
# bias_dg = -30 #m
# bias_arecibo = 37#m
bias_kwaj = 0
bias_dg = 0
# bias_arecibo = 21.5E-3 #km
bias_arecibo = a_bias
bias = np.array([bias_kwaj, bias_dg, bias_arecibo])

#time
obs_time = obs_mat[:, 1]
obs_id = np.array(obs_mat[:, 0], dtype=int)
obs_range = obs_mat[:, 2]
obs_range_rate = obs_mat[:, 3]

m = obs_time[np.where(obs_time < seconds)].shape[0]
t = obs_time[:m]

obs_mat = obs_mat[:m]


#jacobians
F = A_Matrix()
H = H_tilde_matrix()

#propogation function
f = satellite_motion_phi
pre_fit_res = np.zeros((len(t), 2))
post_fit_res = np.zeros((len(t), 2))

#initializing arrays
y_arr = np.zeros((len(t), 6))
P_bar = np.zeros((len(t), 6*6))
P_hat = np.zeros((len(t), 6*6))
P_ZZ = np.zeros((len(t), 2*2))
phi = np.zeros((len(t), 6*6))

y_arr[0] = y0
P_bar[0] = np.eye(6).reshape(6*6)
P_hat[0] = np.reshape(P_hat_0, 6*6)
phi[0] = np.eye(6).reshape(6*6)

#compute first observation
y_pred = np.concatenate((r_ECI, v_ECI))
```

```
del_t = 0
Q_i = Q(del_t)
phi_0 = np.eye(6)

#propogate covariance
P_hat_k = P_hat[0].reshape(6,6)
# print('P_hat_k', P_hat_k)
P_bar_k_1 = phi_0 @ P_hat_k @ phi_0.T + Q_i

P_bar[0] = np.reshape(P_bar_k_1, 6*6)

#if stations don't have the first observation propagate to
   their first observation
if case == 'C' or case == 'E' or case == 'G':

    del_t = t[0]
    t_eval = np.arange(0, t[0]+60, 60)
    y0 = np.concatenate((y_pred[0:6], phi_0.ravel()))
    prop = solve_ivp(f, [0, del_t], y0, args=(F, JD_UTC + t
       [0]/86400), t_eval=t_eval, rtol=3e-14, atol=1e-16)
    y_pred = prop.y[0:6, -1]
    phi_0 = prop.y[6:, -1].reshape(6, 6)
    phi[0] = phi_0.reshape(6*6)

    #process noise
    Q_i = Q(del_t)

    #recalculate P_bar
    P_bar_k_1 = np.matmul(np.matmul(phi_0, P_hat[0].reshape
       (6,6)), phi_0.T) + Q_i

    P_bar[0] = np.reshape(P_bar_k_1, 6*6)

#find station ECI coordinates
station = stations[obs_id[0]-1]

day = 0

x_p, y_p, del_UT1, LOD = interp_EOP(JD_UTC + t[0]/86400)

ECI_station, ECI_station_dot, _= ECEF2ECI(station, np.array
   ([0,0,0]), None, JD_UTC + t[0]/86400, x_p, y_p, leap_sec
   , del_UT1, LOD)

#light time correction
```

```
y_lt = light_time_correction(JD_UTC + t[0]/86400, y_pred
   [0:3], y_pred[3:6], ECI_station, station)

#compute H_k
H_k = np.array(H(y_pred[0], y_pred[1], y_pred[2], y_pred
   [3], y_pred[4], y_pred[5],
                 ECI_station[0], ECI_station[1], ECI_station
                    [2], ECI_station_dot[0], ECI_station_dot
                    [1], ECI_station_dot[2], bias[obs_id
                    [0]-1]))

#compute observation
obs_pred = np.array(range_range_rate(y_lt[0:3], y_lt[3:6],
   ECI_station, ECI_station_dot))

#add bias and compute residuals
if case == 'A ':
    pre_fit_res[0] = np.array([obs_range[0] - (obs_pred[0]
       + bias[obs_id[0]-1]) , 0])
    H_k = np.array([[1, 0], [0, 0]]) @ H_k



elif case == 'B':
    pre_fit_res[0] = np.array([0, obs_range_rate[0] -
       obs_pred[1]])
    H_k = np.array([[0, 0], [0, 1]]) @ H_k

else:
    pre_fit_res[0] = np.array([obs_range[0] - (obs_pred[0]
       + bias[obs_id[0]-1]), obs_range_rate[0] - obs_pred
       [1]])
    # print('bias',  bias.T[obs_id[0]-1])

print('pre_fit_res', pre_fit_res[0])


#compute kalman gain
P_ZZ[0] = (H_k @ P_bar_k_1 @ H_k.T + R[obs_id[0]-1]).
   reshape(2*2)

K = P_bar_k_1 @ H_k.T @ np.linalg.inv(P_ZZ[0].reshape(2,2))

#state error estimate
del_y = K @ pre_fit_res[0]
```

```
#error covariance estimate
P_hat[0] = np.reshape((np.eye(6) - K @ H_k) @ P_bar_k_1 @ (
   np.eye(6) - K @ H_k).T + K @ R[obs_id[0]-1] @ K.T, 6*6)

#state estimate
y_arr[0] = y_pred + del_y

#correct for light time
y_lt = light_time_correction(JD_UTC + t[0]/86400, y_arr
   [0][0:3], y_arr[0][3:6], ECI_station, station)
#compute rho and rho_dot
rho, rho_dot = range_range_rate(y_lt[0:3], y_lt[3:6],
   ECI_station, ECI_station_dot)
# print('rho', rho)
#add bias and compute residuals
post_fit_res[0] = np.array([obs_range[0] - (rho +  bias[
   obs_id[0]-1]) , obs_range_rate[0] - rho_dot])
# print('post_fit_res', post_fit_res[0])


day_old = 1
old_obs_id = obs_id[0]
print('Running...')
#run EKF
for k in range(1, len(t)):
    del_t = t[k] - t[k-1]
    # print('t[k-1]', t[k-1])
    # print('t[k]', t[k])

    day = int(((JD_UTC + t[k]/86400) - JD_UTC_day))
    new_obs_id = obs_id[k]
    if new_obs_id != old_obs_id:
        # print('station: ', new_obs_id)
        old_obs_id = new_obs_id
    if day != day_old:
        print('Day: ', day)
        day_old = day
    phi_k = np.eye(6)

    #initial conditions for propogation
    y0 = np.concatenate((y_arr[k-1][0:6], phi_k.ravel()))
    t_eval = np.arange(0, del_t+60, 60)

    #propogate state
```

```
prop = solve_ivp(f, [0, del_t], y0, args=(F, JD_UTC + t
   [k-1]/86400), t_eval=t_eval, rtol=3e-14, atol=1e-16)

y_pred = prop.y[0:6, -1]

phi_k = prop.y[6:, -1].reshape(6, 6)

phi[k] = phi_k.reshape(6*6)

#process noise
Q_i = Q(del_t)

#propogate covariance
P_hat_k = P_hat[k-1].reshape(6,6)

P_bar_k_1 = phi_k @ P_hat_k @ phi_k.T + Q_i
P_bar[k] = np.reshape(P_bar_k_1, 6*6)

#calculate station ECI coordinates
station = stations[obs_id[k]-1]

x_p, y_p, del_UT1, LOD = interp_EOP(JD_UTC + t[k
   ]/86400)
ECI_station, ECI_station_dot, _= ECEF2ECI(station, np.
   array([0,0,0]), None, JD_UTC + t[k]/86400, x_p, y_p,
    leap_sec, del_UT1, LOD)

#light time correction
y_lt = light_time_correction(JD_UTC + t[k]/86400,
   y_pred[0:3], y_pred[3:6], ECI_station, station)
# print('light_time difference', np.linalg.norm(y_lt
   [0:3] - y_pred[0:3]))

#compute H_k
H_k = np.array(H(y_pred[0], y_pred[1], y_pred[2],
   y_pred[3], y_pred[4], y_pred[5], ECI_station[0],
   ECI_station[1],
                ECI_station[2], ECI_station_dot[0],
                   ECI_station_dot[1], ECI_station_dot
                   [2], bias[obs_id[k]-1]))

#compute observation
obs_pred = np.array(range_range_rate(y_lt[0:3], y_lt
   [3:6], ECI_station, ECI_station_dot))
```

```
#add bias and compute residuals
if case == 'A':
    pre_fit_res[k] = np.array([obs_range[k] - (obs_pred
       [0] + bias[obs_id[k]-1]) , 0])
    H_k = np.array([[1, 0], [0, 0]]) @ H_k

elif case == 'B':
    pre_fit_res[k] = np.array([0, obs_range_rate[k] -
       obs_pred[1]])
    H_k = np.array([[0, 0], [0, 1]]) @ H_k

else:
    pre_fit_res[k] = np.array([obs_range[k] - (obs_pred
       [0] +  bias[obs_id[k]-1]) , obs_range_rate[k] -
       obs_pred[1]])

# print('bias', bias[obs_id[k]-1])
print('pre_fit_res', pre_fit_res[k])

#compute kalman gain
K = P_bar_k_1 @ H_k.T @ np.linalg.inv(H_k @ P_bar_k_1 @
    H_k.T + R[obs_id[k]-1])

#innovation covariance
P_ZZ[k] = (H_k @ P_bar_k_1 @ H_k.T + R[obs_id[k]-1]).
   reshape(2*2)

#state error estimate
del_y = K @ pre_fit_res[k]

#error covariance estimate
P_hat[k] = np.reshape((np.eye(6) - K @ H_k) @ P_bar_k_1
    @ (np.eye(6) - K @ H_k).T + K @ R[obs_id[k]-1] @ K.
   T, 6*6)

#update state estimate
y_arr[k] = y_pred + del_y
y_hat_lt = light_time_correction(JD_UTC + t[k]/86400,
   y_arr[k][0:3], y_arr[k][3:6], ECI_station, station)
rho, rho_dot = range_range_rate(y_hat_lt[0:3], y_hat_lt
   [3:6], ECI_station, ECI_station_dot)
```

```
        post_fit_res[k] = np.array([obs_range[k] - (rho + bias[
            obs_id[k]-1]), obs_range_rate[k] - rho_dot])
        # print('post_fit_res',post_fit_res[k])


    print('EKF complete')
    return y_arr, P_hat, P_bar, P_ZZ, obs_mat, Q, pre_fit_res,
        post_fit_res, phi



def plot_ellipses(x_mean, covariance, sig, ax, case, color):
    # Mean and Covariance
    P_xx = covariance

    eigenval, eigenvec = np.linalg.eig(P_xx)

    # Get the index of the largest eigenvector
    max_evc_ind_c = np.argmax(eigenval)
    max_evc = eigenvec[:, max_evc_ind_c]

    # Get the largest eigenvalue
    max_evl = np.max(eigenval)
    min_evl = np.max(eigenval)


    # Calculate the angle between the x-axis and the largest
        eigenvector
    phi = np.arctan2(max_evc[1], max_evc[0])


    theta_grid = np.linspace(0, 2*np.pi, 100)

    X0  = x_mean[0]
    Y0  = x_mean[1]

    # Find semi-major and semi-minor axis for 1,2,3sigma

    a = np.sqrt(sig*max_evl)      # Semi-major axis
    b = np.sqrt(sig*min_evl)      # Semi- minor axis


    # the ellipse in x and y coordinates
    ellipse_x_r  = a*np.cos(theta_grid)
    ellipse_y_r  = b*np.sin(theta_grid)
```

```
    # Define a rotation matrix (x',y') to (x,y)
    R = np.array([[np.cos(phi), -np.sin(phi)],
                  [np.sin(phi), np.cos(phi)]])

    # let's rotate the ellipse to some angle phi
    r_1 = np.zeros((len(theta_grid), 2))

    for j in range(len(theta_grid)):
        r_1[j,:] = R @ np.array([ellipse_x_r[j],ellipse_y_r[j
           ]])


    X1 = r_1.T.reshape(2,len(theta_grid))

    ax.plot(X0 + X1[0,:], Y0 + X1[1,:], linestyle='--',
       linewidth=1, label='Case ' + case, color=color)


    return X1


def eci_to_ric(r_ref, v_ref, r, cov):

    '''
    Transforms a state vector from ECI to RIC frame
    Inputs:
        r_ref: reference position vector
        v_ref: reference velocity vector
        r: position vector to be transformed
        cov: covariance matrix of position vector to be
           transformed

    Outputs:
        r_ric: position vector in RIC frame
        cov_ric: covariance matrix of position vector in RIC
           frame
    '''


   # Calculate the rotation matrix gamma
    u_R = r_ref / np.linalg.norm(r_ref) # Calculate unit vector
       in radial direction
    u_N = np.cross(r_ref, v_ref) / np.linalg.norm(np.cross(
       r_ref, v_ref)) # Calculate unit vector in normal
```

```
        direction
    u_T = np.cross(u_N, u_R) # Calculate unit vector in
        tangential direction

    gamma = np.vstack((u_R, u_T, u_N)) # Create rotation matrix
        from unit vectors

    # Transform position and covariance matrix to radial-
        inertial frame
    r_ric = gamma @ (r_ref - r) # Transform position vector
    cov_ric = gamma @ cov @ gamma.T # Transform covariance
        matrix

    return r_ric, cov_ric # Return transformed position and
        covariance matrix

def plot_residuals(case, obs_mat, residuals, res_type, t, P_ZZ,
    Ps, Psv, Qs):
    '''
    Saves the plot of the residuals for each station

    Inputs:
        case: string of the case letter
        obs_mat: data of the observations
        column_names: list of the column names for the
            observations
        t: time array
        P_ZZ: covariance matrix
        m: number of observations


    '''
    #postfit RMS
    # df_calc = pd.read_csv('range_rate_case' + case + '.csv',
        names=column_names)
    #Comparing Pre-fit residuals
    fig, ax = plt.subplots(2, 1, figsize=(10, 10))

    if case == 'A' or case == 'B' or case == 'C' or case == 'F'
        or case == 'G':

        r_residuals_kwaj = residuals[np.where(obs_mat[:, 0] ==
            1)[0], 0]
        rr_residuals_kwaj = residuals[np.where(obs_mat[:, 0] ==
            1)[0], 1]
```

```
        RMS_Kwaj_r = np.sqrt(np.mean((r_residuals_kwaj)**2))
        RMS_Kwaj_rr = np.sqrt(np.mean((rr_residuals_kwaj)**2))


        t_kwaj = obs_mat[np.where(obs_mat[:, 0] == 1)[0], 1]

        print("Kwaj Range RMS:", RMS_Kwaj_r, "[km]")
        print("Kwaj Range Rate RMS:", RMS_Kwaj_rr,"[km/s]")

        ax[0].scatter(t_kwaj, r_residuals_kwaj, label='Kwaj
           Range Residuals')
        ax[1].scatter(t_kwaj, rr_residuals_kwaj, label='Kwaj
           Range Rate Residuals')

    if case == 'A' or case == 'B' or case == 'D' or case == 'F'
        or case == 'G':


        r_residuals_dg = residuals[np.where(obs_mat[:, 0] == 2)
           [0], 0]
        rr_residuals_dg = residuals[np.where(obs_mat[:, 0] ==
           2)[0], 1]

        RMS_Diego_r = np.sqrt(np.mean((r_residuals_dg)**2))
        RMS_Diego_rr = np.sqrt(np.mean((rr_residuals_dg)**2))

        t_diego = obs_mat[np.where(obs_mat[:, 0] == 2)[0], 1]

        print("Diego Range RMS:", RMS_Diego_r, "[km]")
        print("Diego Range Rate RMS:", RMS_Diego_rr,"[km/s]")

        ax[0].scatter(t_diego, r_residuals_dg, label='Diego
           Range Residuals')
        ax[1].scatter(t_diego, rr_residuals_dg, label='Diego
           Range Rate Residuals')


    if case == 'A' or case == 'B' or case == 'E' or case == 'F'
        or case == 'G':


        r_residuals_arecibo = residuals[np.where(obs_mat[:, 0]
           == 3)[0], 0]
```

```
        rr_residuals_arecibo = residuals[np.where(obs_mat[:, 0]
            == 3)[0], 1]

        RMS_Arecibo_r = np.sqrt(np.mean((r_residuals_arecibo)
           **2))
        RMS_Arecibo_rr = np.sqrt(np.mean((rr_residuals_arecibo)
           **2))

        t_arecibo = obs_mat[np.where(obs_mat[:, 0] == 3)[0], 1]

        print("Arecibo Range RMS:", RMS_Arecibo_r, "[km]")
        print("Arecibo Range Rate RMS:", RMS_Arecibo_rr,"[km/s
           ]")

        ax[0].scatter(t_arecibo, r_residuals_arecibo, label='
           Arecibo Range Residuals')
        ax[1].scatter(t_arecibo, rr_residuals_arecibo, label='
           Arecibo Range Rate Residuals')


three_sigma_pos = np.array([np.sqrt(P_ZZ[i][0])*3 for i in
   range(len(t))])
three_sigma_vel = np.array([np.sqrt(P_ZZ[i][3])*3 for i in
   range(len(t))])

# print(three_sigma_pos)
ax[0].plot(t, three_sigma_pos, label='+3$\sigma$')
ax[0].plot(t, -three_sigma_pos, label='-3$\sigma$')
ax[1].plot(t, three_sigma_vel, label='+3$\sigma$')
ax[1].plot(t, -three_sigma_vel, label='-3$\sigma$')



ax[0].set_title('Range Residuals')
ax[0].set_xlabel('Time [s]')
ax[0].set_ylabel('Range Residuals [km]')
ax[0].set_ylim(-0.05, 0.05)
ax[0].legend()


ax[1].set_title('Range Rate Residuals')
ax[1].set_xlabel('Time [s]')
ax[1].set_ylabel('Range Rate Residuals [km/s]')
ax[1].set_ylim(-0.00002, 0.00002)
ax[1].legend()
```

```
    # plt.show()
    fig.savefig(res_type + 'residuals_case' + case+ str(Qs) +
       str(Ps) + str(Psv) +'.png', dpi=300)


    import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint, solve_ivp
from scipy.special import lpmn
from scipy.io import savemat, loadmat
import pandas as pd
from datetime import datetime
import sympy as sym
import csv
from numba import jit

leap_sec = 37 #s
x_ps = np.array([20.816, 22.156, 23.439, 24.368, 25.676,
   26.952, 28.108, 29.400, 31.034])/1000 #arcsec
y_ps = np.array([381.008, 382.613, 384.264, 385.509, 386.420,
   387.394, 388.997, 390.776, 392.669])/1000 #arcsec
del_UT1s = np.array([144.0585, 143.1048, 142.2335, 141.3570,
   140.4078, 139.3324, 138.1510, 136.8455, 135.4165])/1000 #s
LODs = np.array([1.0293,0.9174, 0.8401, 0.8810, 1.0141, 1.1555,
    1.2568, 1.3360, 1.3925])/1000 #s


def gregorian_to_jd(year, month, day, hour, minute, second):
    '''Convert Gregorian calendar date to Julian date time.
    Input
        year : int
        month : int
        day : int
        hour : int
        minute : int
        second : float
    Returns
        jd : float '''

    a = int((14 - month)/12)
    y = year + 4800 - a
    m = month + 12*a - 3
    jd = day + int((153*m + 2)/5) + 365*y + int(y/4) - int(y
       /100) + int(y/400) - 32045
    jd = jd + (hour - 12)/24 + minute/1440 + second/86400
```

```
    return jd

JD_UTC_st = gregorian_to_jd(2018, 3, 23, 8, 55, 3)
JD_UTC_day = gregorian_to_jd(2018, 3, 23, 0, 0, 0)
JD_days = np.arange(JD_UTC_day, JD_UTC_day+9, 1)

def interp_EOP(JD_UTC):
    '''Linear interpolation of Earth Orientation Parameters (
       EOP) data

    Inputs:
    JD_UTC: time in Julian Centuries since J2000

    Outputs:
    interp_x_p: interpolated x polar motion in arcseconds
    interp_y_p: interpolated y polar motion in arcseconds
    interp_del_UT1: interpolated UT1-UTC in seconds
    interp_LOD: interpolated length of day in seconds
    '''

    interp_x_p = np.interp(JD_UTC, JD_days, x_ps)
    interp_y_p = np.interp(JD_UTC, JD_days, y_ps)
    interp_del_UT1 = np.interp(JD_UTC, JD_days, del_UT1s)
    interp_LOD = np.interp(JD_UTC, JD_days, LODs)


    return interp_x_p, interp_y_p, interp_del_UT1, interp_LOD


def orthodcm(DCM):
    '''Orthogonalize a direction cosine matrix (DCM) using the
       Rodriguez formula
    Inputs:
    DCM: 3x3 direction cosine matrix
    Outputs:
    orthoDCM: 3x3 orthogonalized direction cosine matrix
    '''
    delT = DCM @ DCM.T - np.eye(3)
    orthoDCM = DCM @ (np.eye(3) - (1/2)*delT + (1/2)*(3/4)*delT
       **2 - (1/2)*(3/4)*(5/6)*delT**3 + (1/2)*(3/4)*(5/6)
       *(7/8)*delT**4 -
                     (1/2)*(3/4)*(5/6)*(7/8)*(9/10)*delT**5 +
                        (1/2)*(3/4)*(5/6)*(7/8)*(9/10)*(11/12)*
                        delT**6 -
```

```
                        (1/2)*(3/4)*(5/6)*(7/8)*(9/10)*(11/12)
                           *(13/14)*delT**7 + (1/2)*(3/4)*(5/6)
                           *(7/8)*(9/10)*(11/12)*(13/14)*(15/16)*
                           delT**8 -
                        (1/2)*(3/4)*(5/6)*(7/8)*(9/10)*(11/12)
                           *(13/14)*(15/16)*(17/18)*delT**9 +
                           (1/2)*(3/4)*(5/6)*(7/8)*(9/10)*(11/12)
                           *(13/14)*(15/16)*(17/18)*(19/20)*delT
                           **10)
    return orthoDCM


#Rotation Martices

def Rot1(theta: float) -> np.ndarray:
    return np.array([[1, 0, 0], \
                     [0, np.cos(theta), np.sin(theta)], \
                     [0, -np.sin(theta), np.cos(theta)]])

def Rot2(theta: float) -> np.ndarray:
    return np.array([[np.cos(theta), 0, -np.sin(theta)],\
                    [0, 1, 0], \
                    [np.sin(theta), 0, np.cos(theta)]])

def Rot3(theta: float) -> np.ndarray:
    return np.array([[np.cos(theta), np.sin(theta), 0], \
                     [-np.sin(theta), np.cos(theta), 0], \
                    [0, 0, 1]])

#read in nutation data file
def read_nut80():
    # IAU1980 Theory of Nutation model
    dat_file = "nut80.dat"

    #nutaton model column names
    column_names = ['ki1', 'ki2', 'ki3', 'ki4', 'ki5', 'Aj', '
       Bj', 'Cj', 'Dj', 'j']

    #nutation dataframe
    df_nut80 = pd.read_csv(dat_file, sep="\s+", names=
       column_names)
    return df_nut80

df_nut80 = read_nut80()
Aj = df_nut80['Aj'].values
```

```
Bj = df_nut80['Bj'].values
Cj = df_nut80['Cj'].values
Dj = df_nut80['Dj'].values
k = df_nut80[df_nut80.columns[0:5]].values




def PrecessionMatrix(T_TT):
    '''
    Calculates the precession matrix

    Inputs:
    T_TT: Julian Centuries since J2000

    returns: precession matrix
    '''

    arc_sec_to_rad = np.pi/(180*3600)

     #precession angles
    C_a = (2306.2181*T_TT + 0.30188*T_TT**2 + 0.017998*T_TT**3)
       *arc_sec_to_rad
    theta_a = (2004.3109*T_TT - 0.42665*T_TT**2 - 0.041833*T_TT
       **3)*arc_sec_to_rad
    z_a = (2306.2181*T_TT + 1.09468*T_TT**2 + 0.018203*T_TT**3)
       *arc_sec_to_rad

    #precession matrix
    P = Rot3(C_a) @Rot2(-theta_a) @ Rot3(z_a)

    return P

def PolarMotionMatrix(x_p, y_p):
    '''
    Calculates the polar motion matrix

    Inputs:
    x_p: x polar motion in arcseconds
    y_p: y polar motion in arcseconds

    returns: polar motion matrix
    '''
```

```
    arc_sec_to_rad = np.pi/(180*3600)

    #radians conversions
    x_p = x_p*arc_sec_to_rad
    y_p = y_p*arc_sec_to_rad

    # Polar Motion Matrix
    W = Rot1(y_p) @ Rot2(x_p)

    return W

def SiderealTimeMatrix(T_UT1, T_TT):
    '''
    Calculates the sidereal time matrix

    Inputs:
    T_UT1: Julian Centuries since J2000
    T_TT: Julian Centuries since J2000

    returns: sidereal time matrix
    '''
    w = 7.2921151467E-5*180/np.pi #earth rotation rate in deg/s
    deg2rad = np.pi/180
    arc_sec_to_rad = np.pi/(180*3600)

    #Greenwich Mean Sidereal Time
    GMST = 67310.54841 + (876600*3600 + 8640184.812866)*T_UT1 +
        0.093104*T_UT1**2 - 6.2E-6*T_UT1**3

    #convert GMST to radians
    GMST = (GMST%86400)/240*deg2rad

    # print('GMST: ', GMST*180/np.pi)

    #anamolies
    r = 360
    Mmoon = (134.96298139 + (1325*r + 198.8673981)*T_TT +
       0.0086972*T_TT**2 + 1.78E-5*T_TT**3)
    Mdot = (357.52772333 + (99*r + 359.0503400)*T_TT -
       0.0001603*T_TT**2 - 3.3E-6*T_TT**3)
    uMoon = (93.27191028 + (1342*r + 82.0175381)*T_TT -
       0.0036825*T_TT**2 + 3.1E-6*T_TT**3)
    Ddot = (297.85036306 + (1236*r + 307.1114800)*T_TT -
       0.0019142*T_TT**2 + 5.3E-6*T_TT**3)
```

```
    lamMoon = (125.04452222 - (5*r + 134.1362608)*T_TT +
       0.0020708*T_TT**2 + 2.2E-6*T_TT**3)
    alpha = np.array([Mmoon, Mdot, uMoon, Ddot, lamMoon])*
       deg2rad

    #nutation in lam
    del_psi = np.dot((Aj*10**-4 + Bj*10**-4*T_TT)*
       arc_sec_to_rad, np.sin(np.dot(k, alpha)))

    #mean obliquity of the ecliptic
    epsilon_m = 84381.448 - 46.8150*T_TT - 0.00059*T_TT**2 +
       0.001813*T_TT**3

    #conversion to radians
    epsilon_m = epsilon_m*arc_sec_to_rad

    #equation of the equinoxes
    Eq_eq = del_psi*np.cos(epsilon_m) + 0.000063*arc_sec_to_rad
       *np.sin(2*alpha[4]) + 0.00264*arc_sec_to_rad*np.sin(
       alpha[4])

    #greenwich apparent sidereal time
    GAST = GMST + Eq_eq


    #sidereal rotation matrix
    R_mat = Rot3(-GAST)

    return R_mat


def NutationMatrix(T_TT):
    '''
    Calculates the nutation matrix

    Inputs:
    T_TT: Julian Centuries since J2000

    returns: nutation matrix
    '''

    deg2rad = np.pi/180
    arc_sec_to_rad = np.pi/(180*3600)

    #anamolies
```

```
    r = 360
    Mmoon = (134.96298139 + (1325*r + 198.8673981)*T_TT +
       0.0086972*T_TT**2 + 1.78E-5*T_TT**3)
    Mdot = (357.52772333 + (99*r + 359.0503400)*T_TT -
       0.0001603*T_TT**2 - 3.3E-6*T_TT**3)
    uMoon = (93.27191028 + (1342*r + 82.0175381)*T_TT -
       0.0036825*T_TT**2 + 3.1E-6*T_TT**3)
    Ddot = (297.85036306 + (1236*r + 307.1114800)*T_TT -
       0.0019142*T_TT**2 + 5.3E-6*T_TT**3)
    lamMoon = (125.04452222 - (5*r + 134.1362608)*T_TT +
       0.0020708*T_TT**2 + 2.2E-6*T_TT**3)
    alpha = np.array([Mmoon, Mdot, uMoon, Ddot, lamMoon])*
       deg2rad


    #nutation in lam
    del_psi = np.dot((Aj*10**-4 + Bj*10**-4*T_TT)*
       arc_sec_to_rad, np.sin(np.dot(k, alpha)))

    #nutation in obliquity
    del_epsilon = np.dot((Cj*10**-4 + Dj*10**-4*T_TT)*
       arc_sec_to_rad, np.cos(np.dot(k, alpha)))

    #mean obliquity of the ecliptic
    epsilon_m = 84381.448 - 46.8150*T_TT - 0.00059*T_TT**2 +
       0.001813*T_TT**3

    #conversion to radians
    epsilon_m = epsilon_m*arc_sec_to_rad

    #true obliquity of the ecliptic
    epsilon = epsilon_m + del_epsilon


    #nutation matrix Rot1, Rot3, Rot1
    N = Rot1(-epsilon_m) @ Rot3(del_psi) @ Rot1(epsilon)


    return N

def ECI2ECEF(r_ECI, v_ECI, JD_UTC, x_p, y_p, leap_sec, del_UT1,
    LOD):

    '''
    Converts ECI to ECEF using IAU-76/FK5
```

```
Inputs:
r_ECI: ECI position vector
JD_UTC: Julian Date in UTC
x_p: x polar motion in arc seconds
y_p: y polar motion in arc seconds
leap_sec: leap seconds
del_UT1: UT1-UTC in seconds

returns: ECI position vector
'''

# time constants
JD2000 = 2451545.0

#T_UT1
JD_UT1 = JD_UTC + del_UT1/86400
T_UT1 = (JD_UT1-JD2000)/36525

#T_TT
TAI = JD_UTC + leap_sec/86400
JD_TT = TAI + 32.184/86400
T_TT = (JD_TT-JD2000)/36525

# Polar Motion Matrix
W = PolarMotionMatrix(x_p, y_p)

# #sidereal rotation matrix
R = SiderealTimeMatrix(T_UT1, T_TT)
# # r_TOD = np.matmul(R, r_PEF)

#nutation matrix Rot1, Rot3, Rot1
N = NutationMatrix(T_TT)
# r_mod = np.matmul(N, r_TOD)

#precession matrix
P = PrecessionMatrix(T_TT)


r_ECEF = None
v_ECEF = None

if r_ECI is not None:
    r_ECEF = W.T @ R.T @ N.T @ P.T @ r_ECI
if v_ECI is not None:
```

```
        w = np.array([0, 0, 7.2921158553E-5]) #rad/s
        w_rate = w*(1-LOD/86400)

        v_ECEF = np.matmul(W.T, np.matmul(R.T, np.matmul(N.T,
           np.matmul(P.T, v_ECI) - np.cross(w_rate, np.matmul(W
           , r_ECEF)))))
    return r_ECEF, v_ECEF


def ECEF2ECI(r_ECEF, v_ECEF, a_ECEF, JD_UTC, x_p, y_p, leap_sec
   , del_UT1, LOD):

    '''
    Converts ECEF to ECI using IAU-76/FK5

    Inputs:
    r_ECEF: ECEF position vector
    JD_UTC: Julian Date in UTC
    x_p: x polar motion in arc seconds
    y_p: y polar motion in arc seconds
    leap_sec: leap seconds
    del_UT1: UT1-UTC in seconds

    returns: ECI position vector
    '''

    # time constants
    JD2000 = 2451545.0

    #T_UT1
    JD_UT1 = JD_UTC + del_UT1/86400
    T_UT1 = (JD_UT1-JD2000)/36525

    #T_UT1
    TAI = JD_UTC + leap_sec/86400
    JD_TT = TAI + 32.184/86400
    T_TT = (JD_TT-JD2000)/36525
    w = np.array([0, 0, 7.2921158553E-5]) #rad/s
    w_rate = w*(1-LOD/86400)
    # print(np.cross(w_rate, np.matmul(W, r_ECEF)))

     # Polar Motion Matrix
    W = PolarMotionMatrix(x_p, y_p)
```

```
    #sidereal rotation matrix
    R = SiderealTimeMatrix(T_UT1, T_TT)



    #nutation matrix Rot1, Rot3, Rot1
    N = NutationMatrix(T_TT)



    #precession matrix
    P = PrecessionMatrix(T_TT)


    r_ECI = None
    a_ECI = None
    v_ECI = None
    if r_ECEF is not None:
        r_ECI = np.matmul(np.matmul(np.matmul(np.matmul(P, N),
           R), W), r_ECEF)
    if v_ECEF is not None:
        # v_ECI = np.matmul(np.matmul(np.matmul(P, N), R), (np.
           matmul(W, v_ECEF) + np.cross(w_rate, np.matmul(W,
           r_ECEF))))
        v_ECI = P @ N @ R @ (W @ v_ECEF + np.cross(w_rate, W @
           r_ECEF))
    if a_ECEF is not None:
        a_ECI = P @ N @ R @ (W @ a_ECEF)

    return r_ECI, v_ECI, a_ECI

def sun_position_vector(JD_UTC, leap_sec, del_UT1):
    '''
    Returns the position vector of the sun in ECI coordinates

    Inputs:
        JD_UTC: Julian Date in UTC
        del_UT1: difference between UT1 and UTC in milliseconds
        leap_sec: number of leap seconds
    Returns:
        r_ECI: position vector of the sun in ECI coordinates
    '''

    # Constants
```

```
deg2rad = np.pi / 180.0
au = 149597870.7 # Astronomical unit [km]
# Time variables

JD2000 = 2451545.0

#T_TT
TAI = JD_UTC + leap_sec/86400
JD_TT = TAI + 32.184/86400
T_TT = (JD_TT-JD2000)/36525

# Mean lam of the Sun
l = (280.460 + 36000.771285 * T_TT) %360

# Mean anomaly of the Sun
M = (357.528 + 35999.050957 * T_TT) %360

# Ecliptic lam of the Sun
lambda_sun = l + 1.915 * np.sin(M * deg2rad) + 0.020 * np.
   sin(2 * M * deg2rad)

# Obliquity of the ecliptic
epsilon = 23.439291 - 0.01461 * T_TT

#magnitude of the sun
R = 1.00014 - 0.01671 * np.cos(M * deg2rad) - 0.00014 * np.
   cos(2 * M * deg2rad)

#sun position vector in ecliptic coordinates
r_ecliptic = np.array([R * np.cos(lambda_sun * deg2rad),
                       R * np.cos(epsilon * deg2rad) * np.
                          sin(lambda_sun * deg2rad),
                       R * np.sin(epsilon * deg2rad) * np.
                          sin(lambda_sun * deg2rad)])

#rotation from TOD to ECI

#nutation matrix Rot1, Rot3, Rot1
N = NutationMatrix(T_TT)

# r_mod = np.matmul(N, r_TOD)

#precession matrix
P = PrecessionMatrix(T_TT)
```

```
    #sun position vector in ECI
    r_ECI = np.matmul(P, np.matmul(N, r_ecliptic))*au

    return r_ECI


def moon_position_vector(JD_UTC, leap_sec, del_UT1):
    '''
    Calculates the position vector of the moon in ECI
       coordinates

    Inputs:
    JD_UTC - Julian date in UTC
    del_UT1 - UT1-UTC in ms
    leap_sec - leap seconds

    Returns:
    r_ECI - position vector of the moon in ECI coordinates
    '''

    # Constants
    deg2rad = np.pi / 180.0
    # Time variables

    JD2000 = 2451545.0

    #T_UT1
    JD_UT1 = JD_UTC + del_UT1/86400
    T_UT1 = (JD_UT1-JD2000)/36525

    #T_TT
    TAI = JD_UTC + leap_sec/86400

    JD_TT = TAI + 32.184/86400
    T_TT = (JD_TT-JD2000)/36525

    # Mean lam of the Moon
    l = (218.32 + 481267.8813*T_TT + 6.29*np.sin((134.9 +
       477198.85*T_TT)*deg2rad) \
         - 1.27*np.sin((259.2 - 413335.38*T_TT)*deg2rad) + 0.66*
            np.sin((235.7 + 890534.23*T_TT)*deg2rad) \
         + 0.21*np.sin((269.9 + 954397.70*T_TT)*deg2rad) - 0.19*
            np.sin((357.5 + 35999.05*T_TT)*deg2rad) \
         - 0.11*np.sin((186.6 + 966404.05*T_TT)*deg2rad)) % 360
```

```
#ecliptic lattitude of the Moon
phi = (5.13*np.sin((93.3 + 483202.03*T_TT)*deg2rad) + 0.28*
   np.sin((228.2 + 960400.87*T_TT)*deg2rad) \
    - 0.28*np.sin((318.3 + 6003.18*T_TT)*deg2rad) - 0.17*np
       .sin((217.6 - 407332.20*T_TT)*deg2rad)) %360

# Horizontal parallax of the Moon
O = (0.9508 + 0.0518*np.cos((134.9 + 477198.85*T_TT)*
   deg2rad) \
    + 0.0095*np.cos((259.2 - 413335.38*T_TT)*deg2rad) +
       0.0078*np.cos((235.7 + 890534.23*T_TT)*deg2rad) \
    + 0.0028*np.cos((269.9 + 954397.70*T_TT)*deg2rad)) %
       360

#oblquity of the ecliptic
epsilon = (23.439291 - 0.0130042*T_TT - 1.64E-7*T_TT**2 +
   5.04E-7*T_TT**3) % 360
#magnitude of the vector from the Earth to the Moon
R_earth = 6378.1363 #km
r_moon = R_earth/np.sin(O*deg2rad)
#moon position vector in ecliptic coordinates
r_ecliptic = np.array([r_moon*np.cos(phi*deg2rad)*np.cos(l*
   deg2rad), \
                       r_moon*(np.cos(epsilon*deg2rad)*np.
                          cos(phi*deg2rad)*np.sin(l*deg2rad
                          ) - np.sin(epsilon*deg2rad)*np.
                          sin(phi*deg2rad)), \
                        r_moon*(np.sin(epsilon*deg2rad)*np.
                           cos(phi*deg2rad)*np.sin(l*
                           deg2rad) + np.cos(epsilon*
                           deg2rad)*np.sin(phi*deg2rad))])

#rotation from TOD to ECI

#nutation matrix Rot1, Rot3, Rot1
N = NutationMatrix(T_TT)
# r_mod = np.matmul(N, r_TOD)

#precession matrix
P = PrecessionMatrix(T_TT)

#sun position vector in ECI
r_ECI = np.matmul(P, np.matmul(N, r_ecliptic))
```

```
    return r_ECI

def satellite_motion_phi(t, R, A, JD):
    '''Calculates the derivative of the state vector for the
       satellite motion
    Inputs:
        t: time in seconds
        R: state vector
        A: matrix of the linearized dynamics
        JD: Julian date in UTC

    Outputs:
        r_ddot: derivative of the position vector
        phi: derivative of the state vector

    '''

    phi = R[6:].reshape(6, 6)
    r = R[0:3]
    r_dot = R[3:6]
    x, y, z = R[0:3]
    x_dot, y_dot, z_dot = R[3:6]
    leap_sec = 37
    JD += t/86400
    # print('JD', JD)

    x_p, y_p, del_UT1, LOD = interp_EOP(JD)

    #gravity
    # r_ddot_grav = a_gravity_J2(r)
    r_ECEF, _ = ECI2ECEF(r, None, JD, x_p, y_p, leap_sec,
       del_UT1, LOD)

    g_ECEF = grav_odp(r_ECEF)
    _, _, r_ddot_grav = ECEF2ECI(None, None, g_ECEF, JD, x_p,
       y_p, leap_sec, del_UT1, LOD)

    #drag
    leap_sec = 37
    r_sun = np.zeros((1, 3))
    r_sun[0] = sun_position_vector(JD, leap_sec, del_UT1)

    A_Cross = 6E-6 #[km^2]
    C_D = 1.88
    # r_ddot_drag = a_drag(C_D, r, r_dot, A_Cross)
```

```
    r_ddot_drag = a_drag_sol(C_D, r, r_dot, r_sun[0])
    # r_ddot_drag = dragAccel2(C_D, r, r_dot, r_sun[0])


    #solar

    # d_UT1 = 196.5014/1000 #[s]
    C_s = 0.04
    C_d = 0.04
    A_Cross_sol = 6E-6 #[km^2]
    r_ddot_sol = a_solar(r, r_sun[0], C_s, C_d, A_Cross_sol)
    # r_ddot_sol = a_solar_facet(r, r_dot, r_sun[0])




    #third body
    r_moon = np.zeros((1, 3))
    r_moon[0] = moon_position_vector(JD, leap_sec, del_UT1)
    r_ddot_tb = a_third_body(r, r_sun[0], r_moon[0])



    #total acceleration
    r_ddot = r_ddot_grav + r_ddot_drag  + r_ddot_sol +
       r_ddot_tb
    #

    #A matrix
    A_1 = np.array(A(x, y, z, x_dot, y_dot, z_dot, C_D, A_Cross
       , A_Cross_sol, r_sun, r_moon))

    #state transition matrix
    phi_dot = A_1 @ phi

    dydt = np.concatenate((r_dot, r_ddot, phi_dot.ravel()))

    return dydt

def satellite_motion(t, R, JD):
    '''
    Calculates the state vector of a satellite in ECI
       coordinates

    '''
```

```
leap_sec = 37
r = R[0:3]
r_dot = R[3:6]
JD += t/86400


x_p, y_p, del_UT1, LOD = interp_EOP(JD)
#J2
# r_ddot_grav = a_gravity_J2(r)
r_ECEF, _ = ECI2ECEF(r, None, JD, x_p, y_p, leap_sec,
   del_UT1, LOD)

g_ECEF = grav_odp(r_ECEF)
# print('g_ECEF', g_ECEF)
_, _, r_ddot_grav = ECEF2ECI(None, None, g_ECEF, JD, x_p,
   y_p, leap_sec, del_UT1, LOD)


#drag
leap_sec = 37
# del_UT1 = 196.5014 #[s]
r_sun = np.zeros((1, 3))
r_sun[0] = sun_position_vector(JD, leap_sec, del_UT1)
A_Cross = 6E-6 #km^2
C_D = 1.88
# r_ddot_drag = a_drag(C_D, r, r_dot, A_Cross)
r_ddot_drag = a_drag_sol(C_D, r, r_dot, r_sun[0])
# r_ddot_drag = dragAccel2(C_D, r, r_dot, r_sun[0])

#solar

C_s = 0.04
C_d = 0.04
A_Cross_sol = 6E-6 #km^2
r_ddot_sol = a_solar(r, r_sun[0], C_s, C_d, A_Cross_sol)
# r_ddot_sol = a_solar_facet(r, r_dot, r_sun[0])


#third body
r_moon = np.zeros((1, 3))
r_moon[0] = moon_position_vector(JD, leap_sec, del_UT1)
r_ddot_tb = a_third_body(r, r_sun[0], r_moon[0])
```

```
    #total acceleration
    r_ddot = r_ddot_grav + r_ddot_drag  + r_ddot_sol +
       r_ddot_tb

    dydt = np.concatenate((r_dot, r_ddot))

    return dydt


def light_time_correction(JD, r_0, v_0, ECI_station, station):
    '''Light time correction for satellite position and
       velocity
    Inputs:
        JD_UTC: Julian date in UTC
        r_0: satellite position vector in ECI at time t
        v_0: satellite velocity vector in ECI at time t
        station: station position vector in ECEF at time t

    Outputs:
        r_0: satellite position vector in ECI at time t - lt
        v_0: satellite velocity vector in ECI at time t - lt
    '''

    x_p, y_p, del_UT1, LOD = interp_EOP(JD)
    c = 299792.458 #km/s
    rho_station = np.linalg.norm(r_0 - ECI_station)
    lt = rho_station/c
    threshold = 1E-6
    old_X_lt = r_0
    JD_lt = JD - lt/86400
    times = np.linspace(lt, 0, 11)

    y0 = np.concatenate((r_0, v_0))
    sol = solve_ivp(satellite_motion, [lt, 0], t_eval=times, y0
       =y0, args=(JD_lt,), rtol=3E-14, atol=1E-16)
    new_X_lt = sol.y[0:6, -1]
    delta = np.linalg.norm(new_X_lt[0:3] - old_X_lt)

    while delta > threshold:
        old_X_lt = new_X_lt
        x_p, y_p, del_UT1, LOD = interp_EOP(JD_lt)
        # new_station, _, _ = ECEF2ECI(station, np.array
           ([0,0,0]), None, JD_lt, x_p, y_p, leap_sec, del_UT1,
            LOD)
```

```
        new_rho = np.linalg.norm(new_X_lt[0:3] - ECI_station)
        lt = new_rho/c
        JD_lt -= lt/86400
        # print('lighttime pos diff', np.linalg.norm(new_X_lt
           [0:3] - r_0))
        times = np.linspace(lt, 0, 11)
        sol = solve_ivp(satellite_motion, [lt, 0], y0=y0,
           t_eval=times, args=(JD_lt,), rtol=3E-14, atol=1E-16)

        new_X_lt = sol.y[0:6, -1]
        delta = np.linalg.norm(new_X_lt[0:3] - old_X_lt[0:3])

    return new_X_lt

def A_Matrix(drag=True, gravity=True, solar=True, third_body=
   True):
    '''Calculates the A matrix for the equations of motion

    Inputs:
    drag: boolean, if True, drag is included in the equations
       of motion
    gravity: boolean, if True, gravity is included in the
       equations of motion
    solar: boolean, if True, solar radiation pressure is
       included in the equations of motion
    third_body: boolean, if True, third body perturbations are
       included in the equations of motion

    Outputs:
    A: A_Matrix function
    '''

    #base equation of motion
    x = sym.Symbol('x')
    y = sym.Symbol('y')
    z = sym.Symbol('z')
    A_Cross= sym.Symbol('A_Cross')
    A_Cross_Sol = sym.Symbol('A_Cross_Sol')
    x_dot = sym.Symbol('x_dot')
    y_dot = sym.Symbol('y_dot')
    z_dot = sym.Symbol('z_dot')
    C_D = sym.Symbol('C_D')
    r = (x**2 + y**2 + z**2)**(1/2)
    mu = 398600.4415 #km^3/s^2
```

```
#with gravity
if gravity:
    R_earth = 6378.1363 #[km]
    J_2 = 0.00108248
    phi = z/r
    F_x = sym.diff(mu/r*(1-J_2*(R_earth/r)**2*(3/2*phi
       **2-1/2)), x)
    F_y = sym.diff(mu/r*(1-J_2*(R_earth/r)**2*(3/2*phi
       **2-1/2)), y)
    F_z = sym.diff(mu/r*(1-J_2*(R_earth/r)**2*(3/2*phi
       **2-1/2)), z)


#with atmospheric drag
if drag:

    R_earth = 6378.1363 #[km]
    m = 2000 #[kg]
    theta_dot = 7.292115146706979E-5 #[rad/s]
    rho_0 = 3.614E-4 #[kg/km^3]
    H = 88667.0/1E3 #[km]
    r0 = (700.0 + R_earth) #[km]

    rho_A = rho_0*sym.exp(-(r-r0)/H)

    V_A_bar = sym.Matrix([x_dot+theta_dot*y, y_dot-
       theta_dot*x, z_dot])
    V_A = sym.sqrt((x_dot + theta_dot*y)**2 + (y_dot-
       theta_dot*x)**2 + z_dot**2)

    r_ddot = -1/2*C_D*A_Cross/m*rho_A*V_A*V_A_bar
    F_x += r_ddot[0]
    F_y += r_ddot[1]
    F_z += r_ddot[2]

#with solar radiation pressure
if solar:
    r_sun = sym.MatrixSymbol('r_sun', 1, 3)
    m = 2000 #kg
    AU = 149597870.7 #m
    d = ((r_sun[0] - x)**2 + (r_sun[1] - y)**2 + (r_sun[2]
       - z)**2)**(1/2)/AU

    P = 4.56E-3 #N/km^2
    C_r = 0.63
```

```
    F_x += -P*A_Cross_Sol/m*C_r*(r_sun[0]-x)/d**2
    F_y += -P*A_Cross_Sol/m*C_r*(r_sun[1]-y)/d**2
    F_z += -P*A_Cross_Sol/m*C_r*(r_sun[2]-z)/d**2

#with thrird body perturbations
if third_body:
    mu_sun = 132712440018 #[km^3/s^2]
    mu_moon = 4902.800066 #[km^3/s^2]
    r_sun = sym.MatrixSymbol('r_sun', 1, 3)
    r_moon = sym.MatrixSymbol('r_moon', 1, 3)
    r_sun_mag = (r_sun[0]**2 + r_sun[1]**2 + r_sun[2]**2)
       **(1/2)
    r_moon_mag = (r_moon[0]**2 + r_moon[1]**2 + r_moon
       [2]**2)**(1/2)

    del_sun_mag = ((r_sun[0]-x)**2 + (r_sun[1]-y)**2 + (
       r_sun[2]-z)**2)**(1/2)
    del_moon_mag = ((r_moon[0]-x)**2 + (r_moon[1]-y)**2 + (
       r_moon[2]-z)**2)**(1/2)
    F_x += mu_sun*((r_sun[0]-x)/(del_sun_mag)**3 - r_sun
       [0]/r_sun_mag**3) + mu_moon*((r_moon[0]-x)/(
       del_moon_mag**3) - r_moon[0]/r_moon_mag**3)
    F_y += mu_sun*((r_sun[1]-y)/(del_sun_mag)**3 - r_sun
       [1]/r_sun_mag**3) + mu_moon*((r_moon[1]-y)/(
       del_moon_mag**3) - r_moon[1]/r_moon_mag**3)
    F_z += mu_sun*((r_sun[2]-z)/(del_sun_mag)**3 - r_sun
       [2]/r_sun_mag**3) + mu_moon*((r_moon[2]-z)/(
       del_moon_mag**3) - r_moon[2]/r_moon_mag**3)

#F functions
F1 = x_dot
F2 = y_dot
F3 = z_dot
F4, F5, F6 = F_x, F_y, F_z
F7 = 0

#A matrix
A = [[sym.diff(F1, x), sym.diff(F1, y), sym.diff(F1, z),
   sym.diff(F1, x_dot), sym.diff(F1, y_dot), sym.diff(F1,
   z_dot)],
    [sym.diff(F2, x), sym.diff(F2, y), sym.diff(F2, z), sym
       .diff(F2, x_dot), sym.diff(F2, y_dot), sym.diff(F2,
       z_dot)],
```

```
            [sym.diff(F3, x), sym.diff(F3, y), sym.diff(F3, z), sym
               .diff(F3, x_dot), sym.diff(F3, y_dot), sym.diff(F3,
               z_dot)],
            [sym.diff(F4, x), sym.diff(F4, y), sym.diff(F4, z), sym
               .diff(F4, x_dot), sym.diff(F4, y_dot), sym.diff(F4,
               z_dot)],
            [sym.diff(F5, x), sym.diff(F5, y), sym.diff(F5, z), sym
               .diff(F5, x_dot), sym.diff(F5, y_dot), sym.diff(F5,
               z_dot)],
            [sym.diff(F6, x), sym.diff(F6, y), sym.diff(F6, z), sym
               .diff(F6, x_dot), sym.diff(F6, y_dot), sym.diff(F6,
               z_dot)]]

    if gravity and drag and solar and third_body:
     A = sym.lambdify([x, y, z, x_dot, y_dot, z_dot, C_D,
        A_Cross, A_Cross_Sol, r_sun, r_moon], A)

    elif gravity:
        A = sym.lambdify([x, y, z], A)

    elif drag:
        A = sym.lambdify([x, y, z, x_dot, y_dot, z_dot, C_D,
           A_Cross], A)

    elif solar:
        A = sym.lambdify([x, y, z, A_Cross_Sol, d, m], A)

    elif third_body:
        A = sym.lambdify([x, y, z, r_sun, r_moon], A)

    else:
        A = sym.lambdify([x, y, z], A)

    return A

#H_tilde
def H_tilde_matrix():
    x = sym.Symbol('x')
    y = sym.Symbol('y')
    z = sym.Symbol('z')
    x_dot = sym.Symbol('x_dot')
    y_dot = sym.Symbol('y_dot')
    z_dot = sym.Symbol('z_dot')
    C_D = sym.Symbol('C_D')
```

```
    x_s = sym.Symbol('x_s')
    y_s = sym.Symbol('y_s')
    z_s = sym.Symbol('z_s')

    x_s_dot = sym.Symbol('x_s_dot')
    y_s_dot = sym.Symbol('y_s_dot')
    z_s_dot = sym.Symbol('z_s_dot')

    bias = sym.Symbol('bias')


    rho = sym.sqrt((x - x_s)**2 + (y - y_s)**2 + (z - z_s)**2)
       - bias
    #for project omega x r ECEF frame
    #vallado chapter 4 ECEF to ECI transformation
    rho_dot = ((x-x_s)*(x_dot-x_s_dot) + (y-y_s)*(y_dot-y_s_dot
       ) + (z-z_s)*(z_dot-z_s_dot))/rho

    H_tilde_sym = [[sym.diff(rho, x), sym.diff(rho, y), sym.
       diff(rho, z), sym.diff(rho, x_dot), sym.diff(rho, y_dot)
       , sym.diff(rho, z_dot)],[
            sym.diff(rho_dot, x), sym.diff(rho_dot, y), sym.diff
               (rho_dot, z), sym.diff(rho_dot, x_dot), sym.diff(
               rho_dot, y_dot), sym.diff(rho_dot, z_dot)]]

    H_tilde_func = sym.lambdify((x, y, z, x_dot, y_dot, z_dot,
       x_s, y_s, z_s, x_s_dot, y_s_dot, z_s_dot, bias),
       H_tilde_sym, 'numpy')

    return H_tilde_func


def a_third_body(r, r_sun, r_moon):
    '''
    Calculates the acceleration due to third body perturbations

    Inputs:
        r: position vector of satellite
        r_sun: position vector of sun
        r_moon: position vector of moon

    Outputs:
        a_x: acceleration in x direction
        a_y: acceleration in y direction
        a_z: acceleration in z direction
```

```
    '''


    mu_sun = 132712440018 #km^3/s^2
    mu_moon = 4902.800066 #km^3/s^2
    r_sun_mag = np.linalg.norm(r_sun)
    r_moon_mag = np.linalg.norm(r_moon)

    del_sun_mag = np.linalg.norm(r_sun-r)
    del_moon_mag = np.linalg.norm(r_moon-r)
    a = mu_sun*((r_sun - r)/(del_sun_mag**3) - r_sun/r_sun_mag
       **3) + mu_moon*((r_moon - r)/(del_moon_mag**3) - r_moon/
       r_moon_mag**3)

    return a


def a_solar(r, s, C_s, C_d, A_Cross_sol):
    '''
    Calculates the acceleration due to solar radiation pressure

    Inputs:
        r: position vector of satellite
        s: position vector of sun
        C_s: solar radiation pressure coefficient
        C_d: solar radiation pressure coefficient
        A_Cross_sol: cross sectional area of satellite

    Outputs:
        r_ddot: acceleration vector of satellite
    '''

    r_ddot = np.zeros(3)
    tau_min = (np.linalg.norm(r)**2 - np.dot(r, s))/(np.linalg.
       norm(r)**2 + np.linalg.norm(s)**2 - 2*np.dot(r, s))
    if tau_min < 0:
        AU = 149597870.7 #km
        d = np.linalg.norm(s-r)/AU
        m = 2000 #kg
        P = 4.56E-3 #N/km^2
        C_r = 0.63
        u = (s-r)/np.linalg.norm(s-r)
        r_ddot = -P*A_Cross_sol/m*C_r/d**2*u

    return r_ddot
```

```
def a_solar_facet(R, V, Rs):
    '''
    Calculates the acceleration due to solar radiation pressure
    '''
    x = R[0]
    y = R[1]
    z = R[2]
    rs_x = Rs[0]
    rs_y = Rs[1]
    rs_z = Rs[2]
    xd = V[0]
    yd = V[1]
    zd = V[2]
    r = np.sqrt(x ** 2 + y ** 2 + z ** 2)
    sat_vec = np.array([x, y, z])
    sun_vec = np.array([rs_x, rs_y, rs_z]).T
    srp = np.array([0, 0, 0])
    tau_min = (r**2 - np.dot(sat_vec, sun_vec))/(r**2 + np.
       linalg.norm(sun_vec)**2 - 2*np.dot(sat_vec, sun_vec))

    Cd_sp = 0.04
    Cs_sp = 0.04
    Cd_xy = 0.04
    Cs_xy = 0.59
    Cd_zpos = 0.80
    Cs_zpos = 0.04
    Cd_zneg = 0.28
    Cs_zneg = 0.18
    Asp = 15E-6
    Axx = 6E-6
    Ayy = 8E-6
    Azz = 12E-6
    theta = 0
    m_sc = 2000 #kg
    au = 149597870.7 #km
    sol_cons = 1367 #W/km^2
    c = 299792.458 #km/s


    r = np.sqrt(x**2 + y**2 + z**2)
    sat_vec = np.array([x, y, z])
    sun_vec = np.array([rs_x, rs_y, rs_z]).T
    srp = np.array([0.0, 0.0, 0.0])
```

```
tau_min = (r**2 - np.dot(sat_vec, sun_vec))/(r**2 + np.
   linalg.norm(sun_vec)**2 - 2*np.dot(sat_vec, sun_vec))

# only apply srp if sat is in sunlight
if tau_min < 0:
    d = np.sqrt((rs_x - x)**2 + (rs_y - y)**2 + (rs_z - z)
       **2)/au
    nu_sp = Cd_sp/3
    nu_xy = Cd_xy/3
    nu_zpos = Cd_zpos/3
    nu_zneg = Cd_zneg/3
    mu_sp = Cs_sp/2
    mu_xy = Cs_xy/2
    mu_zpos = Cs_zpos/2
    mu_zneg = Cs_zneg/2
    B_sp = 2*nu_sp*np.cos(theta) + 4*mu_sp*np.cos(theta)**2
    A1_sp = Asp/m_sc
    C1 = sol_cons/c
    u = (sun_vec-sat_vec)/np.linalg.norm(sun_vec-sat_vec)

    n_hat = np.eye(3)
    r = sat_vec
    sat2sun = (sun_vec - sat_vec)
    v = np.array([xd, yd, zd]).T

    # Frame Definitions
    R_hat = r/np.linalg.norm(r)
    N_hat = np.cross(r, v)/np.linalg.norm(np.cross(r, v))
    T_hat = np.cross(N_hat, R_hat)

    # 1. define Qbf2
    Qbf2eci = np.vstack([R_hat, T_hat, N_hat]).T
    Qbf2eci = orthodcm(Qbf2eci)

    # 2. convert sat2sun in BF
    sat2sun_bf = np.linalg.inv(Qbf2eci) @ sat2sun/np.linalg
       .norm(sat2sun)

    # 3. take dot with normal to find angle
    theta_x = np.arccos(np.dot(sat2sun_bf, n_hat[:,0]))
    theta_y = np.arccos(np.dot(sat2sun_bf, n_hat[:,1]))
    theta_z = np.arccos(np.dot(sat2sun_bf, n_hat[:,2]))

    # 4. constraints for back side illumination
    if theta_z < 0:
```

```
            mu_z = mu_zneg
            nu_z = nu_zneg
        else:
            mu_z = mu_zpos
            nu_z = nu_zpos

        # 5. compute acceleration
        # for i in range(7):
        #     n_hat_eci[:,i] = Qbf2eci @ n_hat[:,i]
        #     if (np.dot(u, n_hat_eci[:,i]) != 0):
        #         brakt_term = areas[i]
        #     else:
        #         brakt_term = 0
        #     Across = Across + brakt_term
        B_x = 2*nu_xy*np.cos(theta_x) + 4*mu_xy*np.cos(theta_x)
           **2
        A1_x = Axx/m_sc

        B_y = 2*nu_xy*np.cos(theta_y) + 4*mu_xy*np.cos(theta_y)
           **2
        A1_y = Ayy/m_sc

        B_z = 2*nu_z*np.cos(theta_z) + 4*mu_z*np.cos(theta_z)
           **2
        A1_z = Azz/m_sc

        srp_sp = (-C1/d**2)*(B_sp*n_hat[:,1]*-1 + (1-mu_sp)*np.
           cos(theta)**2*sat2sun_bf)*A1_sp
        srp_x = (-C1/d**2)*(B_x*n_hat[:,0] + (1-mu_xy)*np.cos(
           theta_x)**2*sat2sun_bf)*A1_x
        srp_y = (-C1/d**2)*(B_y*n_hat[:,1] + (1-mu_xy)*np.cos(
           theta_y)**2*sat2sun_bf)*A1_y
        srp_z = (-C1/d**2)*(B_z*n_hat[:,2] + (1-mu_z)*np.cos(
           theta_z)**2*sat2sun_bf)*A1_z
        srp_tot = srp_sp + srp_x + srp_y + srp_z

        # 6. convert back to ECI
        srp = Qbf2eci @ srp_tot

    return srp


def a_drag(C_D, r, v, A_Cross):

    '''
```

```
    Computes the acceleration due to atmospheric drag

    Inputs:
    r - position vector in ECI frame [m]
    v - velocity vector in ECI frame [m/s]
    A_Cross - cross sectional area of satellite [m^2]

    Outputs:
    a_drag - acceleration due to atmospheric drag [m/s^2]

    '''

    #drag parameters
    R_earth = 6378.1363#[m]
    m = 2000 #[kg]
    theta_dot = 7.292115146706979E-5 #[rad/s]
    rho_0 = 3.614E-4#[kg/km^3]
    H = 88667.0/1E3 #[km]
    r0 = (700.0 + R_earth) #[m]

    r_mag = np.linalg.norm(r)
    rho_A = rho_0*np.exp(-(r_mag-r0)/H)
    V_A_bar = np.array([v[0]+theta_dot*r[1], v[1]-theta_dot*r
       [0], v[2]])
    V_A = np.sqrt((v[0] + theta_dot*r[1])**2 +(v[1]-theta_dot*r
       [0])**2 + v[2]**2)


    return -1/2*C_D*A_Cross/m*rho_A*V_A*V_A_bar


def a_drag_sol(C_D, r, v, s):

    '''
    Computes the acceleration due to atmospheric drag

    Inputs:
    C_D - drag coefficient
    r - position vector in ECI frame [m]
    v - velocity vector in ECI frame [m/s]
    r_sun - position vector of sun in ECI frame [m]

    Outputs:
    a_drag - acceleration due to atmospheric drag [m/s^2]
```

```
'''
r_mag = np.linalg.norm(r)

#drag parameters
R_earth = 6378.1363 #[m]
m = 2000 #[kg]
theta_dot = 7.292115146706979E-5 #[rad/s]


#compute the rotation angles of the satellite so that it is
    nadir pointed
phi_sat = np.arctan2(r[1], r[0])
theta_sat = np.arctan2(r[2], np.sqrt(r[0]**2 + r[1]**2))

#compute Cross sectional area of the satellite body in each
    direction
A_Cross_x = np.abs((np.pi/2 - phi_sat))/(np.pi/2)*6E-6
A_Cross_y = np.abs((phi_sat)/np.pi/2)*8E-6
A_Cross_z = np.abs((theta_sat)/np.pi/2)*12E-6

rho_0 = 3.614E-4 #[kg/km^3]
H = 88667.0/1E3 #[km]
r0 = (700.0 + R_earth) #[km]

#check if the satellite is in the shadow of the earth, if
   it is then don't add the solar panel to drag
tau_min = (np.linalg.norm(r)**2 - np.dot(r, s))/(np.linalg.
   norm(r)**2 + np.linalg.norm(s)**2 - 2*np.dot(r, s))
if tau_min < 0:

    #compute vector between sun and satellite
    r_sat_sun = s-r

    #compute phi and theta of the satellite in relation to
       the sun
    phi_sat_sun = np.arctan2(r_sat_sun[1], r_sat_sun[0])
    theta_sat_sun = np.arctan2(r_sat_sun[2], np.sqrt(
       r_sat_sun[0]**2 + r_sat_sun[1]**2))

    #compute the amount of the solar panel in front of the
       satellite
    solar_theta = (np.pi/2 - np.abs(theta_sat_sun -
       theta_sat))/(np.pi/2)*15E-6
```

```
            #compute Cross sectional area of the solar panel in
               each direction
            A_Cross_x += np.abs(np.pi/2-np.abs(phi_sat_sun -
               phi_sat))/(np.pi/2)*solar_theta
            A_Cross_y += np.abs(phi_sat_sun - phi_sat)/(np.pi/2)*
               solar_theta
            A_Cross_z += np.abs(theta_sat_sun - theta_sat)/(np.pi
               /2)*solar_theta

        # A_Cross = np.array([A_Cross_x, A_Cross_y, A_Cross_z])
        A_Cross = A_Cross_x + A_Cross_y + A_Cross_z
        rho_A = rho_0*np.exp(-(r_mag-r0)/H)
        V_A_bar = np.array([v[0]+theta_dot*r[1], v[1]-theta_dot*r
           [0], v[2]])
        V_A = np.sqrt((v[0] + theta_dot*r[1])**2 +(v[1]-theta_dot*r
           [0])**2 + v[2]**2)


        return -1/2*C_D*rho_A*V_A*A_Cross*V_A_bar/m


    def a_drag_sol_2(C_D, r, v, s):

        '''
        Computes the acceleration due to atmospheric drag

        Inputs:
        C_D - drag coefficient
        r - position vector in ECI frame [m]
        v - velocity vector in ECI frame [m/s]
        r_sun - position vector of sun in ECI frame [m]

        Outputs:
        a_drag - acceleration due to atmospheric drag [m/s^2]

        '''
        # Calculate the magnitude of the position vector
        r_mag = np.linalg.norm(r)

        # Define drag parameters
        R_earth = 6378.1363 #[m]
        m = 2000 #[kg]
        theta_dot = 7.292115146706979E-5 #[rad/s]
        C_D = 1.88 # Drag coefficient
```

```
# Calculate velocity in atmosphere
V_A_bar = np.array([v[0]+theta_dot*r[1], v[1]-theta_dot*r
   [0], v[2]])

areas = np.array([6, 6, 8, 8, 12, 12, 15])*1E-6
A_cross = 0

# Create unit vectors in body frame
n_hat = np.array([[1, 0, 0],
                  [-1, 0, 0],
                  [0, 1, 0],
                  [0, -1, 0],
                  [0, 0, 1],
                  [0, 0, -1],
                  [0, -1, 0]])
n_hat_eci = np.zeros((7, 3))

T_hat = v/np.linalg.norm(v)
W_hat = np.cross(r, v)/np.linalg.norm(np.cross(r, v))
N_hat = np.cross(T_hat, W_hat)

# Transform from body frame to ECI frame
Qbf2eci = np.vstack([T_hat, W_hat, N_hat])

Qbf2eci = orthodcm(Qbf2eci)

# Calculate components of n_hat in ECI frame
for i in range(len(n_hat)):
    n_hat_eci[i] = Qbf2eci @ n_hat[i]
    if not np.isclose(np.dot(T_hat, n_hat_eci[i]), 0, 1E
       -16):
        facet_a = areas[i]
    else:
        facet_a = 0
    A_cross += facet_a
print('A_cross: ', A_cross)



# Calculate cross sectional area for each component

# A_Cross = np.zeros(3)
# A_Cross[0] = (n_hat_eci_x @ T_hat)*(6E-6)
# A_Cross[1] = (n_hat_eci_y @ T_hat)*(8E-6)
# A_Cross[2] = (n_hat_eci_z @ T_hat)*(12E-6)
```

```
    # Define atmospheric parameters
    rho_0 = 3.614E-4 #[kg/km^3]
    H = 88667.0/1E3 #[km]
    r0 = (700.0 + R_earth) #[m]

    # # # Check if satellite is in shadow of earth
    # tau_min = (np.linalg.norm(r)**2 - np.dot(r, s))/(np.
       linalg.norm(r)**2 + np.linalg.norm(s)**2 - 2*np.dot(r, s
       ))
    # if tau_min < 0:
    # Compute vector between sun and satellite
    # r_sat_sun = (s-r)/np.linalg.norm(s-r)

    # Calculate solar panel contribution to drag
    # n_hat_bf_sol = np.cross(n_hat, r_sat_sun)
    # n_hat_eci_sol = (Qbf2eci @ n_hat_bf_sol)

    # A_Cross[0] += (n_hat_eci_sol[0] @ T_hat)*(15E-6)
    # A_Cross[1] += (n_hat_eci_sol[1] @ T_hat)*(15E-6)
    # A_Cross[2] += (n_hat_eci_sol[2] @ T_hat)*(15E-6)

    # Calculate total cross sectional area
    # A_Cross = np.array([A_Cross_x, A_Cross_y, A_Cross_z])


    # Calculate atmospheric density
    rho_A = rho_0*np.exp(-(r_mag-r0)/H)

    # Calculate velocity in atmosphere
    V_A = np.sqrt((v[0] + theta_dot*r[1])**2 +(v[1]-theta_dot*r
       [0])**2 + v[2]**2)

    # Return drag force
    return -1/2*C_D*A_cross/m*rho_A*V_A*V_A_bar

def dragAccel2(Cd, R, V, r_sun):

    x = R[0]
    y = R[1]
    z = R[2]

    xd = V[0]
```

```
yd = V[1]
zd = V[2]

rs_x = r_sun[0]
rs_y = r_sun[1]
rs_z = r_sun[2]

R_earth = 6378.1363 #[m]
m_sc = 2000 #[kg]
w = 7.292115146706979E-5 #[rad/s]
C_D = 1.88 # Drag coefficient
rho0 = 3.614E-4 #[kg/km^3]
H = 88667.0/1E3 #[km]
r0_drag = (700.0 + R_earth) #[m]
Axx = 6E-6
Ayy = 8E-6
Azz = 12E-6
Asp = 15E-6


r = np.array([x,y,z])
v = np.array([xd, yd, zd])
r_mag = np.sqrt(x**2 + y**2 + z**2)
sun_vec = np.array([rs_x, rs_y, rs_z])
sat2sun = (sun_vec-r)/np.linalg.norm(sun_vec-r)
rho_A = rho0*np.exp(-(r_mag-r0_drag)/H)
V_A_bar = [xd + w*y, yd - w*x, zd]
V_A = np.sqrt((xd + w*y)**2 + (yd - w*x)**2 + zd**2)

n_hat = np.identity(3)

# Frame Definitions
T_hat = v/np.linalg.norm(v)
W_hat = np.cross(r, v)/np.linalg.norm(np.cross(r, v))
N_hat = np.cross(T_hat, W_hat)

# 1. define Qbf2
Qbf2eci = np.transpose(np.vstack((N_hat, T_hat, W_hat)))
Qbf2eci = orthodcm(Qbf2eci)

# 2. convert sat2sun in BF
sat2sun_bf = np.linalg.inv(Qbf2eci) @ sat2sun

# 3. take dot with normal to find angle
theta_x = np.arccos(np.abs(np.dot(T_hat, n_hat[0,:])))
```

```
    theta_y = np.arccos(np.abs(np.dot(T_hat, n_hat[1,:])))
    theta_z = np.arccos(np.abs(np.dot(T_hat, n_hat[2,:])))
    theta_srp = np.arccos(np.abs(np.dot(T_hat, sat2sun_bf)))

    # 5. compute acceleration
    Across_x = np.cos(theta_x)*Axx*T_hat
    Across_y = np.cos(theta_y)*Ayy*T_hat
    Across_z = np.cos(theta_z)*Azz*T_hat
    Across_sp = np.cos(theta_srp)*Asp*T_hat
    drag_x = -1/2*Cd*rho_A*V_A*(Across_x*np.transpose(V_A_bar))
       /m_sc
    drag_y = -1/2*Cd*rho_A*V_A*(Across_y*np.transpose(V_A_bar))
       /m_sc
    drag_z = -1/2*Cd*rho_A*V_A*(Across_z*np.transpose(V_A_bar))
       /m_sc
    tau_min = (np.linalg.norm(r)**2 - np.dot(r, r_sun))/(np.
       linalg.norm(r)**2 + np.linalg.norm(r_sun)**2 - 2*np.dot(
       r, r_sun))
    if tau_min < 0:

        drag_sp = -1/2*Cd*rho_A*V_A*(Across_sp*np.transpose(
           V_A_bar))/m_sc

        drag_sp = np.zeros(3)
    drag_tot = drag_x + drag_y + drag_z + drag_sp

    # 6. convert back to ECI
    drag = Qbf2eci @ drag_tot

    return drag

def a_gravity_J2(r):
    '''
    Computes the acceleration due to J2 perturbation

    Inputs:
    r - position vector in ECI frame [m]

    Outputs:
    a_gravity_J2 - acceleration due to J2 perturbation [m/s^2]
    '''
    x = r[0]
    y = r[1]
    z = r[2]
    J2 = 0.00108248
```

```
    Re = 6378.1363
    mu = 398600.4415
        # The equation for calculating dUdx
    dUdx = -(2*mu*x*(x**2 + y**2 + z**2)**2 - 15*J2*Re**2*mu*x*
       z**2 +
            3*J2*Re**2*mu*x*(x**2 + y**2 + z**2))/(2*(x**2 + y
               **2 + z**2)**(7/2))

    # The equation for calculating dUdy
    dUdy = -(2*mu*y*(x**2 + y**2 + z**2)**2 - 15*J2*Re**2*mu*y*
       z**2 +
            3*J2*Re**2*mu*y*(x**2 + y**2 + z**2))/(2*(x**2 + y
               **2 + z**2)**(7/2))

    # The equation for calculating dUdz
    dUdz = -(2*mu*z*(x**2 + y**2 + z**2)**2 - 15*J2*Re**2*mu*z
       **3 +
            9*J2*Re**2*mu*z*(x**2 + y**2 + z**2))/(2*(x**2 + y
               **2 + z**2)**(7/2))

    a_gravity_J2 = np.array([dUdx, dUdy, dUdz])

    return a_gravity_J2

def load_egm96_coefficients():
    '''
    Loads the EGM96 coefficients from the CSV files

    Inputs:
    None

    Outputs:
    C - EGM96 C coefficients
    S - EGM96 S coefficients
    '''

    EGM96_C_file = 'EGM96_C.csv'
    EGM96_S_file = 'EGM96_S.csv'

    # Load EGM96 coefficients from file
    C = []
    S = []
    # Read in the CSV file and populate the matrix
    with open(EGM96_C_file, 'r') as file:
        csv_reader = csv.reader(file)
```

```
        for row in csv_reader:
            C.append(row)
    # S = np.loadtxt(EGM96_S_file)
    with open(EGM96_S_file, 'r') as file:
        csv_reader = csv.reader(file)
        for row in csv_reader:
            S.append(row)
    C = np.array(C).astype(float)
    S = np.array(S).astype(float)
    return C, S

C, S = load_egm96_coefficients()

@jit
def grav_odp(r):

    degree = 20
    C_deg = C[0:degree, 0:degree]
    S_deg = S[0:degree, 0:degree]
    # print(C_deg.shape)

    x, y, z = r
    mu = 398600.4415 #km^3/s^2
    RE = 6378.1363 #km
    nmaxp1, mmaxp1 = C_deg.shape

    nmax = nmaxp1 - 1
    mmax = mmaxp1 - 1
    Anm = np.zeros(nmaxp1+1)
    Anm1 = np.zeros(nmaxp1+1)
    Anm2 = np.zeros(nmaxp1+2)
    R = np.zeros(nmaxp1+1)
    I = np.zeros(nmaxp1+1)
    rb2 = x*x + y*y + z*z
    rb = np.sqrt(rb2)
    mur2 = mu/rb2
    mur3 = mur2/rb

    # direction of spacecraft position
    s = x/rb
    t = y/rb
    u = z/rb

    # Calculate contribution of only Zonals
    Anm1[0] = 0
```

```
Anm1[1] = np.sqrt(3)

Anm2[1] = 0
Anm2[2] = np.sqrt(3.75)

as_ = 0
at = 0
au = 0
ar = 0
rat1 = 0
rat2 = 0
Dnm = 0
Enm = 0
Fnm = 0
Apor = np.zeros(nmaxp1)
# print(Apor.shape)
Apor[0] = 1
Apor[1] = RE/rb

for n in range(1, nmax +1):
    i = n+1
    an2 = 2*n
    rat1 = np.sqrt((an2+3.0)*(((an2+1.0)/n)/(n+2.0)))
    rat2 = np.sqrt((n+1.0)*(((n-1.0)/(an2-1.0))/(an2+1.0)))
    Anm1[i] = rat1*(u*Anm1[i-1] - rat2*Anm1[i-2])
    Apor[i-1] = Apor[i-2]*Apor[1]
    if n < mmaxp1:
        rat1 = np.sqrt((an2+5.0)*(((an2+3.0)/n)/(n+4.0)))
        rat2 = np.sqrt((n+3.0)*(((n-1.0)/(an2+1.0))/(an2
           +3.0)))
        Anm2[i+1] = rat1*(u*Anm2[i] - rat2*Anm2[i-1])
    if n < nmaxp1:
        rat1 = np.sqrt(0.5*n*(n+1.0))
        au = au - Apor[i-1]*rat1*Anm1[i-1]*(-C_deg[i-1, 0])
        rat2 = np.sqrt(0.5*((an2+1.0)/(an2+3.0))*(n+1.0)*(n
           +2.0))
        ar = ar + Apor[i-1]*rat2*Anm1[i]*(-C_deg[i-1, 0])

# Calculate contribution of Tesserals
R = np.array([1], dtype=np.float64) # Initialize R as a
   numpy array with one element = 1
I = np.array([0], dtype=np.float64) # Initialize I as a
   numpy array with one element = 0
```

```
for m in range(1, mmax + 1): # Loop from 1 to mmax (
   inclusive)
    j = m
    am2 = 2*m
    R = np.append(R, s*R[j-1] - t*I[j-1])
    I = np.append(I, s*I[j-1] + t*R[j-1])

    for l in range(m, mmax + 1): # Loop from the current
       value of m to mmax (inclusive)
        i = l
        Anm[i] = Anm1[i]
        Anm1[i] = Anm2[i]

    Anm1[mmax+1] = Anm2[mmax+1]

    for l in range(m, mmax + 1):
        i = l
        an2 = 2*l
        if l == m:
            Anm2[j+1] = 0.0
            Anm2[j+2] = np.sqrt((am2+5.0)/(am2+4.0))*Anm1[j
               +1]
        else:
            rat1 = np.sqrt((an2+5.0)*(((an2+3.0)/(l-m))/(l+
               m+4.0)))
            rat2 = np.sqrt((l+m+3.0)*(((l-m-1.0)/(an2+1.0))
               /(an2+3.0)))
            Anm2[i+2] = rat1*(u*Anm2[i+1] - rat2*Anm2[i])

        Dnm = C_deg[i,j]*R[j] + S_deg[i,j]*I[j]
        Enm = C_deg[i,j]*R[j-1] + S_deg[i,j]*I[j-1]
        Fnm = S_deg[i,j]*R[j-1] - C_deg[i,j]*I[j-1]

        rat1 = np.sqrt((l+m+1.0)*(l-m))
        rat2 = np.sqrt(((an2+1.0)/(an2+3.0))*(l+m+1.0)*(l+m
           +2.0))

        as_ += Apor[i]*m*Anm[i]*Enm
        at += Apor[i]*m*Anm[i]*Fnm
        au += Apor[i]*rat1*Anm1[i]*Dnm
        ar -= Apor[i]*rat2*Anm1[i+1]*Dnm

agx_ECEF = -mur3*x + mur2*(as_ + ar*s)
agy_ECEF = -mur3*y + mur2*(at + ar*t)
agz_ECEF = -mur3*z + mur2*(au + ar*u)
```

```
    return np.array([agx_ECEF, agy_ECEF, agz_ECEF])


def range_range_rate(r, v, station_ECI, station_dot_ECI):
    '''
    Compute range and range rate from a station to a satellite

    Inputs:
        r - position vector of satellite in ECI frame [m]
        v - velocity vector of satellite in ECI frame [m/s]
        station_ECI - position vector of station in ECI frame [
           m]
        station_dot_ECI - velocity vector of station in ECI
           frame [m/s]

    Outputs:
        rho - range from station to satellite [m]
        rho_dot - range rate from station to satellite [m/s]
    '''

    rho = np.linalg.norm(r - station_ECI)
    rho_dot = ((r - station_ECI) @ (v - station_dot_ECI))/rho

    return rho, rho_dot

def EKF(case, r_ECI, v_ECI, stations, JD_UTC, seconds, Ps, Psv,
    Qs, a_bias):
    '''
    Extended Kalman Filter for orbit determination

    Inputs:
        case - case number
        r_ECI - initial position vector of satellite in ECI
           frame [m]
        v_ECI - initial velocity vector of satellite in ECI
           frame [m/s]
        stations - matrix of station locations
        JD_UTC - Julian date in UTC
        seconds - seconds of propagation
        Ps - initial covariance of position
        Psv - initial covariance velocity
        Qs - process noise sigma
```

```
Outputs:
    y_arr - array of state vectors
    P_hat - array of covariance matrices
    P_ZZ - array of innovation covariance matrices
    obs_mat - modified data of observations
    Q - process noise matrix

'''
print('Case: ', case)
obs_mat_3d = loadmat('LEO_DATA_Apparent_3Days.mat')
obs_mat_4_6 = loadmat('LEO_DATA_Apparent_Days4-6.mat')
obs_mat_4_6['LEO_DATA_Apparent'][:,1] += 3*86400
obs_mat = np.concatenate((obs_mat_3d['LEO_DATA_Apparent'],
   obs_mat_4_6['LEO_DATA_Apparent']), axis=0)

if case == 'C':
    obs_mat = obs_mat[obs_mat[:, 0] == 1]


elif case == 'D':
    obs_mat = obs_mat[obs_mat[:, 0] == 2]
    # obs_mat = obs_mat[obs_mat[:, 0] != 3]

elif case == 'E':
    obs_mat = obs_mat[obs_mat[:, 0] == 3]

elif case == 'G':
    obs_mat = obs_mat[obs_mat[:, 1] >= 5*86400]




#Initialize Extended Kalman Filter


y0 = np.concatenate((r_ECI, v_ECI))

#initial covariance


sigma_x = Ps
sigma_y = Ps
sigma_z = Ps
sigma_vx = Psv
sigma_vy = Psv
```

```
sigma_vz = Psv


#initial covariance matrix
P_hat_0 = np.diag([sigma_x**2, sigma_y**2, sigma_z**2,
   sigma_vx**2, sigma_vy**2, sigma_vz**2])

#process noise
sigma_x = Qs
sigma_y = Qs
sigma_z = Qs

# sigma_x = 0
# sigma_y = 0
# sigma_z = 0

del_t = sym.Symbol('del_t')
Q_sym = del_t**2*sym.Matrix([[del_t**2/4*sigma_x**2, 0, 0,
   del_t/2*sigma_x**2, 0, 0],
                        [0, del_t**2/4*sigma_y**2, 0, 0,
                           del_t/2*sigma_y**2, 0],
                        [0, 0, del_t**2/4*sigma_z**2, 0, 0,
                            del_t/2*sigma_z**2],
                        [del_t/2*sigma_x**2, 0, 0, sigma_x
                           **2, 0, 0],
                        [0, del_t/2*sigma_y**2, 0, 0,
                           sigma_y**2, 0],
                        [0, 0, del_t/2*sigma_z**2, 0, 0,
                           sigma_z**2]])

Q = sym.lambdify(del_t, Q_sym, 'numpy')

#measurement noise
sigma_rho_kwaj = 10E-3 #km
sigma_rho_dot_kwaj = 0.5E-6 #km/s
sigma_rho_dg = 5E-3 #km
sigma_rho_dot_dg = 1E-6 #km/s
sigma_rho_arecibo = 10E-3 #km
sigma_rho_dot_arecibo = 0.5E-6 #km/s
R = np.array([np.diag([sigma_rho_kwaj**2,
   sigma_rho_dot_kwaj**2]),
             np.diag([sigma_rho_dg**2, sigma_rho_dot_dg**2])
                ,
             np.diag([sigma_rho_arecibo**2,
                sigma_rho_dot_arecibo**2])])
```

```
#measurement bias
# bias_kwaj = 40 #m
# bias_dg = -30 #m
# bias_arecibo = 37#m
bias_kwaj = 0
bias_dg = 0
# bias_arecibo = 21.5E-3 #km
bias_arecibo = a_bias
bias = np.array([bias_kwaj, bias_dg, bias_arecibo])

#time
obs_time = obs_mat[:, 1]
obs_id = np.array(obs_mat[:, 0], dtype=int)
obs_range = obs_mat[:, 2]
obs_range_rate = obs_mat[:, 3]

m = obs_time[np.where(obs_time < seconds)].shape[0]
t = obs_time[:m]

obs_mat = obs_mat[:m]


#jacobians
F = A_Matrix()
H = H_tilde_matrix()

#propogation function
f = satellite_motion_phi
pre_fit_res = np.zeros((len(t), 2))
post_fit_res = np.zeros((len(t), 2))

#initializing arrays
y_arr = np.zeros((len(t), 6))
P_bar = np.zeros((len(t), 6*6))
P_hat = np.zeros((len(t), 6*6))
P_ZZ = np.zeros((len(t), 2*2))
phi = np.zeros((len(t), 6*6))

y_arr[0] = y0
P_bar[0] = np.eye(6).reshape(6*6)
P_hat[0] = np.reshape(P_hat_0, 6*6)
phi[0] = np.eye(6).reshape(6*6)
```

```
#compute first observation
y_pred = np.concatenate((r_ECI, v_ECI))
del_t = 0
Q_i = Q(del_t)
phi_0 = np.eye(6)

#propogate covariance
P_hat_k = P_hat[0].reshape(6,6)
# print('P_hat_k', P_hat_k)
P_bar_k_1 = phi_0 @ P_hat_k @ phi_0.T + Q_i

P_bar[0] = np.reshape(P_bar_k_1, 6*6)

#if stations don't have the first observation propagate to
   their first observation
if case == 'C' or case == 'E' or case == 'G':

    del_t = t[0]
    t_eval = np.arange(0, t[0]+60, 60)
    y0 = np.concatenate((y_pred[0:6], phi_0.ravel()))
    prop = solve_ivp(f, [0, del_t], y0, args=(F, JD_UTC + t
       [0]/86400), t_eval=t_eval, rtol=3e-14, atol=1e-16)
    y_pred = prop.y[0:6, -1]
    phi_0 = prop.y[6:, -1].reshape(6, 6)
    phi[0] = phi_0.reshape(6*6)

    #process noise
    Q_i = Q(del_t)

    #recalculate P_bar
    P_bar_k_1 = np.matmul(np.matmul(phi_0, P_hat[0].reshape
       (6,6)), phi_0.T) + Q_i

    P_bar[0] = np.reshape(P_bar_k_1, 6*6)

#find station ECI coordinates
station = stations[obs_id[0]-1]

day = 0

x_p, y_p, del_UT1, LOD = interp_EOP(JD_UTC + t[0]/86400)

ECI_station, ECI_station_dot, _= ECEF2ECI(station, np.array
   ([0,0,0]), None, JD_UTC + t[0]/86400, x_p, y_p, leap_sec
   , del_UT1, LOD)
```

```
#light time correction
y_lt = light_time_correction(JD_UTC + t[0]/86400, y_pred
   [0:3], y_pred[3:6], ECI_station, station)

#compute H_k
H_k = np.array(H(y_pred[0], y_pred[1], y_pred[2], y_pred
   [3], y_pred[4], y_pred[5],
                ECI_station[0], ECI_station[1], ECI_station
                   [2], ECI_station_dot[0], ECI_station_dot
                   [1], ECI_station_dot[2], bias[obs_id
                   [0]-1]))

#compute observation
obs_pred = np.array(range_range_rate(y_lt[0:3], y_lt[3:6],
   ECI_station, ECI_station_dot))

#add bias and compute residuals
if case == 'A ':
    pre_fit_res[0] = np.array([obs_range[0] - (obs_pred[0]
       + bias[obs_id[0]-1]) , 0])
    H_k = np.array([[1, 0], [0, 0]]) @ H_k



elif case == 'B':
    pre_fit_res[0] = np.array([0, obs_range_rate[0] -
       obs_pred[1]])
    H_k = np.array([[0, 0], [0, 1]]) @ H_k

else:
    pre_fit_res[0] = np.array([obs_range[0] - (obs_pred[0]
       + bias[obs_id[0]-1]), obs_range_rate[0] - obs_pred
       [1]])
    # print('bias',  bias.T[obs_id[0]-1])

print('pre_fit_res', pre_fit_res[0])


#compute kalman gain
P_ZZ[0] = (H_k @ P_bar_k_1 @ H_k.T + R[obs_id[0]-1]).
   reshape(2*2)

K = P_bar_k_1 @ H_k.T @ np.linalg.inv(P_ZZ[0].reshape(2,2))
```

```
#state error estimate
del_y = K @ pre_fit_res[0]

#error covariance estimate
P_hat[0] = np.reshape((np.eye(6) - K @ H_k) @ P_bar_k_1 @ (
   np.eye(6) - K @ H_k).T + K @ R[obs_id[0]-1] @ K.T, 6*6)

#state estimate
y_arr[0] = y_pred + del_y

#correct for light time
y_lt = light_time_correction(JD_UTC + t[0]/86400, y_arr
   [0][0:3], y_arr[0][3:6], ECI_station, station)
#compute rho and rho_dot
rho, rho_dot = range_range_rate(y_lt[0:3], y_lt[3:6],
   ECI_station, ECI_station_dot)
# print('rho', rho)
#add bias and compute residuals
post_fit_res[0] = np.array([obs_range[0] - (rho +  bias[
   obs_id[0]-1]) , obs_range_rate[0] - rho_dot])
# print('post_fit_res', post_fit_res[0])


day_old = 1
old_obs_id = obs_id[0]
print('Running...')
#run EKF
for k in range(1, len(t)):
    del_t = t[k] - t[k-1]
    # print('t[k-1]', t[k-1])
    # print('t[k]', t[k])

    day = int(((JD_UTC + t[k]/86400) - JD_UTC_day))
    new_obs_id = obs_id[k]
    if new_obs_id != old_obs_id:
        # print('station: ', new_obs_id)
        old_obs_id = new_obs_id
    if day != day_old:
        print('Day: ', day)
        day_old = day
    phi_k = np.eye(6)

    #initial conditions for propogation
    y0 = np.concatenate((y_arr[k-1][0:6], phi_k.ravel()))
    t_eval = np.arange(0, del_t+60, 60)
```

```
#propogate state

prop = solve_ivp(f, [0, del_t], y0, args=(F, JD_UTC + t
   [k-1]/86400), t_eval=t_eval, rtol=3e-14, atol=1e-16)

y_pred = prop.y[0:6, -1]

phi_k = prop.y[6:, -1].reshape(6, 6)

phi[k] = phi_k.reshape(6*6)

#process noise
Q_i = Q(del_t)

#propogate covariance
P_hat_k = P_hat[k-1].reshape(6,6)

P_bar_k_1 = phi_k @ P_hat_k @ phi_k.T + Q_i
P_bar[k] = np.reshape(P_bar_k_1, 6*6)

#calculate station ECI coordinates
station = stations[obs_id[k]-1]

x_p, y_p, del_UT1, LOD = interp_EOP(JD_UTC + t[k
   ]/86400)
ECI_station, ECI_station_dot, _= ECEF2ECI(station, np.
   array([0,0,0]), None, JD_UTC + t[k]/86400, x_p, y_p,
    leap_sec, del_UT1, LOD)

#light time correction
y_lt = light_time_correction(JD_UTC + t[k]/86400,
   y_pred[0:3], y_pred[3:6], ECI_station, station)
# print('light_time difference', np.linalg.norm(y_lt
   [0:3] - y_pred[0:3]))

#compute H_k
H_k = np.array(H(y_pred[0], y_pred[1], y_pred[2],
   y_pred[3], y_pred[4], y_pred[5], ECI_station[0],
   ECI_station[1],
                ECI_station[2], ECI_station_dot[0],
                   ECI_station_dot[1], ECI_station_dot
                   [2], bias[obs_id[k]-1]))

#compute observation
```

```
obs_pred = np.array(range_range_rate(y_lt[0:3], y_lt
   [3:6], ECI_station, ECI_station_dot))

#add bias and compute residuals
if case == 'A':
    pre_fit_res[k] = np.array([obs_range[k] - (obs_pred
       [0] + bias[obs_id[k]-1]) , 0])
    H_k = np.array([[1, 0], [0, 0]]) @ H_k

elif case == 'B':
    pre_fit_res[k] = np.array([0, obs_range_rate[k] -
       obs_pred[1]])
    H_k = np.array([[0, 0], [0, 1]]) @ H_k

else:
    pre_fit_res[k] = np.array([obs_range[k] - (obs_pred
       [0] +  bias[obs_id[k]-1]) , obs_range_rate[k] -
       obs_pred[1]])

# print('bias', bias[obs_id[k]-1])
print('pre_fit_res', pre_fit_res[k])

#compute kalman gain
K = P_bar_k_1 @ H_k.T @ np.linalg.inv(H_k @ P_bar_k_1 @
    H_k.T + R[obs_id[k]-1])

#innovation covariance
P_ZZ[k] = (H_k @ P_bar_k_1 @ H_k.T + R[obs_id[k]-1]).
   reshape(2*2)

#state error estimate
del_y = K @ pre_fit_res[k]

#error covariance estimate
P_hat[k] = np.reshape((np.eye(6) - K @ H_k) @ P_bar_k_1
    @ (np.eye(6) - K @ H_k).T + K @ R[obs_id[k]-1] @ K.
   T, 6*6)

#update state estimate
y_arr[k] = y_pred + del_y
y_hat_lt = light_time_correction(JD_UTC + t[k]/86400,
   y_arr[k][0:3], y_arr[k][3:6], ECI_station, station)
rho, rho_dot = range_range_rate(y_hat_lt[0:3], y_hat_lt
   [3:6], ECI_station, ECI_station_dot)
```

```
            post_fit_res[k] = np.array([obs_range[k] - (rho + bias[
                obs_id[k]-1]), obs_range_rate[k] - rho_dot])
            # print('post_fit_res',post_fit_res[k])


    print('EKF complete')
    return y_arr, P_hat, P_bar, P_ZZ, obs_mat, Q, pre_fit_res,
        post_fit_res, phi



def plot_ellipses(x_mean, covariance, sig, ax, case, color):
    # Mean and Covariance
    P_xx = covariance

    eigenval, eigenvec = np.linalg.eig(P_xx)

    # Get the index of the largest eigenvector
    max_evc_ind_c = np.argmax(eigenval)
    max_evc = eigenvec[:, max_evc_ind_c]

    # Get the largest eigenvalue
    max_evl = np.max(eigenval)
    min_evl = np.max(eigenval)


    # Calculate the angle between the x-axis and the largest
        eigenvector
    phi = np.arctan2(max_evc[1], max_evc[0])


    theta_grid = np.linspace(0, 2*np.pi, 100)

    X0  = x_mean[0]
    Y0  = x_mean[1]

    # Find semi-major and semi-minor axis for 1,2,3sigma

    a = np.sqrt(sig*max_evl)      # Semi-major axis
    b = np.sqrt(sig*min_evl)      # Semi- minor axis


    # the ellipse in x and y coordinates
    ellipse_x_r  = a*np.cos(theta_grid)
```

```
    ellipse_y_r  = b*np.sin(theta_grid)

    # Define a rotation matrix (x',y') to (x,y)
    R = np.array([[np.cos(phi), -np.sin(phi)],
                  [np.sin(phi), np.cos(phi)]])

    # let's rotate the ellipse to some angle phi
    r_1 = np.zeros((len(theta_grid), 2))

    for j in range(len(theta_grid)):
        r_1[j,:] = R @ np.array([ellipse_x_r[j],ellipse_y_r[j
           ]])


    X1 = r_1.T.reshape(2,len(theta_grid))

    ax.plot(X0 + X1[0,:], Y0 + X1[1,:], linestyle='--',
       linewidth=1, label='Case ' + case, color=color)


    return X1


def eci_to_ric(r_ref, v_ref, r, cov):

    '''
    Transforms a state vector from ECI to RIC frame
    Inputs:
        r_ref: reference position vector
        v_ref: reference velocity vector
        r: position vector to be transformed
        cov: covariance matrix of position vector to be
           transformed

    Outputs:
        r_ric: position vector in RIC frame
        cov_ric: covariance matrix of position vector in RIC
           frame
    '''


   # Calculate the rotation matrix gamma
    u_R = r_ref / np.linalg.norm(r_ref) # Calculate unit vector
        in radial direction
```

```
    u_N = np.cross(r_ref, v_ref) / np.linalg.norm(np.cross(
        r_ref, v_ref)) # Calculate unit vector in normal
        direction
    u_T = np.cross(u_N, u_R) # Calculate unit vector in
        tangential direction

    gamma = np.vstack((u_R, u_T, u_N)) # Create rotation matrix
         from unit vectors

    # Transform position and covariance matrix to radial-
        inertial frame
    r_ric = gamma @ (r_ref - r) # Transform position vector
    cov_ric = gamma @ cov @ gamma.T # Transform covariance
        matrix

    return r_ric, cov_ric # Return transformed position and
        covariance matrix

def plot_residuals(case, obs_mat, residuals, res_type, t, P_ZZ,
    Ps, Psv, Qs):
    '''
    Saves the plot of the residuals for each station

    Inputs:
        case: string of the case letter
        obs_mat: data of the observations
        column_names: list of the column names for the
            observations
        t: time array
        P_ZZ: covariance matrix
        m: number of observations


    '''
    #postfit RMS
    # df_calc = pd.read_csv('range_rate_case' + case + '.csv',
        names=column_names)
    #Comparing Pre-fit residuals
    fig, ax = plt.subplots(2, 1, figsize=(10, 10))

    if case == 'A' or case == 'B' or case == 'C' or case == 'F'
        or case == 'G':

        r_residuals_kwaj = residuals[np.where(obs_mat[:, 0] ==
            1)[0], 0]
```

```
        rr_residuals_kwaj = residuals[np.where(obs_mat[:, 0] ==
            1)[0], 1]

        RMS_Kwaj_r = np.sqrt(np.mean((r_residuals_kwaj)**2))
        RMS_Kwaj_rr = np.sqrt(np.mean((rr_residuals_kwaj)**2))


        t_kwaj = obs_mat[np.where(obs_mat[:, 0] == 1)[0], 1]

        print("Kwaj Range RMS:", RMS_Kwaj_r, "[km]")
        print("Kwaj Range Rate RMS:", RMS_Kwaj_rr,"[km/s]")

        ax[0].scatter(t_kwaj, r_residuals_kwaj, label='Kwaj
           Range Residuals')
        ax[1].scatter(t_kwaj, rr_residuals_kwaj, label='Kwaj
           Range Rate Residuals')

    if case == 'A' or case == 'B' or case == 'D' or case == 'F'
        or case == 'G':


        r_residuals_dg = residuals[np.where(obs_mat[:, 0] == 2)
           [0], 0]
        rr_residuals_dg = residuals[np.where(obs_mat[:, 0] ==
           2)[0], 1]

        RMS_Diego_r = np.sqrt(np.mean((r_residuals_dg)**2))
        RMS_Diego_rr = np.sqrt(np.mean((rr_residuals_dg)**2))

        t_diego = obs_mat[np.where(obs_mat[:, 0] == 2)[0], 1]

        print("Diego Range RMS:", RMS_Diego_r, "[km]")
        print("Diego Range Rate RMS:", RMS_Diego_rr,"[km/s]")

        ax[0].scatter(t_diego, r_residuals_dg, label='Diego
           Range Residuals')
        ax[1].scatter(t_diego, rr_residuals_dg, label='Diego
           Range Rate Residuals')


    if case == 'A' or case == 'B' or case == 'E' or case == 'F'
        or case == 'G':
```

```
    r_residuals_arecibo = residuals[np.where(obs_mat[:, 0]
       == 3)[0], 0]
    rr_residuals_arecibo = residuals[np.where(obs_mat[:, 0]
        == 3)[0], 1]

    RMS_Arecibo_r = np.sqrt(np.mean((r_residuals_arecibo)
       **2))
    RMS_Arecibo_rr = np.sqrt(np.mean((rr_residuals_arecibo)
       **2))

    t_arecibo = obs_mat[np.where(obs_mat[:, 0] == 3)[0], 1]

    print("Arecibo Range RMS:", RMS_Arecibo_r, "[km]")
    print("Arecibo Range Rate RMS:", RMS_Arecibo_rr,"[km/s
       ]")

    ax[0].scatter(t_arecibo, r_residuals_arecibo, label='
       Arecibo Range Residuals')
    ax[1].scatter(t_arecibo, rr_residuals_arecibo, label='
       Arecibo Range Rate Residuals')


three_sigma_pos = np.array([np.sqrt(P_ZZ[i][0])*3 for i in
   range(len(t))])
three_sigma_vel = np.array([np.sqrt(P_ZZ[i][3])*3 for i in
   range(len(t))])

# print(three_sigma_pos)
ax[0].plot(t, three_sigma_pos, label='+3$\sigma$')
ax[0].plot(t, -three_sigma_pos, label='-3$\sigma$')
ax[1].plot(t, three_sigma_vel, label='+3$\sigma$')
ax[1].plot(t, -three_sigma_vel, label='-3$\sigma$')



ax[0].set_title('Range Residuals')
ax[0].set_xlabel('Time [s]')
ax[0].set_ylabel('Range Residuals [km]')
ax[0].set_ylim(-0.05, 0.05)
ax[0].legend()


ax[1].set_title('Range Rate Residuals')
ax[1].set_xlabel('Time [s]')
ax[1].set_ylabel('Range Rate Residuals [km/s]')
```

```
ax[1].set_ylim(-0.00002, 0.00002)
ax[1].legend()
# plt.show()
fig.savefig(res_type + 'residuals_case' + case+ str(Qs) +
   str(Ps) + str(Psv) +'.png', dpi=300)


---------------------------------------

def main():


cases = ['A', 'B', 'C', 'D', 'E', 'F', 'G']


#initial state
r_ECI = np.array([6984.45711518852, 1612.2547582643,
   13.0925904314402]) #km

v_ECI = np.array([-1.67667852227336, 7.26143715396544,
   0.259889857225218]) #km/s


best_case_pos = np.array([445.825159257239,
   -7100.15487544519, -183.846642900717]) #km
best_case_vel = np.array([7.48607847654087,
   0.468696626655197, 0.177504090224814]) #km/s
#station locations
stations = np.array([[-6143.584, 1364.250, 1033.743],
                     [1907.295, 6030.810, -817.119],
                     [2390.310, -5564.341 , 1994.578]]) #km

JD_UTC = gregorian_to_jd(2018, 3, 23, 8, 55, 3)

Qs= 1E-11
Ps = 1E0
Psv = 1E-3
a_bias = 20E-3

solar = 'SC'
drag = 'DF_old'
print('Solar:', solar, 'Drag:', drag)
print('Qs:', Qs, 'Ps:', Ps, 'Psv:', Psv)

seconds = 7*86400
```

```
#switch for observations
for case in cases:

    obs_mat_3d = loadmat('LEO_DATA_Apparent_3Days.mat')
    obs_mat_4_6 = loadmat('LEO_DATA_Apparent_Days4-6.mat')
    obs_mat_4_6['LEO_DATA_Apparent'][:,1] += 3*86400
    obs_mat = np.concatenate((obs_mat_3d['LEO_DATA_Apparent
       '], obs_mat_4_6['LEO_DATA_Apparent']), axis=0)


    y_arr, P_hat, P_bar, P_ZZ, obs_mat, Q, pre_fit_res,
       post_fit_res, phi = EKF(case, r_ECI, v_ECI, stations
       , JD_UTC, seconds, Ps, Psv, Qs, a_bias)

    # print(obs_mat[:, 1])
    t = obs_mat[:, 1]

    #plot residuals
    plot_residuals(case, obs_mat, pre_fit_res, 'pre', t,
       P_ZZ, Ps, Psv, Qs)
    plot_residuals(case, obs_mat, post_fit_res, 'post', t,
       P_ZZ, Ps, Psv, Qs)

    print('propogating state...')
    #propagate state
    del_t = seconds-t[-1]
    phi = np.eye(6)
    F = A_Matrix()
    t_eval = np.arange(t[-1], seconds + 60, 60)
    y0 = np.concatenate((y_arr[-1][0:6], phi.ravel()))
    sol_final = solve_ivp(satellite_motion_phi, [t[-1],
       seconds], y0, t_eval=t_eval, args=(F, JD_UTC), rtol
       =3E-14, atol=1E-16)

    y_final = sol_final.y[0:6, -1]
    print('y_final:', y_final)
    if seconds == 86400:
        print('residuals:', best_case_pos[0:3] - y_final
           [0:3])
        print('distance:', np.linalg.norm(best_case_pos -
           y_final[0:3]))
    elif seconds == 7*86400:

        print('residuals:', best_case_pos - y_final[0:3])
```

```
            print('distance:', np.linalg.norm(best_case_pos -
                y_final[0:3]))
        phi_k = sol_final.y[6:, -1].reshape(6, 6)


        #process noise
        Q_k = Q(del_t)

        #propogate covariance
        P_hat_k = P_hat[-1].reshape(6,6)
        P_bar_k_1 = phi_k @ P_hat_k @ phi_k.T + Q_k
        cov = P_bar_k_1
        print('covariance:', cov)
        print('writing to mat file...')
        #save to .mat file
        savemat('smithT_pos_vel_case' + case + str(Qs) + str(Ps
            ) + str(Psv) + solar + drag + '.mat', {'
            smithT_pos_vel_case' + case : y_final}, oned_as='
            column')
        savemat('smithT_poscov_case' + case + str(Qs) + str(Ps)
             + str(Psv) + solar + drag +'.mat', {'
            smithT_pos_case' + case : y_final[0:3], '
            smithT_poscov_case' + case : cov[0:3, 0:3]}, oned_as
            ='column')


if __name__ == "__main__":
    main()

-------------------------------------------------------------

import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp
from scipy.io import savemat
from fp_functions import *

def main():
    fig, ax = plt.subplots(1, 3, figsize=(18, 5))
    ax[0].plot(0, 0, 'o', label='Truth')
    ax[1].plot(0, 0, 'o', label='Truth')
    ax[2].plot(0, 0, 'o', label='Truth')

    cases = ['A','B', 'C', 'D', 'E', 'F', 'G']
    colors = ['r', 'b', 'g', 'c', 'm', 'y', 'k']
```

```
best_case = loadmat('smithT_best.mat')
best_pos = best_case['smithT_pos_vel_caseF'].T[0][0:3]
best_vel = best_case['smithT_pos_vel_caseF'].T[0][3:6]
case_pos_cov = loadmat('smithT_050923.mat')

for i in range(len(cases)):
    y_final = case_pos_cov['smithT_pos_case' + cases[i]]
    cov = case_pos_cov['smithT_poscov_case' + cases[i]]
    y_final = y_final.T[0]

    print('Case ' + cases[i] + ' residuals:', best_pos[0:3]
        - y_final[0:3])
    print('distance:', np.linalg.norm(best_pos[0:3] -
       y_final[0:3]))
    RIC_pred, RIC_cov = eci_to_ric(best_pos, best_vel,
       y_final, cov[0:3, 0:3].T)


    radial = RIC_pred[0]
    in_track = RIC_pred[1]
    cross_track = RIC_pred[2]


    ax[0].plot(radial, in_track, 'o', color=colors[i])
    plot_ellipses(RIC_pred[0:2], RIC_cov[0:2, 0:2],3, ax
       [0], cases[i], colors[i])

    ax[1].plot(radial, cross_track, 'o', color=colors[i])

    plot_ellipses([RIC_pred[0], RIC_pred[2]], np.array([[
       RIC_cov[0, 0], RIC_cov[0, 2]],[RIC_cov[2, 0],
       RIC_cov[2, 2]]]),3, ax[1], cases[i], colors[i])

    ax[2].plot(in_track, cross_track, 'o', color=colors[i])

    plot_ellipses(RIC_pred[1:3], RIC_cov[1:3, 1:3],3, ax
       [2], cases[i], colors[i])


    ax[0].set_xlabel('Radial (km)')
    ax[0].set_ylabel('In-track (km)')
    ax[0].set_title('Radial vs In-track')
    ax[0].legend(loc = 'upper left')
    ax[0].grid()
```

```
        ax[1].set_xlabel('Radial (km)')
        ax[1].set_ylabel('Cross-track (km)')
        ax[1].set_title('Radial vs Cross-track')
        ax[1].legend(loc = 'upper left')
        ax[1].grid()


        ax[2].set_xlabel('In-track (km)')
        ax[2].set_ylabel('Cross-track (km)')
        ax[2].set_title('In-track vs Cross-track')
        ax[2].legend(loc = 'upper left')
        ax[2].grid()

    plt.show()



if __name__ == "__main__":
    main()
```