Lab 04 High Level Function Implementation and Debugging

Adbul Cisman and Michael Stewart

October 09 2020

Purpose

The Purpose of this lab is to combine and refine the high level program created in P3 as well as create a Status Report of the Progress. This lab focuses on constructing the AOS/LOS table, comparison of correct test values and code values and to provide a summary of the progress completed.

Procedure

The Lab was completed by Abdul and Mike together with Mike focusing on the Pointing and Visibility Functions and Abdul focusing on the Link Calculations and Debugging. The Theory behind the lab process was to construct a working code using the functions developed over the past several weeks. The code must output similar values to the correct ones.This lab focused on implementing the AOS/LOS table and selecting a satellite to output the pointing information.

The Steps Conducted in this lab include:

- Discuss issues with P3 code and how to implement P4 code
- Coding
- New Code
- Debugging
- Report Making

Pre-Lab Discussion

We had discussed issues surrounding the P3 code. Most notably: the wrong answers being outputted by the algorithm. Another issue arose is the Debugging: the methods being used weren’t great for Low Level Functions.

Coding

After we had discussed briefly what we needed for this Lab we started Coding. Coding was loosely divided into two sections:

Creating New Code

Coding of the Visibility Block of the Programming Structure started by asking the question “what is the most efficient way to produce the AOS/LOS?”. The answer to this was through the Pointing function since it already had the for loops and if statements to find availability. The only thing left to do was to compare the availability of the Satellite to its previous value, which surprisingly turned out to be more difficult than we thought (more on this in Results/Discussion). The next step was to format the code correctly. Here a Visibility() function was created to turn the AOS/LOS variables collected in the Pointing() function and turned them into a list also containing the Signal Loss variables.

The Signal Loss variables come from the Tracking Data. The Tracking Data(so far) is formed from the Link Calculations and the Range of the Topocentric System and only contains Signal Loss-- more variables will be calculated later.. A function called linkcal takes in the LinkInputs.txt file and creates variables of frequency, Antenna Diameter and Antenna Efficiency. A created function called Tracking_Data() takes in the linkcal variables and R_ti and produces a signal loss for each Satellite at each time period..

The Pointing() and Visibility() functions take in this Signal_loss parameter so that it can be easily displayed in the AOS/LOS Table.

The Signal Loss Equation used was the Ls variable in Figure 1 in the Results Section of this lab.

Debugging

After the new functions were added, the code underwent many debugging exercises, but still suffers from inaccurate calculations. Bugs were first fixed when the problem was evident: this meant that an error occurred during the codes operation. As of now, the code runs but doesn’t output the right answer. We are currently looking at the Kepler Equation, and Mean Anomaly functions to see why they give faulty results.

A running account of all changes made to the code can be found at :
[https://github.com/mstewart2525/ENG4350Lab03Draft2/commits/main/MainCall.py (https://github.com/mstewart2525/ENG4350Lab03Draft2/commits/main/MainCall.py)](https://github.com/mstewart2525/ENG4350Lab03Draft2/commits/main/MainCall.py)

Report

Throughout and after the Coding process this lab report was constructed to detail the construction and implementation of updated code.

Results and Discusssion

The function sat_pos_velcall will input the stationinstance, satlist, and tracking. Using the tracking time and timestep specified in the tracking function, it will iterate the time. Running the code will input values for the empty arrays for the time, satellite number, Azimuth, elevation, rate of Az, rate of EL, topocentric range, and topocentric velocity. For the pointing function, it uses the station instance, azimuth, elevation and time to make a list of available look angles. By using the station instance limits, it will make a list of the time , azimuth and elevation in that limit. In the pointing function, it will create an array of azimuth and elevation AOS that will be empty. When running the code, each iteration will update the azimuth AOS and elevation AOS. For the LOS, it will also create a list of the available azimuth and elevation from each iteration.

Doing some tests runs for the AOS/LOS list provides a list of data where it tells us which satellites are within the antennas field of regard during a given time interval. Saving and extracting that list from python and pasting it into a txt file gave us the following; **For the AOS data**,

[0.6443993039618087, 1.3065213971607885, 0.8820609149866927, 1.0550772131713888, 0.5625506543524079, 1.5651529071094603, 0.3317653048302409, 1.5674485050388631, 1.5661010059394367, 0.1966352640783303, 0.6751775678668904, 0.7297122368941906, 1.5655487928600278, 1.4339290781144196]

[0.4279817803282175, 0.7102474890992245, 0.5732126015304646, 0.6661437471931716, 0.08830566720932495, 1.0209368022530299, 0.08820102623434249, 1.2386909223084297, 1.1600268999020429, 0.08755565312124093, 0.08859707424721387, 0.08917675250243381, 0.6574878950128256, 0.08753001755878748]

[datetime.datetime(2020, 9, 28, 16, 0), datetime.datetime(2020, 9, 28, 16, 0), datetime.datetime(2020, 9, 28, 16, 0), datetime.datetime(2020, 9, 28, 16, 0), datetime.datetime(2020, 9, 28, 16, 9), datetime.datetime(2020, 9, 28, 17, 5), datetime.datetime(2020, 9, 28, 17, 16), datetime.datetime(2020, 9, 28, 17, 32), datetime.datetime(2020, 9, 28, 17, 50), datetime.datetime(2020, 9, 28, 18, 3), datetime.datetime(2020, 9, 28, 18, 5), datetime.datetime(2020, 9, 28, 19, 50), datetime.datetime(2020, 9, 28, 19, 53), datetime.datetime(2020, 9, 28, 19, 55)]

[5, 21, 28, 29, 25, 10, 9, 23, 15, 7, 1, 17, 6, 3]

For the LOS data list [-0.000558598619825001, -0.0017923312367464548, -0.002920747118975327, -0.0007432726125056399, -0.004123906115591683, 0.8749801084026285, 0.6318431269674142, 0.7362346919902719, -0.0006970303948109347, 0.3788777527177542, -0.00024095548723373464]

[1.2797246583941533, 1.2144750266447715, 0.8382021700452705, 0.2097067900475309, 1.0703283838317064, 0.08649913349137837, 0.08675435047983115, 0.08635255413029086, 0.13558980994994027, 0.08657383474055819, 0.21902695347923595]

[datetime.datetime(2020, 9, 28, 16, 4), datetime.datetime(2020, 9, 28, 16, 7), datetime.datetime(2020, 9, 28, 16, 54), datetime.datetime(2020, 9, 28, 18, 37), datetime.datetime(2020, 9, 28, 18, 37), datetime.datetime(2020, 9, 28, 18, 58), datetime.datetime(2020, 9, 28, 19, 10), datetime.datetime(2020, 9, 28, 19, 11), datetime.datetime(2020, 9, 28, 19, 27), datetime.datetime(2020, 9, 28, 19, 30), datetime.datetime(2020, 9, 28, 19, 47)]

[23, 15, 6, 8, 24, 21, 28, 29, 7, 5, 9]

As stated in the result above, the pointing function will create a list of available azimuth, elevation, time and the satellite number. The available azimuth, elevation, and satellite have a size of 3404. Just for reference on how the output of the list will look like, I have selected the first 15 values from each list. See below for the output result.

The list of Azimuth available (First 15 values) [0.7818110464888374, 0.6540685924222422, 0.6443993039618087, 0.24991120649144485, 0.3505383042166046, 0.7574265298144688, 0.8315895504525326, 0.0676931764019619, 1.2310028582138839, 1.3065213971607885, 0.04700625400192833, 0.598077413764019, 0.8820609149866927, 1.0550772131713888, 0.7804963303337401]

The list of Elevation available (First 15 values)
[0.13270527152966294,0.3318209975632347,0.7060500116315267,0.1969524166005553,0.4786757495531191 \

0.30281286349532255,0.5215040022538783,0.6788826773751089,0.6817981722626224,1.1979614545734716,

The following classes/functions were edited to update the functionality of the module. Additionally some new subroutines were added like the **refepoch_to_dt()**.

New Subroutines

Def refepoch_to_dt(referepoch)

This function was created to easily convert TLE epoch time format to a datetime object. A datetime object was used to easily translate between times.

In [4]:
```
# import section of Code
import datetime as dt
import numpy as np
import math
from scipy.spatial.transform import Rotation as R
```

In [3]:
```
def refepoch_to_dt(refepoch):
    Epochyrday = dt.datetime.strptime((refepoch[0:5]),'%y%j')
    dfrac = np.modf(np.float(refepoch))[0]
    dfracdt = dt.timedelta(microseconds=np.int(dfrac*24*3600*10**6))
    Epochdt = Epochyrday + dfracdt
    return Epochdt
```

doy(YR,MO,D)

This function was created to easily convert year, month and day to the day of the year. It is an edited version of P1's lab

referenceepoch_propagate(Tracking_Data_Instance)

After reviewing the code of THETAN with Professor Chesser, the refernpoch_propagte function was created. This function creates an array of TLE formatted times to plug into the function THETAN.

In [ ]:
```
# Finds the Day of Year from the Year Month and Day parameters
def doy(YR,MO,D):
    if len(str(MO))==1:
        MO='0'+ str(MO)
    if len(str(D))==1:
        D='0'+ str(D)
    String=str(YR)+str(MO)+str(D)
    Time = dt.datetime.strptime(String,'%Y%m%d')
    #Converts First to dt
    DOY=dt.datetime.strftime(Time,'%j')
    #Converts from dt to day of the year
    return DOY
```

Visibility

The Visibility function takes in the AOS and LOS data collected from the Pointing function and essentially formats the data into a List containing [Satellite number, Satellite Name, Satellite AOS Time, Satellite LOS Time(datetime), Signal Loss(dB)]

Inserted later since it uses the Pointing Function which has not be implemented yet

LinkCal

This function calculates the signal loss from the link data provided. So it will take in the link input data parsed through the user parser. Using the formula provided in the lecture notes, we calculated the signal loss. See below of a screenshot of the formula.

In [13]:
```
Image(filename="Linkcal.jpg")
```

Out[13]:

$$C = \overbrace{PL_1G_t}^{EIRP}\ \overbrace{L_s}^{L_s=\left(\frac{c}{4\pi Rf}\right)^2}\ \overbrace{G_r}^{G_r=\frac{\pi^2 f^2 D_r^2 \eta}{c^2}}\ \overbrace{G_s}^{\text{System Gain}}$$

Focusing on the loss, c is the speed of light, f is the frequency and R is the path length. This will calculate the loss in dB for every satellite that is in the field of regard.

In [ ]:
```
def linkcal(linkdat):
  Linkcalcfile=open(linkdat, 'rt')
  frequency=Linkcalcfile.readline()
  Antennaeff=Linkcalcfile.readline()
  AntennaDia=Linkcalcfile.readline()
  Linkcalcfile.close
  #signalloss=20*math.log(10,4*math.pi*(float(AntennaDia))/(3.0e8*float(freque
ncy)*1e6))
  return frequency,Antennaeff,AntennaDia
```

TrackingData

This function creates a Signal Loss List for each Satellite at each time interval

In [18]:
```
def TrackingData(freq,Antennaeff,AntennaDia,R_ti):
    Signal_loss=[]
    for i in range(0,len(R_ti)):
        R=np.linalg.norm(R_ti[i])
        #Signal_loss.append(20*math.log(10,4*math.pi*(float(R))/(3.0e8*float(f
req)*1e6)))
        Signal_loss.append((3.0e8)/(4*math.pi*(float(frequency)*1e6)*((R_ti
))))**(2)
    return Signal_loss
```

Updated Subroutines

THETAN(refepoch)

The updated function now takes in the iterative times as referepoch TLE formats. The referpoch times are created from the tracking data and propagated into a list from referencepoch_propagate(Tracking_Data_Instance). The functions has also been edited to fix minor bugs (i.e. Time Bug)

In [7]:
```
def THETAN(refepoch):
    #Input is a refepoch array this dsoesn't make sense as Tracking Data contains an easily parsible


    GMST_list=[]


    J2000=dt.datetime.strptime('2000-01-01 12:00:00','%Y-%m-%d %H:%M:%S')
    Starttime=dt.datetime.strptime(Tracking.starttime,'%Y-%m-%d-%H:%M:%S')
    Midtime=Starttime.replace(hour=0,minute=0,second=0)
    D_u=(Midtime-J2000).days+(Midtime-J2000).seconds/86400
    T_u=D_u/36525
    GMST_00=(99.9677947+36000.7700631*T_u+0.00038793*T_u**2-2.6e-8*T_u**3)%360




    for i in range(0,len(refepoch)):
        times=(refepoch_to_dt(refepoch[i]))

        #Creates T mid for Observation Day
        #Notice how we replace hour,min and sec to 0. This makes the time midnight!
        del_sec=(times-Midtime).total_seconds()


        D_u_2=(times-J2000).days+(times-J2000).seconds/86400

        T_u_2=D_u_2/36525

        r=1.002737909350795+5.9006e-11*(T_u_2-T_u)-5.9e-15*(T_u_2-T_u)**2
        GMST_t=(GMST_00+360*r/86400*(del_sec))%360

        GMST_list.append(GMST_t)

    #Low Level Debug Helper
    global zTest_GMST_List
    zTest_GMST_List=GMST_list

    return GMST_list
```

Pointing()

During a revision of the Suggested Programming Structure it was noticed that it would be easier to find the AOS/LOS within the Pointing Function. This was decided since the Availability of the Azimuth and Elevation could be found and an Acquisition and Loss of Signal requires that parameter. So the Program uses the Pointing function to output AOS/LOS values used by the Visibility function.

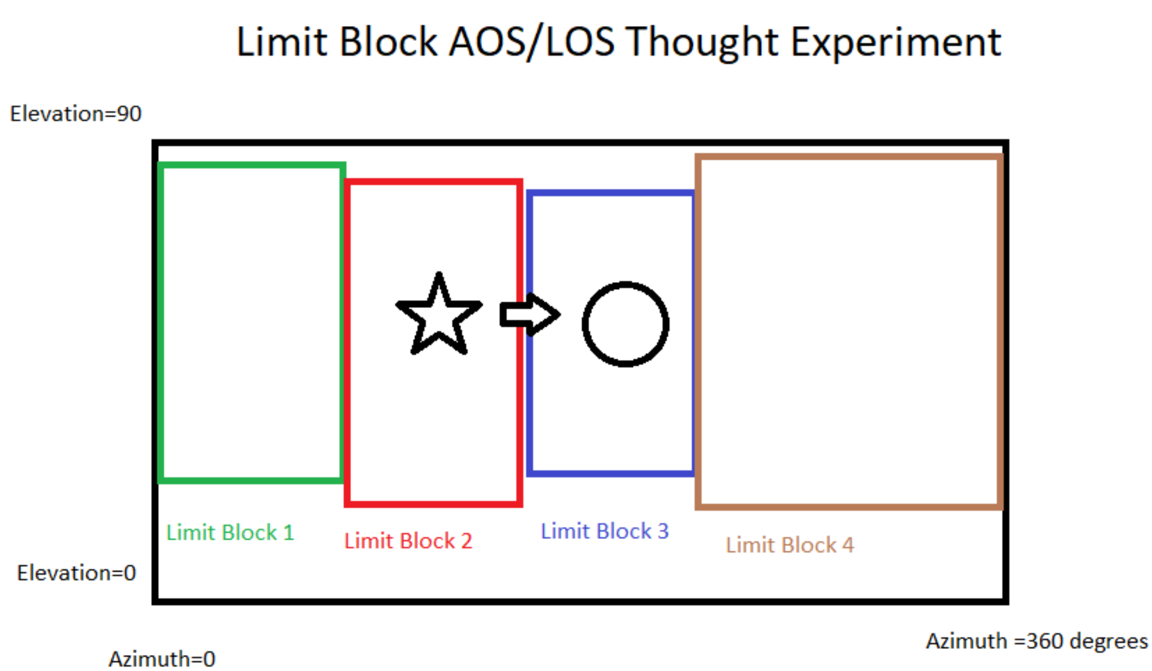
The Updated Pointing Subroutine continues to iterate through the Station Limits and check the availability of the satellite but now it compares the previous value for that specific satellite to see if it is available. If a Satellite signal is available when it wasn't, then a signal is acquired.

It has been noticed that after the Satellite iterates through one whole Azimuth and Elevation limit, some Acquisition and Loss of Signal points might be noted. This is because the code works in what I like to call "limit blocks". Passing into a limit block will grant an acquisition of signal even if the Satellite was previously in another "limit block" the timestep earlier- this is unwanted. Due to the nature of the iterative limits, we have attached a special section of code that checks and sees if the Satellite, at its previous timestep had a Loss of Signal, due to leaving that "limit block". If the Satellite has an Acquisition of Signal at the current timestep but a Loss of of Single at that same step in a different "limit block" than the Loss of Signal gets deleted from the entry system and an Acquisition of Signal gets prevented from being added.

To fully understand this we devise a Block Limit Thought Experiment

In [6]:
```
from IPython.display import Image
Image(filename="Limit_Block.png")
```

Out[6]:



Let's imagine that we have an Object that Moves from Star to Circle through two different Limit Blocks. Our code, before it was fixed, would tell us that the Satellite has lost a Signal and simultaneously acquired a signal in the Circle Position, both in different Limit Blocks. The Limit Block 2 would lose a signal while Limit Block 3 would acquire a signal. The appended code in our Acquisition of Signal Section searches through the LOS list generated and sees if a Satellite with the same number and Time as our Acquisition Signal and deletes it’s entry from the LOS list.

In [8]:
```
def Pointing(StnInstance,AZ_list,EL_list,time,Satnum_list,Signal_lost):


  #avail=available for viewing
  AZ_avail=[]
  EL_avail=[]
  Times_avail=[]
  Satnum_avail=[]
  #creates blanks arrays to be filled
  AZ_AOS=[]
  EL_AOS=[]
  Times_AOS=[]
  SatNum_AOS=[]
  AZ_LOS=[]
  EL_LOS=[]
  Times_LOS=[]
  SatNum_LOS=[]
  Satnum_iteration=max(Satnum_list)+1
  time_delta=time[1]-time[0]
  Signal_lost_AOS=[]
  Signal_lost_LOS=[]
  Signal_lost_avail=[]




  for i in range(0,int(StnInstance.az_el_nlim)):

      # For Each iteration of the Station Instation Limits
    ThisStationLimit=StnInstance.az_el_lim[i].split(",")

    # Covnerts Limits from Degrees to Rads
    AZ_muth_limit=float(ThisStationLimit[0])*math.pi/180
    EL_lim_max=float(ThisStationLimit[2])*math.pi/180
    EL_lim_min=float(ThisStationLimit[1])*math.pi/180


    for j in range(0,len(AZ_list)):

      if AZ_list[j] > (AZ_muth_limit) and (EL_lim_max) > EL_list[j] and EL_lis
t[j] > (EL_lim_min):
        #compares Azimuth and Elevations to Limits

          #Satellite Available


          #Compares to Previous Value
          # If previous value is not in limits but is now either AOS or from a
nother limit iteration**
        if j >= (Satnum_iteration) and  Satnum_list[j] == Satnum_list[j-Satnum
_iteration] and AZ_list[j-Satnum_iteration] < float(AZ_muth_limit) or float(EL
_lim_max) < EL_list[j-Satnum_iteration] or EL_list[j-Satnum_iteration] < float
(EL_lim_min):
            # A Signal has been acquired

            #** Here we test to see if the AOS was caused by limit bound issue
```

```
s or from actual AOS

            if i > 0:

                for k in range(0,len(Times_LOS)):

                    if time[j] == Times_LOS[k] and Satnum[j]== SatNum_LOS[k]:
                        append=0
                        del Times_LOS[k]
                        del SatNum_LOS[k]
                        del AZ_LOS[k]
                        del EL_LOS[k]
                    else:
                        append=1
                        break
                if append==1:
                    AZ_AOS.append(AZ_list[j])
                    EL_AOS.append(EL_list[j])
                    Times_AOS.append(time[j])
                    SatNum_AOS.append(Satnum_list[j])
                    Signal_lost_AOS.append(Signal_lost[j])

                #Then no AOS, LOS

            # This is actual AOS
            else:
                AZ_AOS.append(AZ_list[j])
                EL_AOS.append(EL_list[j])
                Times_AOS.append(time[j])
                SatNum_AOS.append(Satnum_list[j])
                Signal_lost_AOS.append(Signal_lost[j])



        AZ_avail.append(AZ_list[j])
        EL_avail.append(EL_list[j])
        Times_avail.append(time[j])
        Satnum_avail.append(Satnum_list[j])
        Signal_lost_avail.append(Signal_lost[j])
      else:
          # Satellite Unavailable

          #Compares previous iteration of Satellite to current Satellite Avail
able
          if j >= (Satnum_iteration)  and Satnum_list[j]==Satnum_list[j-(Satnu
m_iteration)] and AZ_list[j-(Satnum_iteration)] > float(AZ_muth_limit) and flo
at(EL_lim_max) > EL_list[j-(Satnum_iteration)] and EL_list[j-(Satnum_iteration
)] > float(EL_lim_min):
              #If Satellite was available but is no longer then it is either L
OS or out of our Limit Range
              #

              AZ_LOS.append(AZ_list[j])
              EL_LOS.append(EL_list[j])
              Times_LOS.append(time[j])
              SatNum_LOS.append(Satnum_list[j])
              Signal_lost_LOS.append(Signal_lost[j])
```

```
  AOS_List=[AZ_AOS,EL_AOS,Times_AOS,SatNum_AOS,Signal_lost_AOS]
  LOS_List=[AZ_LOS,EL_LOS,Times_LOS,SatNum_LOS,Signal_lost_LOS]
#Creates a list of Available Azimuth, Elevation, Times and Satnum available


  return AZ_avail,EL_avail,Times_avail,Satnum_avail,AOS_List,LOS_List
```

Now We Can Insert the Visibility Function

In [9]:

```
def Visibility(StationInstance,AZ,EL,times,Satnum,Signal_lost):
    #[AZ_avail,EL_avail,Times_avail,Satnum_avail]=Pointing(StationInstance,AZ,EL,Satnum)
    AOS=Pointing(StationInstance,AZ,EL,times,Satnum,Signal_lost)[4]
    LOS=Pointing(StationInstance,AZ,EL,times,Satnum,Signal_lost)[5]
    Satnum_AOS=AOS[3]
    Satnum_LOS=LOS[3]
    Sat_Signal_Lost=AOS[4]

    Sat_AOS_Time=AOS[2]
    Sat_LOS_Time=LOS[2]
    AOS_LOS_list=[]
    for i in range(0,len(Satnum_AOS)):
        for j in range(0,len(Satnum_LOS)):
            if Satnum_AOS[i] == Satnum_LOS[j]:
                Templist=[Satnum_AOS[i],SatList[Satnum_AOS[i]].name,Sat_AOS_Time[i],Sat_LOS_Time[j],Sat_Signal_Lost[i]]
                #creates a temporary list to store Sat number,Sat list name ,AOS time,LOS time and
                AOS_LOS_list.append(Templist)






    return AOS_LOS_list
```

Debugging

Since the last Lab, The modules built in P3 have been patched together and now work without any obvious error. The unknown issues now lie within output discrepancy-- which will be ironed out by the end of the Tracking section of this course. Multiple Methods were used to identify errors. A Method preferred and implemented by Mike was to create a cell at the bottom of the code that would create instances of low-level functions so that their values would be available in the variable explorer. From here the faults in our code were isolated. Another method used was to print out important variables into the console to view if those variables were low-level

The Debugging Process can be divided into two parts:

Intial Debugging

The first bug to be noticed was the Azimuth/Elevation comparison bug. Since The Azimuth and Elevation were output in Radians by the range_topo2look_angle() and have degrees in our Station File the Pointing() function had to have a conversions of degrees to radians to compare see if the AZ and EL were within the Station File Limits.

The second bug to be noticed is the Time bug. This bug was caused by a whitespace misplacement leading to the Time datetime object not containing a + deltatime(timestep) every iteration of Time_iterations in the sat_pos_velcall() function

Comparison of Values

Close or Correct Values

Running the Algorithm created the PRN 01 satellite at timestep 0, it has a Position and Velocity in the Perifocal Frame of [17793.41,19477,0] and [-2.86,2.6514,0]. Creating an STK simulation with same gps-ops file and transforming it to a perifocal frame results in a position of PRN 01 of [17914.14, 19850.007,0].

The GMST Angle Calculated from the STK Simulation for the first time step is 247.623 degrees. This agrees with our Calculations. Our Algorithm spits out a GMST value of 247.4 which is relatively close. An issue with referen_to_dt() and doy function resulted in the time values being before the Mid_Time Values, resulting in faulty GMST angles. This was quickly remedied

Incorrect Values

Running the created Algorithm for PRN 01 satellite results in the ECI Range Coordinates of [15815.03,-21110.571,-449.696] The STK simulation resulted in [-13331.93,6829.82,22148.414]. The wrong answers outputted from the ECI function could have resulted from numerous sources. Our guesses as of now are the incorrect values for the Kepler Equation function or Mean Anomaly function.

Unaltered Code

This Code, from previous labs has either not been altered or altered so little it doesn't make much of a difference.

In [ ]:
```
def mean_anomaly_motion(time,ts_sat_epoch,M0_mean_anomaly,n_mean_motion, \
                        n_dot_mean_motion,n_2dots_mean_motion):

    #Assume Reference Epcoh is in TLE format
    refepoch=ts_sat_epoch
    Epochyrday = dt.datetime.strptime(refepoch[:4],'%y%j')
    dfrac = np.modf(np.float(refepoch))[0]
    dfracdt = dt.timedelta(microseconds=np.int(dfrac*24*3600*10**6))
    Epochdt = Epochyrday + dfracdt

    #assume Time is datetime object

    t=(time-Epochdt).total_seconds()



    Mt_mean_anomaly=M0_mean_anomaly+ \
        n_mean_motion*(360*t/86400)+360*(n_dot_mean_motion)*(t/86400)**2+ \
        360*(n_2dots_mean_motion)*(t/86400)**3
    Nt_mean_anomaly_motion=n_mean_motion* \
        (360*t/86400) + 2*360*(n_dot_mean_motion)*(t/86400**2)+ \
        3*360*(n_2dots_mean_motion)*(t**2/86400**3)

    #Removing Mutlples
    Mt_mean_anomaly=Mt_mean_anomaly%360

    return Mt_mean_anomaly,Nt_mean_anomaly_motion
```

In [ ]:
```
def KeplerEqn(Mt_mean_anomaly,eccentricity):

    #Examples Permitted Error
    permitted_error=0.32
    #Example Permitted Error


    Mt_mean_anomaly=float(Mt_mean_anomaly)*math.pi/180
    #Converts to Radians

    e=float(eccentricity)
    #ensures that e is in the right format



    #Initialize lists
    E_=[]
    Del_M_=[]
    Del_E_=[]
    i=0

    #Calculates First Iteration
    E_.append(float(Mt_mean_anomaly))
    Del_M_.append(float(E_[i])-e*math.sin((E_[i]))-float(Mt_mean_anomaly))
    Del_E_.append(Del_M_[i]/(1-e*math.cos(E_[i])))
    Del_E_mag=abs(Del_E_[i])
    E_.append(E_[i]+Del_E_[i])

    #Calculates Further Iterations
    while Del_E_mag > permitted_error:
        i=i+1
        Del_M_.append(float(E_[i])-e*math.sin((E_[i]))-float(Mt_mean_anomaly))
        Del_E_.append(Del_M_[i]/(1-e*math.cos(E_[i])))
        Del_E_mag=abs(Del_E_[i])
        (Del_E_mag)
        E_.append(E_[i]+Del_E_[i])


    return E_[i+1]%(2*math.pi) #reduces Eccentric Anom
#! Returns in Radians
```

In [ ]:
```
def perifocal(eccentricity,ecc_anomaly,a_semi_major_axis,omega_longitude_ascen
ding_node, \
              omega_argument_periapsis,inclination,nt_mean_motion):

    #Assuming Earth
    mu=398600.4418 #km^3/s^2

    #Ensuring Orbital ELements are in right format
    eccentricity=float(eccentricity)
    ecc_anomaly=float(ecc_anomaly)
    omega_longitude_ascending_node=float(omega_longitude_ascending_node)
    omega_argument_periapsis=float(omega_argument_periapsis)
    inclination=float(inclination)
    nt_mean_motion=float(nt_mean_motion)

    #Calculating True Anomaly
    true_anom=2*(math.atan(math.sqrt((1+eccentricity)/ \
                                     (1-eccentricity))*math.tan(ecc_anomaly)))
    #Calculating R and its components
    r=a_semi_major_axis*(1-eccentricity**2)/(1+eccentricity*math.cos(true_anom
))
    r_px=r*math.cos(true_anom)
    r_py=r*math.sin(true_anom)
    r_pz=0
    R_per=[r_px,r_py,r_pz]
    #Calculating Velocity Components
        #Note: We could not differentiate r in python so we used a work around
method

    semi_lactus_rectum=a_semi_major_axis*(1-eccentricity**2)
    angular_mo=math.sqrt(mu*semi_lactus_rectum)
    v_px=-mu*math.sin(true_anom)/angular_mo
    v_py=mu*(eccentricity+math.cos(true_anom))/angular_mo
    v_pz=0
    v_per=[v_px,v_py,v_pz]
    #km/s
    print("Perifocal:",R_per,v_per)

    return R_per,v_per
```

In [ ]:
```
def sat_ECI(eccentricity,ecc_anomaly,a_semi_major_axis,omega_longitude_ascendi
ng_node,omega_argument_periapsis,inclination,nt_mean_motion):
    #Perifocal to ECI
    #Earth Centred Inertial Frame

    #Finds Perifocal Components
    r_per,v_per=perifocal(eccentricity,ecc_anomaly,a_semi_major_axis,omega_lon
gitude_ascending_node,omega_argument_periapsis,inclination,nt_mean_motion)
    print("Position and Velocity in Perifocal",r_per,v_per)

    #Creates transformation
    Per_to_ECI=R.from_euler('ZXZ',[-float(omega_longitude_ascending_node),-flo
at(inclination),-float(omega_argument_periapsis)],degrees=True)
    #Note: RAAN,omega,inc in degrees
    pos_ECI=(Per_to_ECI.apply(r_per)).tolist()
    vel_ECI=(Per_to_ECI.apply(v_per)).tolist()

    print("Position in ECI",pos_ECI)

    vel_ECI=(Per_to_ECI.apply(v_per)).tolist()
    return pos_ECI,vel_ECI
```

In [ ]:
```
def sat_ECF(theta_t,eci_position,eci_velocity):

    print("This is the intake ECI Position",eci_position)
    #Creates rotational transformation
    ECI_to_ECF=R.from_euler('Z',[float(theta_t)], degrees=True)

    #Applies Rotational Transformation
    pos_ECF=ECI_to_ECF.apply(eci_position).tolist()[0] # had to perform weird
    #weird conversion to get back to list
    print(pos_ECF)
    vel_ECF=ECI_to_ECF.apply(eci_velocity).tolist()[0]
    #km/s

    print("This is the Position in ECI",eci_position)
    #Relative Velocity
    Siderial_rotation=[1,1,360/86164.091] #Degrees/s
    vel_rel=ECI_to_ECF.apply(eci_velocity-np.matmul(Siderial_rotation,eci_posi
tion))
    #km/s


    return pos_ECF,vel_ECF,vel_rel
```

In [ ]:
```
def station_ECF(station_longitude,station_latitude,station_elevation):
    #input as rads only

    f=1/298.25223563
    R_e=6378.137 #km
    #geodetic longitde must be in degs
    #geodetic latitude must be in degs
    #station elevation must be in km
    phi=(float(station_latitude)*math.pi/180)
    h=float(station_elevation)
    lambda_=float(station_longitude)*math.pi/180
    e=math.sqrt(2*f-f**2)
    n_phi=R_e/(math.sqrt(1-(e**2)*(math.sin(phi))**2))


    T_x=(n_phi+h)*math.cos(phi)*math.cos(lambda_)
    T_y=(n_phi+h)*math.cos(phi)*math.sin(lambda_)
    T_z=(((1-e**2)*n_phi+h)*math.sin(phi))

    #R_x=station_body_position[0]-T_x
    #R_y=station_body_position[1]-T_y
    #R_z=station_body_position[2]-T_z

    #where station_body_poisitonh is the sat_ECF coordinates

    return [T_x,T_y,T_z]


  #Note: Professor said that we did not have to do the second station_EFC fucn
tion
```

In [ ]:
```
def range_ECF2topo(station_body_position, \
                   sat_ecf_position,sat_ecf_velocity,station_longitude, \
                   station_latitude):

    station_longitude=float(station_longitude)*math.pi/180
    station_latitude=float(station_latitude)*math.pi/180
    #input as Rads only

    print("This is Station Body Positon",station_body_position)

    #assuming that station body positon is [Tx,Ty,Tz]

    R=[sat_ecf_position[0]-station_body_position[0],\
       sat_ecf_position[1]-station_body_position[1],\
        sat_ecf_position[2]-station_body_position[2]]
    print("This is R:  ",R)

    #Intializes Transformation Matrix
    T_ECF_to_topo=[[-math.sin(station_longitude), \
                    math.cos(station_latitude),0], \
                   [math.cos(station_longitude)*math.sin(station_latitude), \
                    -math.sin(station_longitude)*math.sin(station_latitude), \
                    math.cos(station_latitude)],[math.cos(station_longitude) \
                                             *math.cos(station_latitude),
 \
                                             math.sin(station_longitude)*
 \
                                             math.cos(station_latitude), \
                                             math.sin(station_latitude)]]
    print("This is Transformation: ",T_ECF_to_topo)
    print("This is R_Transpose: ",(R))
    #Transform Range Vector
    R_ti=np.matmul(np.array(R),np.array(T_ECF_to_topo))

    # Assuming that sat_ecf Velocity is Relative
    vel_rel=sat_ecf_velocity

    #Transform Velocity Vector
    v_rel_ti=np.matmul(np.array(vel_rel),np.array(T_ECF_to_topo)).tolist()[0]

    return R_ti,v_rel_ti
```

In [ ]:
```
def range_topo2look_angle(range_topo_position,range_topo_velocity):
    R=range_topo_position
    print("REange Topo position",range_topo_position[0])
    v_rel=range_topo_velocity
    print("This is Topo Range",range_topo_position)
    #Calculates the AZ and EL
    AZ=math.atan(R[0]/R[1])
    EL=math.atan(R[2]/(math.sqrt(R[0]**2+R[1]**2)))


    r=np.linalg.norm(R) #scalar of R

    R_xy=[R[0],R[1]]
    #In Software Specification Rxy is [tx ty]{Rtx;Rty} which would give a resu
lt of a singular value
    #Here we assume the Professor meant R_xy= the x and y components of R
    print(v_rel)
    v_xy=[v_rel[0],v_rel[1]]

    #Calculates rates of AZ and EL
    rate_of_AZ=np.cross(v_xy,R_xy)
    rate_of_EL=(r*v_rel[2]-R[2]*np.dot(R_xy,v_xy)/r)/(r**2)

    return AZ,EL,rate_of_AZ,rate_of_EL
```

In [ ]:
```
def SatListPropagate(SatFIL):
    Satfile=open(SatFIL,'rt')
    entries=(len(open(SatFIL).readlines()))/3
    i=0
    while i<entries:
        line0=Satfile.readline()
        line1=Satfile.readline()
        line2=Satfile.readline()
        try:
            SatList.append(Satellite(line0,line1,line2))
            print('Satellite has been registered as Satellite:',len(SatList),
"Array index: [",len(SatList)-1,"]")
        except:
            SatList=[]
            SatList.append(Satellite(line0,line1,line2))
            print('SatList has been created')
        i=i+1
    return SatList
```

Classes

In [15]:
```
class Station():
    def __init__(self,STNFIL):
        STNfile=open(STNFIL,'rt')
        #opens file using directory extension
        self.name=STNfile.readline()
        self.stnlat=STNfile.readline()
        self.stnlong=STNfile.readline()
        self.stnalt=STNfile.readline()
        self.utc_offset=STNfile.readline()
        self.az_el_nlim=STNfile.readline()
        self.az_el_lim=[]
        Iteration=int(self.az_el_nlim)
        i=0
        while i<Iteration:
            self.az_el_lim.append(STNfile.readline())
            i=i+1

        #self.az_el_lim=STNfile.readline()
        self.st_az_speed_max=STNfile.readline()
        self.st_el_speed_max=STNfile.readline()
        #reads the file line by line

        STNfile.close
        #closes files
```

In [16]:
```
class Satellite():
    def __init__(self,line0,line1,line2):
        self.name=line0
        self.refepoch=line1[18:32]
        self.incl=line2[8:16]
        self.raan=line2[17:25]
        self.eccn="."+line2[26:33]
        self.argper=line2[34:42]
        self.meanan=line2[43:51]
        self.meanmo=line2[52:63]
        self.ndot=line1[33:43]
        self.n2dot=line1[44:50]
        self.bstar=line1[53:61]
        self.orbitnum=line2[63:68]
```

In [17]:
```
class tracking():
    def __init__(self, TrackingData):
        Trackingdatafile=open(TrackingData,'rt')
        self.starttime=(Trackingdatafile.readline())[0:19]
        self.endtime=(Trackingdatafile.readline())[0:19]
        self.timestep=Trackingdatafile.readline()
        #It is assumed that the step time is in seconds
        Trackingdatafile.close
```

In [ ]:
```
class linkinput():
    def __init__(self, LinkInputs):
        Linkinputfile=open(LinkInputs, 'rt')
        self.frequency=Linkinputfile.readline()
        self.Antennaeff=Linkinputfile.readline()
        self.AntennaDia=Linkinputfile.readline()
        self.Bandwidth=Linkinputfile.readline()
        self.RCVgain=Linkinputfile.readline()
        self.RCVnoise=Linkinputfile.readline()
        Linkinputfile.close
        #t=4*math.pi*(46)/(3.0e8*1.575e9)
        #signalloss=20*math.log(10,4*math.pi*(46)/(3.0e8*1.57542e9))
        #Will print out signal loss from the given values. 46 is antenna diame
ter
        #3.0e8 is speed of light and 1.57542e9 is frequency in Hz from MHz
        #print(signalloss)
```

Main Calling Functions

In [ ]:
```
def User_Input_parser_Call(StationLocationStr,TLEfilestr,TrackingSchedulestr,L
inkInputsstr):
    StationInstance=Station(StationLocationStr)
    SatList=SatListPropagate(TLEfilestr)
    Tracking=tracking(TrackingSchedulestr)
    LinkData=linkinput(LinkInputsstr)
    return StationInstance,SatList,Tracking,LinkData
```

In [ ]:
```
def Sat_pos_velCall(StationInstance,SatList,Tracking):
    #StationInstant, Times, and Links are class instances with their own attri
butes
    # Inputs: Station Instance Calculated, Satellite List, Time






    #We are assuming that in the next version of the code we will be iterating
through Start and End times
    #With a time step. This will be our Time value
    Time_start_dt=dt.datetime.strptime(Tracking.starttime,'%Y-%m-%d-%H:%M:%S')
#This is to be used as a place holder until we iterate through the times
    Time_end_dt=dt.datetime.strptime(Tracking.endtime,'%Y-%m-%d-%H:%M:%S')

    #Assuming that timesteps is in seconds
    Time_iterations=(Time_end_dt-Time_start_dt).total_seconds()/float(Tracking
.timestep)

    print("This is Time Iteration", Time_iterations)
    Time_dt=Time_start_dt
    #initializing empty arrays
    time=[]
    Satname=[]
    AZ_list=[]
    EL_list=[]
    Rate_of_AZ_list=[]
    Rate_of_EL_list=[]
    R_ti_list=[]
    v_rel_ti_list=[]
    Satnum_list=[]
    #Satnum List helps with identifying the Satellite

    #Propagtes data for THETAN
    Refepoch=referenceepoch_propagate(Tracking)
        #THETAN has been edited to input time_start_dt and Time_dt
    GMST=THETAN(Refepoch)

#Iterates through satellite list first then for time
#creates list of
    for i in range(0,int(Time_iterations)):

      for p in range(0,int(len(SatList))):


        [Mt_Mean_anomaly,Nt_anomaly_motion]=mean_anomaly_motion(Time_dt,SatLis
t[p].refepoch,float(SatList[p].meanan),float(SatList[p].meanmo),float(SatList[
p].ndot),float(SatList[p].n2dot))
        #degrees,
        ecc_anomaly=KeplerEqn(Mt_Mean_anomaly,SatList[p].eccn)
        #returns in radians
        mu=398600.4418 #km^3/s^2
        a=(mu/(2*np.pi*float(SatList[p].meanmo)/86400)**2)**(1/3)
        [pos_ECI,vel_ECI]=sat_ECI(SatList[p].eccn,KeplerEqn(SatList[p].meanan,
SatList[p].eccn), \
```

```
        a,SatList[p].raan,SatList[p].argper,SatList[p].incl,Nt_anomaly_motion)


        GMST_1=GMST[i]
        [pos_ECF,vel_ECF,vel_rel_ECF]=sat_ECF(GMST_1,pos_ECI,vel_ECI)
        #Note: We assume that station_body_position is Tx,Ty,Tz
        [Tx,Ty,Tz]=station_ECF(StationInstance.stnlong,StationInstance.stnlat,
StationInstance.stnalt)


         #Note: Station Long and Latitude must be in Radians
        [R_ti,v_rel_ti]=range_ECF2topo([Tx,Ty,Tz],pos_ECF,vel_rel_ECF,StationI
nstance.stnlong,StationInstance.stnlat)


        [AZ,EL,Rate_of_AZ,Rate_of_EL]=range_topo2look_angle(R_ti,v_rel_ti)



        Satnum_list.append(p)
        AZ_list.append(AZ)
        EL_list.append(EL)
        Rate_of_AZ_list.append(Rate_of_AZ)
        Rate_of_EL_list.append(Rate_of_EL)
        R_ti_list.append(R_ti)
        v_rel_ti_list.append(v_rel_ti)

        time.append(Time_dt)
      Time_dt=Time_dt+dt.timedelta(seconds=float(Tracking.timestep))
      #At the End change Time



    #Start Time is in EST and is converting inside function
    #Note: we have made changes to the THETAN code to also input Tracking.star
ttime. As of this version,
    #Time now is used for the t variable. This is to be changed in later versi
ons when we iterate through time








    return AZ_list,EL_list,Rate_of_AZ_list,Rate_of_EL_list,R_ti_list,v_rel_ti_
list,time,Satnum_list
```

Main Function

In [ ]:
```
#Assuming That The Use has already initialized all necessary functions and classes
# The Main Program can be deduced to this
[StationInstance,SatList,Tracking,LinkData]=User_Input_parser_Call(r'Station.txt',r'gps-ops.txt',r'TrackingData.txt',r'LinkInputs.txt')
[AZ,EL,Rate_of_AZ,Rate_of_EL,R_ti,v_rel_ti,time,Satnum]=Sat_pos_velCall(StationInstance,SatList,Tracking)

[freq,Antennaeff,AntennaDia]=linkcal(r'LinkInputs.txt')
Signal_loss=TrackingData(freq,Antennaeff,AntennaDia,R_ti)
[AZ_avail,EL_avail,Times_avail,Satnum_avail,AOS_List,LOS_List]=Pointing(StationInstance,AZ,EL,time,Satnum,Signal_loss)
#Visibility creates a formatted list
[AOS_LOS_list]=Visibility(StationInstance,AZ,EL,time,Satnum,Signal_loss)


#Outputs AZ in Rads
```

The Output AOS_LOS_list ouputs a list for [SAtellite number,Satellite Name,Satellite AOS Time,Satellite LOS Time, Signal Loss]

Conclusion

In this section of the lab we learned how much effort debugging takes. We learned that to create a code that runs smoothly is one thing, but to get the right result takes patience and determination. We also learned a variety of things while trying to debug the code. It is best to create a good Debugging system at the start of the coding process and debug as you go rather than spending time to revisit the code. STK is a lot more complex that we had initially assumed and we struggled to find the right way to find convert axes. Learning about field of regard during the office hours with Arvin helped us understand and determine which satellites are within the antenna's field of regard. That is, the calculated satellite Azimuth and Elevation positions have to be inside the field of regard so that the telescope can track the satellite. It would be better if the results of Low Level Functions were provided for this Lab.