Orbit Determination Project Phase 1

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

20 April 2023

Background

For this final project, we are putting together everything we have learned over the semester toward determining the orbit of a satellite given its initial state and observation data. For Phase 1, only **one day** of apparent range and range-rate data is processed. I use an Extended Kalman Filter (EKF) to update predictions based on collected measurement data, and propagate with perturbations from J2, a cannon-ball model for the drag and solar radiation pressure, and luni-solar third-body effects. Figure 1 shows the starting initial condition for the satellite for the final project, and the satellite configuration and properties are given in Figures 2 and 3.

	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.

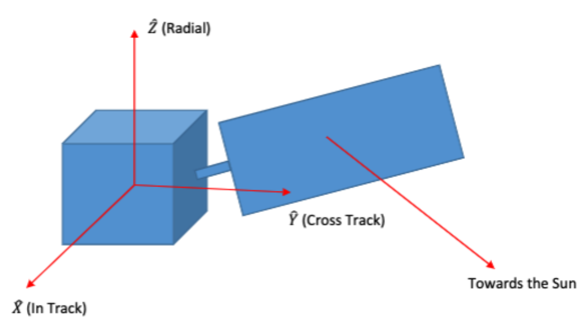

Figure 2: Satellite configuration graphic.

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.

Range and Range-rate residuals

I calculated Range and Range-Rate residuals along with Root Mean Square (RMS) error for the post-fit predicted range and range-rate for the ground station associated with the provided observation data. Five cases were tested as listed below. The locations of the ground stations are given in Figure 4, each with a known measurement noise variance given in Figure 5. The post-fit range residuals are shown in Figure 7 and the range-rate residuals in Figure **??**.

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.

Number	Description	Range σ (m)	Range Rate σ (mm/s)
1	Kwajalein	10	0.5
2	Diego Garcia	5	1
3	Arecibo	10	0.5

Figure 5: Ground station measurement noise variances.

The cases are defined in the project description and given in Figure 6. Because this first phase only propagates for 1 day, only cases (a)-(e) are run.

(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 6: Case descriptions for project deliverables.

The RMS results for each of the cases are presented below:
Case (a):

$$\begin{aligned}
\text{Kwajalein range } RMS &= 9161.23881026789 \text{ km} \\
\text{Diego Garcia range } RMS &= 9060.32352168923 \text{ km} \\
\text{Arecibo range } RMS &= 8176.510487416482 \text{ km} \\
\text{Kwajalein range-rate } RMS &= 5.378535387824855 \text{ km/s} \\
\text{Diego Garcia range-rate } RMS &= 5.450604214663162 \text{ km/s} \\
\text{Arecibo range-rate } RMS &= 4.757038915949603s \text{ km/s}
\end{aligned}$$

Case (b):

$$\begin{aligned}
\text{Kwajalein range } RMS &= 0.6746969797980299 \text{ km} \\
\text{Diego Garcia range } RMS &= 0.933214903970481 \text{ km} \\
\text{Arecibo range } RMS &= 0.41231188408719194 \text{ km} \\
\text{Kwajalein range-rate } RMS &= 0.0007001591808941003 \text{ km/s} \\
\text{Diego Garcia range-rate } RMS &= 0.001272319610468948 \text{ km/s} \\
\text{Arecibo range-rate } RMS &= 0.001511919055315043 \text{ km/s}
\end{aligned}$$

Case (c):

$$\begin{aligned}
\text{Kwajalein range } RMS &= 2.5457893645216707 \text{ km} \\
\text{Kwajalein range-rate } RMS &= 0.0017258142602876534 \text{ km/s}
\end{aligned}$$

Case (d):

$$\begin{aligned}
\text{Diego Garcia range } RMS &= 1.897340333374553 \text{ km} \\
\text{Diego Garcia range-rate } RMS &= 0.0020787286896378223 \text{ km/s}
\end{aligned}$$

Case (e):

$$\text{Arecibo range } RMS = 13520.05232278831 \text{ km}$$
$$\text{Arecibo range-rate } RMS = 2.550121872450854 \text{ km/s}$$

I also plotted the Post-fit range and range-rate residuals with 3σ bounds for each of the cases. An example from case (b) is shown below. As can be seen, the 3σ bounds do not represent the data properly. This is most likely due to some error in my 3σ calculation.

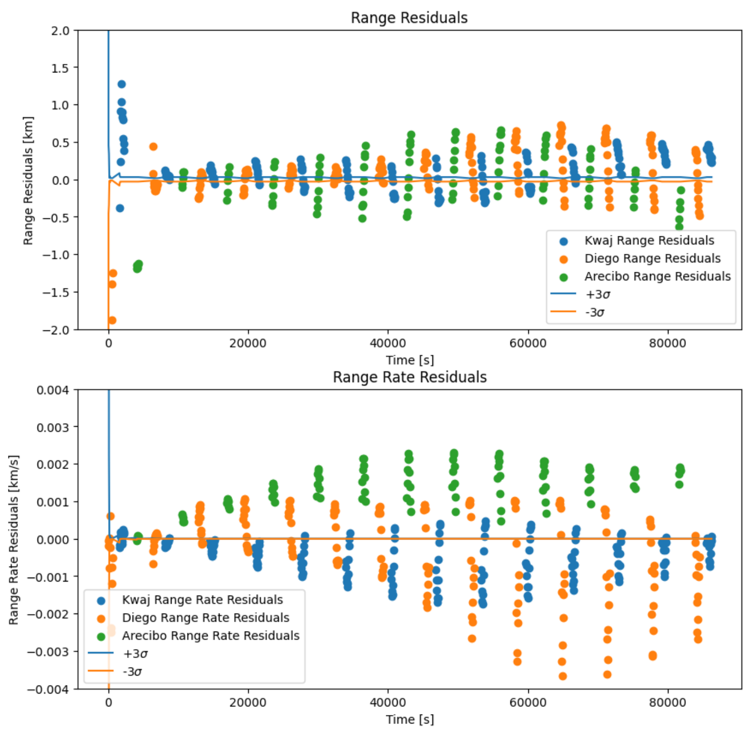

Figure 7: Case B Post-fit range and range-rate residuals over time.

Position and Covariance estimate after 24 hours

I also calculated the final position and position covariance matrix in ECI after 86400s from the starting Epoch for each case. Using just Range-rate data provided the best results as compared to the true position.

$$\text{Final Epoch} = 86400 \text{ s}$$

$$
\begin{aligned}
\text{True Position at Final Epoch} &= \begin{bmatrix}-6.3302e+033.3068e+031.2774e+02\end{bmatrix} \text{ km} \\
\text{Predicted Position at Final Epoch} &= \begin{bmatrix}-6.3303e+03 & ,3.3072e+03 & 1.2831e+02\end{bmatrix} \text{km} \\
\text{Position covariance at Final Epoch} &= \begin{bmatrix} 3.2538e-10 & 4.6449e-10 & -6.0585e-11 \\ 4.6449e-10 & 8.9987e-10 & 1.7660e-10 \\ -6.0585e-11 & 1.7660e-10 & 1.5192e-09 \end{bmatrix}
\end{aligned}
$$

The Way Ahead

My current model only uses the most basic of representations for the space craft. This results in significant error when propagating over long periods without measurement correction. I foresee this causing further error when I move to the 6 day tests. My next step is to implement the 20x20 EGM96 gravity field and the box wing model for drag and Solar Radiation Pressure before moving onto the 6 day implementation.

APPENDIX: Python Code

```
#Initialize Extended Kalman Filter

phi = np.eye(6)
y0 = np.concatenate((r_ECI, v_ECI))

#process noise
sigma_x = 1E-10
sigma_y = 1E-10
sigma_z = 1E-10

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 = 10 #m
sigma_rho_dot_kwaj = 0.0005 #m/s
sigma_rho_dg = 5 #m
sigma_rho_dot_dg = 0.001 #m/s
sigma_rho_arecibo = 10 #m
sigma_rho_dot_arecibo = 0.0005 #m/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 = 5 #m
```

```
bias_dg = 10 #m
bias_arecibo = 5 #m
bias = np.array([[bias_kwaj, bias_dg, bias_arecibo], [0, 0,
   0]])
print(bias.T[0])
#time
seconds = 86400
m = obs_df['time'].index[obs_time > seconds].values[0]
# print(m)
t = obs_df['time'][0:m].values
#initial state
y = y0

#initial covariance
sigma_x = 1E4 #m
sigma_y = 1E4 #m
sigma_z = 1E4 #m
sigma_vx = 1E1 #m/s
sigma_vy = 1E1 #m/s
sigma_vz = 1E1 #m/s
sigma_C_D = 1e-6

#initial covariance

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

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

#propogation function
f = satellite_motion_phi

#measurement function
h = range_range_rate

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

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

#propogate covariance
P_hat_k = P_hat[0].reshape(6,6)
P_bar_k_1 = np.matmul(np.matmul(phi, P_hat_k), phi.T) + Q_i
# print(P_bar_k_1)
P_bar[0] = np.reshape(P_bar_k_1, 6*6)

#if stations don't have the first observation propogate to
   their first observation
if case == 'C' or case == 'E':
    day = 0
    del_t = t[0]
    t_eval = np.arange(0, t[0] + 60, 60)
    y0 = np.concatenate((y_pred[0:6], phi.ravel()))
    prop = solve_ivp(f, [0, del_t+60], y0, args=(F, JD_UTC, day
       ), t_eval=t_eval, rtol=3E-14, atol=1E-16)
    y_pred = prop.y[0:6, -1]
    phi = prop.y[6:, -1].reshape(6, 6)

    #process noise
    Q_i = Q(del_t)

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

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

#calculate kalman gain
station = stations[obs_id[0]-1]

day = 0

ECI_station, ECI_station_dot, _ = ECEF2ECI(station, np.array
   ([0,0,0]), None, JD_UTC+t[0]/86400, x_p[day], y_p[day],
   leap_sec, del_UT1[day], LOD[day])
print('y_pred', y_pred[0:3])
#light time correction
```

```
y_lt = light_time_correction(JD_UTC + t[0]/86400, y_pred[0:3],
   y_pred[3:6], station)
print('y_lt', y_lt[0:3])
#compute H_k
H_k = np.array(H(y_lt[0], y_lt[1], y_lt[2], y_lt[3], y_lt[4],
   y_lt[5],
                 ECI_station[0], ECI_station[1], ECI_station
                    [2], ECI_station_dot[0], ECI_station_dot
                    [1], ECI_station_dot[2], 1.88))

#compute observation
obs_pred = np.array(range_range_rate(JD_UTC + t[0]/86400, y_lt
   [0:3], y_lt[3:6], station))


if case == 'A':
    b = np.array([obs_range[0]*1000 - obs_pred[0], 0])


elif case == 'B':
    b = np.array([0, obs_range_rate[0]*1000 - obs_pred[1]])

else:
    b = np.array([obs_range[0]*1000, obs_range_rate[0]*1000])
      - obs_pred
    # - bias.T[obs_id[0]-1]

print('b', b/1000)
#compute kalman gain
P_ZZ[0] = (np.matmul(np.matmul(H_k, P_bar_k_1), H_k.T)+ R[
   obs_id[0]-1]).reshape(2*2)

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

#state error estimate
del_y = np.matmul(K, b)


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

#state estimate
```

```python
y_arr[0] = y_pred + del_y


#run EKF


for k in range(1, len(t)):
    print(t[k])
    del_t = t[k] - t[k-1]

    day = int((JD_UTC + t[k]/86400) % JD_UTC)

    phi = np.eye(6)

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

    #propogate state
    prop = solve_ivp(f, [t[k-1], t[k]], y0, args=(F, JD_UTC + t
       [k-1]/86400, day), t_eval=t_eval, rtol=3E-14, atol=1E
       -16)

    y_pred = prop.y[0:6, -1]
    phi = prop.y[6:, -1].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 = np.matmul(np.matmul(phi, P_hat_k), phi.T) + Q_i

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

    #calculate kalman gain
    station = stations[obs_id[k]-1]

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

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

H_k = np.array(H(y_lt[0], y_lt[1], y_lt[2], y_lt[3], y_lt
   [4], y_lt[5], ECI_station[0], ECI_station[1],
                  ECI_station[2], ECI_station_dot[0],
                     ECI_station_dot[1], ECI_station_dot[2],
                      1.88))


obs_pred = np.array(range_range_rate(JD_UTC + t[k]/86400,
   y_lt[0:3], y_lt[3:6], station))

if case == 'A':
    b = np.array([obs_range[k]*1000 - bias.T[obs_id[k
       ]-1][0] -  obs_pred[0], 0])

elif case == 'B':
    b = np.array([0, obs_range_rate[k]*1000 - obs_pred[1]])

else:
    b = np.array([obs_range[k]*1000, obs_range_rate[k
       ]*1000])  - obs_pred
    # - bias.T[obs_id[k]-1]

P_ZZ[k] = (np.matmul(np.matmul(H_k, P_bar_k_1), H_k.T)+ R[
   obs_id[k]-1]).reshape(2*2)

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




#state error estimate
del_y = np.matmul(K, b)


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

```
    # np.set_printoptions(precision=25, suppress=False)

    #state estimate

    y_arr[k] = y_pred + del_y



\\functions


leap_sec = 37 #s
x_p = np.array([20.816, 22.156, 23.439, 24.368, 25.676, 26.952,
    28.108])/1000 #arcsec
y_p = np.array([381.008, 382.613, 384.264, 385.509, 386.420,
   387.394, 388.997])/1000 #arcsec
del_UT1 = np.array([144.0585, 143.1048, 142.2335, 141.3570,
   140.4078, 139.3324, 138.1510]) #ms
LOD = np.array([1.0293,0.9174, 0.8401, 0.8810, 1.0141, 1.1555,
   1.2568])/1000 #s


#Rotation Martices

def R1(theta):
    return np.array([[1, 0, 0], \
                     [0, np.cos(theta), np.sin(theta)], \
                     [0, -np.sin(theta), np.cos(theta)]])

def R2(theta):
    return np.array([[np.cos(theta), 0, -np.sin(theta)],\
                    [0, 1, 0], \
                    [np.sin(theta), 0, np.cos(theta)]])

def R3(theta):
    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 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)


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 = np.matmul(np.matmul(R3(C_a), R2(-theta_a)), R3(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 = np.matmul(R1(y_p), R2(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
'''

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

#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 = R3(-GAST)

    return R

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 R1, R3, R1
    N = np.matmul(np.matmul(R1(-epsilon_m), R3(del_psi)), R1(
       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

    del_UT1 /=1000

    #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 R1, R3, R1
    N = NutationMatrix(T_TT)
    # r_mod = np.matmul(N, r_TOD)

    #precession matrix
    P = PrecessionMatrix(T_TT)

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

    r_ECEF = np.matmul(np.matmul(np.matmul(np.matmul(W.T, R.T),
        N.T), P.T), r_ECI)
    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

del_UT1 /=1000

#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

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

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

#nutation matrix R1, R3, R1
N = NutationMatrix(T_TT)

#precession matrix
P = PrecessionMatrix(T_TT)

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


# print('w_rate', w_rate)
# print('P', P)
# print('N', N)
# print('R', R)
# print('W', W)
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))))
    if a_ECEF is not None:
        a_ECI = np.matmul(np.matmul(np.matmul(P, N), R), (np.
           matmul(W, a_ECEF)))

    return r_ECI, v_ECI, a_ECI

def sun_position_vector(JD_UTC, del_UT1, leap_sec):
    '''
    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 = 149597870.691*1000 # Astronomical unit [m]
    # Time variables

    JD2000 = 2451545.0

    del_UT1 /=1000

    #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 Sun
    l = (280.460 + 36000.771285 * T_UT1) %360
```

```
    # Mean anomaly of the Sun
    M = (357.528 + 35999.050957 * T_UT1) %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_UT1

    #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 R1, R3, R1
    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, del_UT1, leap_sec):
    '''
    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*1000 #m
    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 R1, R3, R1
    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, day):
    '''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_UTC: Julian date in UTC

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

```
'''
mu = 398600.4415*1000**3 #m^3/s^2
J_2 = 0.00108248
phi = R[6:].reshape(6, 6)
R_earth = 6378.1363*1000 # m
r = R[0:3]
r_dot = R[3:6]
x, y, z = R[0:3]
x_dot, y_dot, z_dot = R[3:6]
JD += t/86400

#J2
dUdx = -1.0*mu*x*(-J_2*R_earth**2*(1.5*z**2/(x**2 + y**2 +
   z**2)**1.0 - 0.5)/(x**2 + y**2 + z**2)**1.0 + 1)/(x**2 +
    y**2 + z**2)**1.5 \
    + mu*(3.0*J_2*R_earth**2*x*z**2/(x**2 + y**2 + z**2)
       **3.0 + 2.0*J_2*R_earth**2*x*(1.5*z**2/(x**2 + y**2
       + z**2)**1.0 - 0.5)/(x**2 \
    + y**2 + z**2)**2.0)/(x**2 + y**2 + z**2)**0.5
dUdy = -1.0*mu*y*(-J_2*R_earth**2*(1.5*z**2/(x**2 + y**2 +
   z**2)**1.0 - 0.5)/(x**2 + y**2 + z**2)**1.0 + 1)/(x**2 +
    y**2 + z**2)**1.5 \
    + mu*(3.0*J_2*R_earth**2*y*z**2/(x**2 + y**2 + z**2)
       **3.0 + 2.0*J_2*R_earth**2*y*(1.5*z**2/(x**2 + y**2
       + z**2)**1.0 - 0.5)/(x**2 \
    + y**2 + z**2)**2.0)/(x**2 + y**2 + z**2)**0.5
dUdz = -1.0*mu*z*(-J_2*R_earth**2*(1.5*z**2/(x**2 + y**2 +
   z**2)**1.0 - 0.5)/(x**2 + y**2 + z**2)**1.0 + 1)/(x**2 +
    y**2 + z**2)**1.5\
    + mu*(2.0*J_2*R_earth**2*z*(1.5*z**2/(x**2 + y**2 + z
       **2)**1.0 - 0.5)/(x**2 + y**2 + z**2)**2.0 - J_2*
       R_earth**2*(-3.0*z**3/(x**2 \
        + y**2 + z**2)**2.0 + 3.0*z/(x**2 + y**2 + z**2)
           **1.0)/(x**2 + y**2 + z**2)**1.0)/(x**2 + y**2 +
            z**2)**0.5

#drag
A_Cross = 6
C_D = 1.88
r_ddot_drag = a_drag(C_D, r, r_dot, A_Cross)

#solar
leap_sec = 37
# d_UT1 = 196.5014/1000 #[s]
r_sun = np.zeros((1, 3))
```

```
    r_sun[0] = sun_position_vector(JD, leap_sec, del_UT1[day])
    C_s = 0.04
    C_d = 0.04
    A_Cross_sol = 15
    r_ddot_sol = a_solar(r, r_sun[0], C_s, C_d, A_Cross_sol)


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


    #total acceleration
    r_ddot = np.array([dUdx, dUdy, dUdz]) + 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 = np.matmul(A_1, phi)

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

    return dydt

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

    '''
    mu = 398600.4415*10**9 #m^3/s^2
    J_2 = 0.00108248
    R_earth = 6378.1363*1000 # m
    r = R[0:3]
    r_dot = R[3:6]
    x, y, z = R[0:3]
    JD_UTC += t/86400

    #J2
```

```
dUdx = -1.0*mu*x*(-J_2*R_earth**2*(1.5*z**2/(x**2 + y**2 +
   z**2)**1.0 - 0.5)/(x**2 + y**2 + z**2)**1.0 + 1)/(x**2 +
    y**2 + z**2)**1.5 \
     + mu*(3.0*J_2*R_earth**2*x*z**2/(x**2 + y**2 + z**2)
        **3.0 + 2.0*J_2*R_earth**2*x*(1.5*z**2/(x**2 + y**2
        + z**2)**1.0 - 0.5)/(x**2 \
     + y**2 + z**2)**2.0)/(x**2 + y**2 + z**2)**0.5
dUdy = -1.0*mu*y*(-J_2*R_earth**2*(1.5*z**2/(x**2 + y**2 +
   z**2)**1.0 - 0.5)/(x**2 + y**2 + z**2)**1.0 + 1)/(x**2 +
    y**2 + z**2)**1.5 \
     + mu*(3.0*J_2*R_earth**2*y*z**2/(x**2 + y**2 + z**2)
        **3.0 + 2.0*J_2*R_earth**2*y*(1.5*z**2/(x**2 + y**2
        + z**2)**1.0 - 0.5)/(x**2 \
     + y**2 + z**2)**2.0)/(x**2 + y**2 + z**2)**0.5
dUdz = -1.0*mu*z*(-J_2*R_earth**2*(1.5*z**2/(x**2 + y**2 +
   z**2)**1.0 - 0.5)/(x**2 + y**2 + z**2)**1.0 + 1)/(x**2 +
    y**2 + z**2)**1.5\
     + mu*(2.0*J_2*R_earth**2*z*(1.5*z**2/(x**2 + y**2 + z
        **2)**1.0 - 0.5)/(x**2 + y**2 + z**2)**2.0 - J_2*
        R_earth**2*(-3.0*z**3/(x**2 \
         + y**2 + z**2)**2.0 + 3.0*z/(x**2 + y**2 + z**2)
            **1.0)/(x**2 + y**2 + z**2)**1.0)/(x**2 + y**2 +
             z**2)**0.5

#drag
A_Cross = 6
C_D = 1.88
r_ddot_drag = a_drag(C_D, r, r_dot, A_Cross)

#solar
leap_sec = 37
# del_UT1 = 196.5014 #[s]
r_sun = np.zeros((1, 3))
r_sun[0] = sun_position_vector(JD_UTC, leap_sec, del_UT1[
   day])
C_s = 0.04
C_d = 0.04
A_Cross_sol = 15
r_ddot_sol = a_solar(r, r_sun[0], C_s, C_d, A_Cross_sol)


#third body
r_moon = np.zeros((1, 3))
r_moon[0] = moon_position_vector(JD_UTC, leap_sec, del_UT1[
   day])
```

```
    r_ddot_tb = a_third_body(r, r_sun[0], r_moon[0])


    #total acceleration
    r_ddot = np.array([dUdx, dUdy, dUdz]) + r_ddot_drag  +
       r_ddot_sol + r_ddot_tb

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

    return dydt


def light_time_correction(JD_UTC, r_0, v_0, 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
    '''
    day = int(JD_UTC % JD_UTC_st)

    c = 299792458 #m/s
    ECI_station, _, _= ECEF2ECI(station, np.array([0,0,0]),
       None, JD_UTC, x_p[day], y_p[day], leap_sec, del_UT1[day
       ], LOD[day])
    rho_station = np.linalg.norm(r_0 - ECI_station)
    lt = rho_station/c
    tol = 1e-3 #m
    delta = 1
    old_X_lt = np.zeros(6)
    y0 = np.concatenate((r_0, v_0))
    new_X_lt = y0
    while delta > tol:
        old_X_lt = new_X_lt
        t = JD_UTC - lt/86400
        sol = solve_ivp(satellite_motion, [lt, 0], y0, args=(
           JD_UTC,day), rtol=3E-14, atol=1E-16)
```

```
            new_station, _, _ = ECEF2ECI(station, np.array([0,0,0])
                , None, t, x_p[day], y_p[day], leap_sec, del_UT1[day
                ], LOD[day])
            # print(new_station)
            new_X_lt = sol.y.T[-1]
            new_rho = np.linalg.norm(new_X_lt[0:3] - new_station)
            lt = new_rho/c
            delta = np.linalg.norm(new_X_lt[0:3] - old_X_lt[0:3])
        # print('lighttime', lt, 's')
        # print('lighttime pos diff', np.linalg.norm(new_X_lt[0:3]
            - r_0))
        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*1000**3 #m^3/s^2

        #with gravity
```

```
if gravity:
    R_earth = 6378.1363*1000 #[m]
    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*1000 #[m]
    m = 2000 #[kg]
    theta_dot = 7.292115146706979E-5 #[rad/s]
    rho_0 = 3.614E-13 #[kg/m^3]
    H = 88667.0 #[m]
    r0 = (700000.0 + R_earth) #[m]

    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)

    AU = 149597870700 #[m]
    m = 2000 #kg
    c = 299792458 #m/s
    d = ((r_sun[0]+x)**2 + (r_sun[1]+y)**2 + (r_sun[2]+z)
       **2)**(1/2)
    phi = 1367 #W/m^2
    C1 = phi/c
```

```
        C_s = 0.04
        C_d = 0.04
        v = 1/3*C_d
        mu = 1/2*C_s
        theta = 0

        B = 2*v*sym.cos(theta)+4*mu*sym.cos(theta)**2
        F_x += -C1/(d/AU)**2*(B + (1-mu)*sym.cos(theta))*
           A_Cross_Sol/m*(r_sun[0]+x)/d
        F_y += -C1/(d/AU)**2*(B + (1-mu)*sym.cos(theta))*
           A_Cross_Sol/m*(r_sun[1]+y)/d
        F_z += -C1/(d/AU)**2*(B + (1-mu)*sym.cos(theta))*
           A_Cross_Sol/m*(r_sun[2]+z)/d

    #with thrird body perturbations
    if third_body:
        mu_sun = 132712440018*1000**3 #[m^3/s^2]
        mu_moon = 4902.800066*1000**3 #[m^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, theta], 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')


    rho = sym.sqrt((x - x_s)**2 + (y - y_s)**2 + (z - z_s)**2)
    #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, C_D),
       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
    '''

    x = r[0]
    y = r[1]
    z = r[2]
    mu_sun = 32712440018*1000**3 #m^3/s^2
    mu_moon = 4902.800066*1000**3 #m^3/s^2
    r_sun_mag = np.linalg.norm(r_sun)
    r_moon_mag = np.linalg.norm(r_moon)

    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)
    a_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)
    a_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)
    a_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)

    return np.array([a_x, a_y, a_z])

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:
        m = 2000 #kg
        c = 299792458 #m/s
        AU = 149597870.7*1000 #m
        d = np.linalg.norm(s+r)/AU #distance from sun

        phi = 1367 #W/m^2
        C1 = phi/c
        v = 1/3*C_d
        mu = 1/2*C_s
        theta = 0
        B = 2*v*np.cos(theta)+4*mu*np.cos(theta)**2
        u = (s+r)/np.linalg.norm(s+r)

        r_ddot = (-C1/d**2*(B + (1-mu)*np.cos(theta))*
           A_Cross_sol/m)*u

    return r_ddot

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*1000 #[m]
    m = 2000 #[kg]
    theta_dot = 7.292115146706979E-5 #[rad/s]
    rho_0 = 3.614E-13 #[kg/m^3]
    H = 88667.0 #[m]
```

```
    r0 = (700000.0 + 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_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]
    J_2 = 0.00108248
    R_earth = 6378.1363*1000
    mu = 398600.4415*1000**3
    dUdx = -1.0*mu*x*(-J_2*R_earth**2*(1.5*z**2/(x**2 + y**2 +
       z**2)**1.0 - 0.5)/(x**2 + y**2 + z**2)**1.0 + 1)/(x**2 +
        y**2 + z**2)**1.5 \
        + mu*(3.0*J_2*R_earth**2*x*z**2/(x**2 + y**2 + z**2)
           **3.0 + 2.0*J_2*R_earth**2*x*(1.5*z**2/(x**2 + y**2
           + z**2)**1.0 - 0.5)/(x**2 \
        + y**2 + z**2)**2.0)/(x**2 + y**2 + z**2)**0.5
    dUdy = -1.0*mu*y*(-J_2*R_earth**2*(1.5*z**2/(x**2 + y**2 +
       z**2)**1.0 - 0.5)/(x**2 + y**2 + z**2)**1.0 + 1)/(x**2 +
        y**2 + z**2)**1.5 \
        + mu*(3.0*J_2*R_earth**2*y*z**2/(x**2 + y**2 + z**2)
           **3.0 + 2.0*J_2*R_earth**2*y*(1.5*z**2/(x**2 + y**2
           + z**2)**1.0 - 0.5)/(x**2 \
        + y**2 + z**2)**2.0)/(x**2 + y**2 + z**2)**0.5
    dUdz = -1.0*mu*z*(-J_2*R_earth**2*(1.5*z**2/(x**2 + y**2 +
       z**2)**1.0 - 0.5)/(x**2 + y**2 + z**2)**1.0 + 1)/(x**2 +
        y**2 + z**2)**1.5\
```

```
        + mu*(2.0*J_2*R_earth**2*z*(1.5*z**2/(x**2 + y**2 + z
           **2)**1.0 - 0.5)/(x**2 + y**2 + z**2)**2.0 - J_2*
           R_earth**2*(-3.0*z**3/(x**2 \
            + y**2 + z**2)**2.0 + 3.0*z/(x**2 + y**2 + z**2)
               **1.0)/(x**2 + y**2 + z**2)**1.0)/(x**2 + y**2 +
                z**2)**0.5

    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()
```

```
import numpy as np

def grav_odp(r, C, S):
    x, y, z = r
    mu = 398600.4415*1000**3 #m^3/s^2
    RE = 6378.1363*1000
    #  Initialize variables and determine their size
    nmaxp1, mmaxp1 = C.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**2 + y**2 + z**2
    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], Anm1[1] = 0, np.sqrt(3)
    Anm2[1], Anm2[2] = 0, np.sqrt(3.75)
    as_, at, au, ar, rat1, rat2, Dnm, Enm, Fnm = 0, 0, 0, 0, 0,
        0, 0, 0, 0
    Apor = np.zeros(nmaxp1)
    Apor[0], Apor[1] = 1, RE/rb

    for n in range(1, nmax):
        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] = Apor[i-1]*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 -= Apor[i]*rat1*Anm1[i]*(-C[i-1,0])
        rat2 = np.sqrt(0.5*((an2+1.0)/(an2+3.0))*(n+1.0)*(n
            +2.0))
        ar += Apor[i]*rat2*Anm1[i-1]*(-C[i-1,0])

# Calculate contribution of Tesserals
# Calculate contribution of Tesserals
R = [1]
I = [0]
for m in range(1, mmax+1):
    j = m + 1
    am2 = 2 * m
    R.append(s * R[j-2] - t * I[j-2])
    I.append(s * I[j-2] + t * R[j-2])
    for l in range(m, mmax):
        i = l + 1
        Anm[i] = Anm1[i]
        Anm1[i] = Anm2[i]
    Anm1[mmaxp1] = Anm2[mmaxp1]
    for l in range(m, mmax+1):
        i = l + 1
        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[i-1][j-1] * R[j-1] + S[i-1][j-1] * I[j-1]
        Enm = C[i-1][j-1] * R[j-2] + S[i-1][j-1] * I[j-2]
        Fnm = S[i-1][j-1] * R[j-2] - C[i-1][j-1] * I[j-2]
        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-1] * m * Anm[i] * Enm
        at += Apor[i-1] * m * Anm[i] * Fnm
        au += Apor[i-1] * rat1 * Anm1[i] * Dnm
        ar -= Apor[i-1] * rat2 * Anm1[i+1] * Dnm
```

```
    # Compute the spacecraft accelerations in ECEF
    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 loc_gravLegendre(phi, maxdeg):

    '''
    This function computes the fully normalized associated
       Legendre functions
    and their derivatives up to degree and order maxdeg at
       latitude phi.

    input:
        phi: latitude [rad]
        maxdeg: maximum degree and order of the spherical
           harmonic expansion

    output:
        P: fully normalized associated Legendre functions
        scaleFactor: scaling factor for the fully normalized
           associated Legendre functions

    '''
    # Initialize arrays
    P = np.zeros((maxdeg+3, maxdeg+3, 1))
    scaleFactor = np.zeros((maxdeg+3, maxdeg+3, 1))
    cphi = np.cos(np.pi/2-phi)
    sphi = np.sin(np.pi/2-phi)

    # Force numerically zero values to be exactly zero
    if np.abs(cphi) <= np.finfo(float).eps:
        cphi = 0
    if np.abs(sphi) <= np.finfo(float).eps:
        sphi = 0

    # Seeds for recursion formula
```

```
    P[0,0,:] = 1     # n = 0, m = 0;
    P[1,0,:] = np.sqrt(3)*cphi   # n = 1, m = 0;
    scaleFactor[0,0,:] = 0
    scaleFactor[1,0,:] = 1
    P[1,1,:] = np.sqrt(3)*sphi   # n = 1, m = 1;
    scaleFactor[1,1,:] = 0

    for n in range(2, maxdeg+3):
        k = n + 1
        for m in range(0, n+1):
            p = m + 1
            # Compute normalized associated legendre
               polynomials, P, via recursion relations
            # Scale Factor needed for normalization of dUdphi
               partial derivative
            if n == m:
                P[k-1,k-1,:] = np.sqrt(2*n+1)/np.sqrt(2*n)*sphi
                   *P[k-2,k-2,:]
                scaleFactor[k-1,k-1,:] = 0
            elif m == 0:
                P[k-1,p,:] = (np.sqrt(2*n+1)/n)*(np.sqrt(2*n-1)
                   *cphi*P[k-2,p,:] - (n-1)/np.sqrt(2*n-3)*P[k
                   -3,p,:])
                scaleFactor[k-1,p,:] = np.sqrt((n+1)*n/2)
            else:
                P[k-1,p,:] = np.sqrt(2*n+1)/(np.sqrt(n+m)*np.
                   sqrt(n-m))*(np.sqrt(2*n-1)*cphi*P[k-2,p,:] -
                    np.sqrt(n+m-1)*np.sqrt(n-m-1)/np.sqrt(2*n
                   -3)*P[k-3,p,:])
                scaleFactor[k-1,p,:] = np.sqrt((n+m+1)*(n-m))

    return P, scaleFactor

def loc_gravityPCPF(p, maxdeg, P, C, S, smlambda, cmlambda, GM,
    Re, r, scaleFactor):
    '''
    Computes the gravity acceleration in the ECEF frame

    input:
        p: position vector in ECEF frame [m]
        maxdeg: maximum degree and order of the spherical
           harmonic expansion
        P: fully normalized associated Legendre functions
        C: cosine spherical harmonic coefficients
        S: sine spherical harmonic coefficients
```

```
    smlambda: sine of the product of the longitude and
       degree
    cmlambda: cosine of the product of the longitude and
       degree
    GM: gravitational constant times the mass of the Earth
       [m^3/s^2]
    Re: mean radius of the Earth [m]
    r: magnitude of the position vector [m]
    scaleFactor: scaling factor for the fully normalized
       associated Legendre functions

output:
    gx: x gravity acceleration in the ECEF frame [m/s^2]
    gy: y gravity acceleration in the ECEF frame [m/s^2]
    gz: z gravity acceleration in the ECEF frame [m/s^2]
'''

rRatio   = Re/r
rRatio_n = rRatio.copy()

# initialize summation of gravity in radial coordinates
dUdrSumN      = 1
dUdphiSumN    = 0
dUdlambdaSumN = 0
# summation of gravity in radial coordinates
for n in range(2, maxdeg+1):
    k = n+1
    rRatio_n      = rRatio_n*rRatio
    dUdrSumM      = 0
    dUdphiSumM    = 0
    dUdlambdaSumM = 0
    for m in range(n+1):
        j = m
        dUdrSumM      = dUdrSumM + P[k-1,j]*(C[k-1,j]*
           cmlambda[:,j] + S[k-1,j]*smlambda[:,j])
        dUdphiSumM    = dUdphiSumM + ((P[k-1,j+1]*
           scaleFactor[k-1,j,:] - p[2]/np.sqrt(p[0]**2 + p
           [1]**2)*m*P[k-1,j])*(C[k-1,j]*cmlambda[:,j] + S[
           k-1,j]*smlambda[:,j]))
        dUdlambdaSumM = dUdlambdaSumM + m*P[k-1,j]*(S[k-1,j
           ]*cmlambda[:,j] - C[k-1,j]*smlambda[:,j])
    dUdrSumN      = dUdrSumN      + dUdrSumM*rRatio_n*k
    dUdphiSumN    = dUdphiSumN    + dUdphiSumM*rRatio_n
    dUdlambdaSumN = dUdlambdaSumN + dUdlambdaSumM*rRatio_n
```

```
    # gravity in spherical coordinates
    dUdr      = -GM/(r**2)*dUdrSumN
    dUdphi    =  GM/r*dUdphiSumN
    dUdlambda =  GM/r*dUdlambdaSumN

    # gravity in ECEF coordinates
    gx = ((1/r)*dUdr - (p[2]/(r**2*np.sqrt(p[0]**2 + p[1]**2)))
       *dUdphi)*p[0] \
          - (dUdlambda/(p[0]**2 + p[1]**2))*p[1]
    gy = ((1/r)*dUdr - (p[2]/(r**2*np.sqrt(p[0]**2 + p[1]**2)))
       *dUdphi)*p[1] \
          + (dUdlambda/(p[0]**2 + p[1]**2))*p[0]
    gz = (1.0/r)*dUdr*p[2] + ((np.sqrt(p[0]*p[0] + p[1]*p[1]))
       /(r*r))*dUdphi

    # special case for poles
    atPole = np.abs(np.arctan2(p[2], np.sqrt(p[0]*p[0] + p[1]*p
       [1]))) == np.pi/2
    if np.any(atPole):
        gx[atPole] = 0
        gy[atPole] = 0
        gz[atPole] = (1.0/r[atPole])*dUdr[atPole]*p[atPole,2]
    # print(gx, gy, gz)
    return gx[0], gy[0], gz[0]

def F_gravity_vallado(p):
    '''
    # F_GRAVITY_VALLADO computes the acceleration due to
       gravity in the
    # Earth-Centered Inertial (ECI) frame using the Vallado
       algorithm.

    # Inputs:
    #   p_ECI - Nx3 array of ECI positions [m]

    # Outputs:
    #   g_ECEF - Nx3 array of ECI accelerations [m/s^2]'''


    maxdeg = 6
    mu = 3.986004418e14 # m^3/s^2
    Re = 6378.145*1000 #[m] # m
    r = np.linalg.norm(p)

    # Compute geocentric latitude
```

```
    phic = np.arcsin(p[2] / r)

    # Compute lambda
    lambda_ = np.arctan2(p[1], p[0])

    smlambda = np.zeros((p.shape[0], maxdeg+1))
    cmlambda = np.zeros((p.shape[0], maxdeg+1))

    slambda = np.sin(lambda_)
    clambda = np.cos(lambda_)
    smlambda[:,0] = 0
    cmlambda[:,0] = 1
    smlambda[:,1] = slambda
    cmlambda[:,1] = clambda

    for m in range(2, maxdeg+1):
        smlambda[:,m] = 2.0 * clambda * smlambda[:, m-1] -
           smlambda[:, m-2]
        cmlambda[:,m] = 2.0 * clambda * cmlambda[:, m-1] -
           cmlambda[:, m-2]

    # compute normalized legendre polynomials
    P, scaleFactor = loc_gravLegendre(phic, maxdeg)
    # print(P.shape)

    # Compute gravity in ECEF coordinates
    gx, gy, gz = loc_gravityPCPF(p, maxdeg, P, C[0:maxdeg+1, 0:
       maxdeg+1],
                                  S[0:maxdeg+1, 0:maxdeg+1],
                                     smlambda,
                                  cmlambda, mu, Re, r,
                                     scaleFactor)

    g_ECEF = np.array([gx, gy, gz])
    return g_ECEF

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

    Inputs:
        JD_UTC - Julian date in UTC
        r - position vector of satellite in ECI frame [m]
        v - velocity vector of satellite in ECI frame [m/s]
```

```
    station_ECEF- position vector of station in ECEF frame
       [m]

Outputs:
    rho - range from station to satellite [m]
    rho_dot - range rate from station to satellite [m/s]
'''
day = int(JD_UTC % JD_UTC_st)

leap_sec = 37 #s
station_ECI, station_dot_ECI, _ = ECEF2ECI(station_ECEF, np
   .array([0,0,0]), None, JD_UTC, x_p[day], y_p[day],
   leap_sec, del_UT1[day], LOD[day])
# print('station_dot_ECI', station_dot_ECI/1000)
rho = np.linalg.norm(r - station_ECI)
rho_dot = np.dot(r - station_ECI, v - station_dot_ECI)/rho

return rho, rho_dot
```