Toward Satellite Coverage Optimization for the Contiguous United States

Group Member Participation

- Diego Lopez Fleming: Wrote intro and methods. Code analysis. Wrote angularEarthCoord, arcCrossing, arcCoverage functions, arcLongInterval, deg2rad function. Wrote code calling these functions.
- Kirsten Soules: Wrote Project Proposal Tables, edited & assembled Project Proposal (updated table portion). Code analysis. Ran meetings and took meeting minutes, assigned tasks as necessary. Wrote the following functions: timeSinceStart, convertDateToSecSinceStart, daytimeFunction, callingDaytime, daytimeArcCrossings, daySwath, callDaySwath, dayCoverageConcat, ratioDaySwath. Wrote code calling these functions. Modified multiple other functions and function calls to accommodate time. Edited & encapsulated ground tracking visualization function from last semester's code to work with this semester's code. Wrote the following functions for graphs: groundtrackingViz, dayGroundtrackingViz, locationOfDaytimeCrossings, zoomedInlocationOfDaytimeCrossings, coverageRatioViz. Made additional graphs (not in the final notebook) to validate accuracy of functions. Edited choice of variables description and wrote justification for choices. Wrote and generated graphs for Appendices B & C.
- Marisa Bowens: wrote abstract for project proposal, math analysis (defining Kepler elements, Keplerian equations, swath equation, key contiguous U.S. data points organized into documents for review as well as assisted with catching group up on key equations we would use in the project) wrote function to set Keplerian elements of a satellite as well as the following functions: c3d2, merge_intervals_order_helper, merge_intervals, sumarc. Final Project Abstract, Captioned all graphs and conclusion. Wrote Appendix A.
- Shiro Criszia: Code analysis. Helped Diego edit Introduction portion of final proposal. Made diagrams and wrote captions for all diagrams that were uploaded to Google Colab. Chose end points of arc. Included Required Files section. Wrote table with different combinations of variables and the description that goes with it, as well as uploaded all graphs for this section. Checked for grammatical and spelling errors.

Required Files

Linked below are images of graphs that we use throughout the notebook. All images are stored in the sp2020 / Ball / Final Notebook Images folder in Google Drive.

Diagrams

Result Graphs

Appendix B

Appendix C

Abstract

Ball Aerospace is using the CIRiS (Compact Infrared Radiometer in Space) technology to examine evapotranspiration on the Earth's surface and provide full coverage of the Earth within a 24-hour period by tracking Earth's surface temperatures. This data will help scientists and decision makers evaluate drought conditions and climate models. This project will estimate satellite coverage of the contiguous United States and optimize the global coverage that was explored in previous projects. This project could lead to better visualization and optimization of more finite areas of the globe, such as the coastal regions, to more accurately depict extreme weather patterns.

Introduction

Keplerian elements will be used to define this problem. These elements consist of orbital inclination, semi-major axis, the argument of periapsis, longitude of the ascending node, and true anomaly. These variables will be used in Kepler's Equation, an equation that relates the geometric properties of the orbit of a body subject to a central force. In this case, Ball satellites are the bodies subject to the central force of the Earth. The satellite optimization parameters will be encapsulated in functions.

Ball is focused on optimizing daytime satellite coverage. The optimization of satellite coverage is important because Ball's satellites are equipped with CIRiS (Compact Infrared Radiometer in Space). One capability of CIRiS that is of particular interest is tracking evapotranspiration levels of the Earth. Evapotranspiration is a scientific variable that combines evaporation from the Earth's surface and the transpiration of water vapor from Earth's plants. Our goal is to achieve complete satellite coverage of the contiguous U.S. in one day using as few satellites as possible. If Ball Aerospace's satellites can provide complete coverage of the United States, then valuable data on the evapotranspiration levels of the U.S. could be used in commercial and scientific fields. Achieving complete coverage with fewer satellites would also benefit Ball Aerospace in saving satellite resources.

This project is also a continuation of Ball sponsored projects from previous semesters. The project in Fall 2019 projected the swath onto the Earth to evaluate coverage (ground tracking), focusing on equatorial coverage. The project in Spring 2020 incorporated the ground tracking code from the prior semester and added additional constraints that the coverage needed to be obtained during daylight hours. This project expands on both of the previous semesters and focuses on encapsulating calculations into functions as well as limiting the coverage to the continental United States. The final papers from Fall 2019 and Spring 2020 can be found at the CU Denver Math Clinic website: https://clas.ucdenver.edu/math-clinic/projects-table.

Several assumptions about Ball Aerospace's satellites will be made during calculations. They will be in sun-synchronous orbit, meaning they cross the equator at the same local time on Earth every day, but it is important to note that the precession correction has not been applied yet. The satellite's orbits will also be in a near-polar orbit where the angle between the poles and the orbit is approximately 8 degrees. Each satellite is equipped with two CIRiS instruments that each provide a 15-degree field of view, for a total of 30 degrees. Field of view is the angle at which the instrument can observe the Earth at any instant in time. Using these assumptions, along with tools like Kepler's equation, we hope to optimize the number of satellites that it takes to achieve full coverage of the contiguous United States.

Methods

In the research paper "A Study into Satellite Coverage and Orbital Equations," the Keplerian elements involved in an ECEF (Earth centered Earth fixed) orbit are explained extensively and briefly discussed below. There are six parameters that are necessary to describe the motion of one celestial body in relation to another for a two-body problem. In our case, these parameters are important for calculating satellite coverage on Earth. A brief overview of these elements is listed below:

1. Semimajor axis (a)- half of the length of the line segment that runs through the center and both foci, with ends at the widest points of the perimeter.
2. Eccentricity (e)- parameter that determines the amount by which its orbit around another body deviates from a perfect circle (0 being perfectly circular and 1 being a parabolic escape orbit).
3. Inclination (i)- measurement of the satellite orbit's tilt around from the Earth's equatorial plane.
4. Longitude of ascending node (Ω)- the angle between the reference direction and the upward crossing of the orbit on the reference plane.
5. Argument of Periapsis (ω)- the angle between the ascending node and the periapsis, and is measured in the direction of the satellite's motion.

6. True Anomaly (ν)- the position of the orbiting body along the trajectory, measured from periapsis.

The objective of the Spring 2020 project was to set up a fully functional simulation of equatorial coverage given the input orbital parameters above and to increase the flexibility of this simulation with regard to time. Our goal is to set up a fully functional simulation of the coverage of the contiguous U.S. while continuing to refine the time variable. We decided our approach would be to encapsulate last semester's code into functions, making it easier to use, and use an arc to estimate the boundary of the contiguous U.S. that the satellites are crossing over. Reviewing the code from last semester's project, we identified a few aspects of the code that would be helpful to our project goals. We then made a code outline to discuss what functions would be necessary and what they should be able to accomplish. Taking useful cells from last semester's project, we encapsulated some code cells into functions, such as the conversion of 3D Cartesian coordinates to spherical Earth coordinates. Some functions, such as the calculation of swath and the creation of the 3D satellite coordinates, were preserved. Prof. Fournier then provided assistance by writing a function that creates an arc given two coordinate points, as well as parameters that associate the arc with the longitudinal and latitudinal coordinates of the satellites. One of these parameters was the angular displacement to the endpoints of the arc. When there is a sign change in this parameter, we know that our satellite has crossed the latitude of our arc. Knowing the before and after longitudinal and latitudinal coordinates of the satellites, an interpolated line was used to find the exact crossing of the satellite over the arc latitude. The coverage of the arc was then computed using the mathematical coverage calculations done by the prior semester's group. Another goal of ours was to use Python packages to help with finding the coordinates of satellites in the daytime. We utilized the Ephem package (https://pypi.org/project/ephem/) to give us information as to whether satellite ground tracking corresponded to daytime on Earth by calculating the angle of the sun above the horizon from the point of reference (the point of reference being the ground tracking of the satellite). It's preferable that we only find coverage during the daytime because we are gathering information on evapotranspiration, which relies in part on the ground temperature. We also found a use for the Ephem package in setting the orbital parameters of our satellites. Another important

Packages

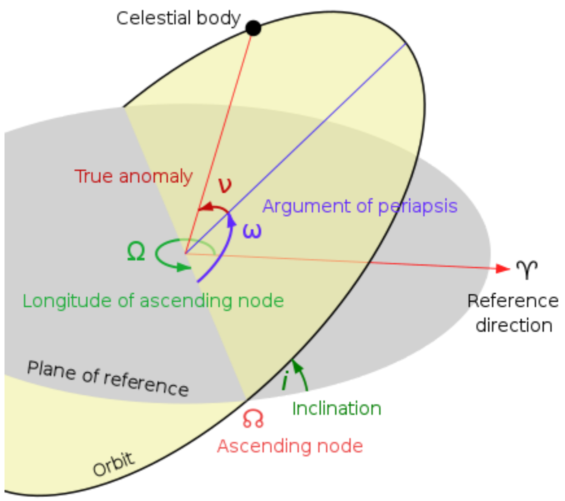
```
from IPython.display import HTML, display
display(HTML("<style>.container { width:95% !important; }</style>"))
from scipy.optimize import fsolve
import matplotlib
import matplotlib.pyplot as plt
from matplotlib import cm
import numpy as np
import ephem    #https://pypi.org/project/ephem/
import datetime
from datetime import time
from datetime import time
%matplotlib inline
  # %matplotlib inline needed for jupyter notebook, displays plot in result of code ce
from IPython.display import Javascript
  # used to show all graphs in one cell without internal scroll bar
```

Function Definitions

Creating 3D Orbit Coordinates

Professor Fournier wrote this function. This function creates the 3D Cartesian coordinates for satellites using Keplerian elements. There are six Keplerian elements necessary to describe the motion of one celestial body in relation to another.

- This function accepts time, orbital inclination, semi-major and minor axis, longitude of the ascending node, eccentricity, argument of perigee, and angular momentum.
- This function outputs 3D orbit coordinates of the satellites.



- The variables shown on the picture were explained on the cell above this, under "Methods" section.

Source for Keplerian elements picture: https://en.wikipedia.org/wiki/Orbital_elements

```
# Eq. 25, https://en.wikipedia.org/wiki/Kepler_orbit#Properties_of_trajectory_equatior
# Note eq. 24 implies a unique E for all t and 0 < e < 1
```

```
# Note eq. 24 implies a unique E for all t and 0 < e < 1.
## keep in mind capital omega for seasonal changes
def sat3D(t, i, ω, Ω,
          a, b, H, E) :          # create 3D orbit coordinates (q[0],q[1],q[2])
    o = [np.cos(Ω),              # longitude of the ascending node
          np.sin(Ω)]             # ... converted to its cosine and sine
    s = np.sin(i)                # sine of inclination angle, radians
    j= 0
    l = t[j:]                    # shift time between plane rotation and E ***j assignec
    l = 2*np.pi*l/24/3600        # convert l to orbital plane normal-vector azimuth (rac
    ν = np.array([np.cos(l)*s, # unit normal vector of orbital plane
                  np.sin(l)*s,
                       np.array([np.cos(i)]*len(l))]).transpose()
    λ = np.cross([0, 0, 1], ν) # vector orthogonal to ν and to reference North Pole
    λ = np.diag(np.sum(λ*λ,    # normalize λ
                axis=1)**-.5)@λ
    μ = np.cross(ν, λ)          # unit vector orthogonal to λ and to ν
    E = E - ω + np.pi/2          # eq. 39 implies E ≈ θ when 0 < e << 1
    x = a*np.cos(E)              # eq. 20, x-coordinate list (shifted to ellipse center)
    y = b*np.sin(E)              # eq. 21, y-coordinate list


    orbit3D = (np.diag(x)@λ + np.diag(y)@μ)@np.array([[o[0], -o[1], 0],
                                                      [o[1],  o[0], 0],
                                                      [   0,     0, 1]])

    return orbit3D
```

Swath Width Function

This function gives the swath width that the satellite detects across its ground track using the mathematical equation for the calculation of swath.

- This function accepts the variable thetaSat (angle of the camera) and h (altitude of satellites).
- This function outputs a numeric value of swath.

```
def swath_coverage(thetaSat, h):
    swath = (2 * h * np.tan( ( (np.pi / 180) / 2 ) * thetaSat))
    return swath
```

This function takes in an angle in degrees and outputs that angle in radians.

```
def deg2rad(degrees):
    radians = degrees * np.pi / 180
    return radians
```

Global Variables

```
# ***Anomaly; Prof. FOURNIER WROTE
# Eq. 25, https://en.wikipedia.org/wiki/Kepler_orbit#Properties_of_trajectory_equatior
# Note eq. 24 implies a unique E for all t and 0 < e < 1.
t2E = lambda e, t : fsolve(lambda E : E - e*np.sin(E) - t, t/(1 - e))
# number of days we want to simulate
num_days = 1
# number of satellites we wish to simulate -- << 13
satellites = 9
# altitude in meters
h = 525000
# eccentricity
e = .01
# np.sin takes radians ==> 84°
inclination = deg2rad(84)
# mean earth radius, m https://en.wikipedia.org/wiki/Earth_radius
# can be changed to 6378.1343e3 --> This value will convert to exactly
# 40,075 km when evaluating coverage. The mean Earth radius will yield
# a maximum of 40,035 km

R = 6371.0088e3
G = 6.67430e-11                 # gravitational constant, m³/(kg s²) https://en.wikipe

m = [1e3, 5.9722e24]            # masses of satellite, Earth, kg https://en.wikipedia.
N = 350                         # nu. plot points

α = G*sum(m)                    # gravitational parameter, eq. 1, m³/s²
a = (R + h)/(1 - e)             # eq. 35, R + minimum altitude solved for semi-major a
p = a*(1 - e*e)                 # eqs. 13--14, r(θ=π/2), θ being the true anomaly
b = a*(1 - e*e)**.5             # eq. 15, semi-minor axis
H = (α*p)**.5                   # eq. 26, specific relative angular-momentum magnitude
P = 2*np.pi*a**1.5/α**.5        # eq. 43, orbital period for an elliptic orbit, s
# orbital period
opd = (24*60*60) / P            # roughly 16 orbits in one day -- exactly for a perioc

orbits = (num_days)*opd         # number of orbits we will simulate
thetaSat = 30
swath = swath_coverage(thetaSat,h)    # swath based on calculation of altitude
halfSwath = swath/2


sun_day = -6                  #Angle of sun is 6 degrees below horizon (civil sunrise/
hours2seconds = 60 * 60       # number of seconds in 1 hour
HOURS = 14                    # Sets the number of hours past 00:00:00 UTC
satellite_start = datetime.datetime(2020, 3, 19, HOURS, 0)  # sets starting time to '2
float_sat_start = float(satellite_start.timestamp())        # turns datetime start tim
# Use equinox for satellite start date, as the equinox ensures both northern and
# southern hemispheres get same amount of sunlight. Hence this is a good model for
# an average amount of sunlight over the year.
# Equinox is on 3/19/2020
# https://en.wikipedia.org/wiki/Equinox#:~:text=This%20occurs%20twice%20each%20year,is
```

```
#sec_to_hrs = 3600                  #this is a global variable that returns values ir
seconds= 240                       # seconds and radm for sunrise/sunset lines in gra
radm = (np.pi/180)*(1/seconds)     #longitude moves 1 degree every 240 second and 1 c

arc_latitude = deg2rad(39.7392)    # Latitude of Denver: 39.7392°, approx 0.69 radian
# Latitude of Denver is a rough approximation for the latitude of the geographic cente
# Future semesters may wish to use the latitude of the geographic center: (39.8283° N,
# https://en.wikipedia.org/wiki/Geographic_center_of_the_contiguous_United_States

Global_longitude = 111320          #1°longitude = cos(latitude) *111.32 km
rad = 57.2958                      # 1 rad = 57.2958°
m2r = (1/Global_longitude) * (1/rad)

#arc_latitude is assigned for y1 and y2 to keep the arc level across the US
# (x1)= longitude of Madawaska, Maine
y1 = arc_latitude
x1 = deg2rad(-68.3217)## long

# (x2)= longitude of Blaine, Washington
y2 = arc_latitude
x2 = deg2rad(-122.7471)## long

# Arc west of Mississippi: Wickliffe, Kentucky (36.966600°N 89.086822°W)
                      # https://en.wikipedia.org/wiki/Wickliffe,_Kentucky
#y1 = arc_latitude
#x1 = deg2rad(-89.086822) ## longitude of Wickliffe, Kentucky
```

Converting 3d Cartesian Coordinates to Spherical Longitude and Latitude Coordinates

This function takes an array of satellite 3D Cartesian coordinates and returns the variables phi and psi of the 3D spherical coordinates. Phi and psi represent longitude and latitude, respectively.

Uses mathematical trigonometric conversions from 3D Cartesian coordinates to spherical coordinates.

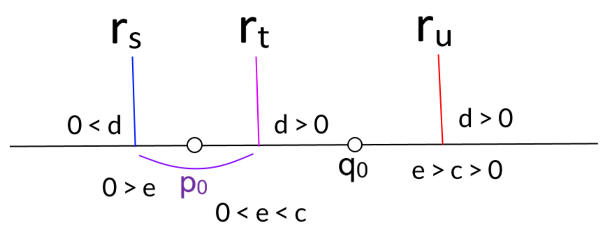
- Accepts variable orbit3d, which is an array of 3D satellite coordinates.
- Returns phi and psi of the 3D Cartesian coordinates.

```
def angularEarthCoord(orbit3D):
    phi = [0]*satellites
    psi = [0]*satellites
    # filling previous two arrays
    for i in range(0,satellites):
        # signed angle between (1,0) and (q[i][:,0],q[i][:,1]), # x-coordinate for grc
        phi[i] = np.arctan2(orbit3D[i][:,1], orbit3D[i][:,0])
        # creating array of zeroes the size of q[i][:,2]
        qq = np.zeros(np.size(orbit3D[i][:,2]))
        for j in range(0,np.size(orbit3D[i][:,2])):
        # qq is manipulating third array of q, latiture
            qq[j] = orbit3D[i][j,2] / np.sqrt( orbit3D[i][j,0]**2 + orbit3D[i][j,1]**
        psi[i] = np.arcsin( qq )
    return phi,psi
```

Arc Display Function

This function tests if the ground tracking is in the arc's longitude limits or not.

- This function accepts 2D (long, lat) pairs p, q, and r. The variable r is the longitude and latitude of the satellite. The variables q and p are the longitudes and latitudes of the endpoints in the US that create our arc. We chose Blaine, Washington, and Madawaska, Maine to represent our longitudinal endpoint as they are on the far west and far east coasts of the contiguous United States. The arc is set at the latitude of Denver, Colorado which is a rough approximation of the latitude of the geographic center of the contiguous United States.
- This function outputs c, e, s, and d. The variable c represents the angular distance from p to q. The variable e represents the angular distance from p to s. The variable s is a vector that points to where r would cross the arc if the satellite was traveling directly towards the arc. The variable d represents the angle of the arc to the ground tracking point. This function was written by Professor Fournier as a means to visualize the crossing of the arc. Note this function is used to determine if a crossing of the arc has happened and not the exact location of that particular crossing. It is also important to note that there are cases in which the ground tracking point is in the sector, but the satellite crosses the arc outside of the sector. There are also cases in which the ground tracking point is outside of the sector, but the satellite crosses the arc inside the sector.



- p_0 and q_0 are the longitude interval (which is what we defined as our "sector"). In this case, p_0 and q_0 will cover the contiguous US.
- d tells us if it is before or after arcLatitude. The arcLatitude is a continuous latitude line that we selected to intersect the contiguous US.

- This drawing shows 3 different situations that could happen (r_s, r_t, and r_u). The variables r_s and r_u implies that e is less than zero or above c, so the point is outside the desired sector. If e is in between 0 and c, then it is in the sector that we want.

```
def dispArc(p, q, r) :                                   # angular displacement of r fr
  u = np.zeros((3,3))
  x = (p, q, r)
  for i in range(3) :                                    # loop over 3 (lon,lat) pairs
    u[i,:2] = np.array([np.cos(x[i][0]), np.sin(x[i][0])]
                      )*np.cos(x[i][1])                  # equatorial-plane Cartesian c
    u[i,2] = np.sin(x[i][1])                             # ... becomes unit vector to x
  v = np.cross(u[0], u[1])                               # vector normal to disk D cont
  c = np.sqrt(np.dot(v, v))                              # norm of v
  v /= c                                                 # normalize v
  c = np.arccos(np.dot(u[0], u[1]))                      # angular distance from p to c
  d = np.arcsin(np.dot(v, u[2]))                         # angle from D to u[2]
  s = u[2] - v*np.dot(v, u[2])                           # projection of u[2] onto plan
  s /= np.sqrt(np.dot(s, s))                             # unit vector to where r woul
  e = np.arccos(np.dot(u[0], s))                         # angular distance from u[0] t
  #print(d, s, e, c, u, sep='\n')                        # comment this out when satis
  return d, s, e                                         # e < c ensures r lies in the
```

Arc Crossing Detection Function

This function checks for a sign change in the angle of the arc to the ground tracking of the satellite. The angle of the arc to the satellite is given in the return of the dispArc function in the d variable. A positive d value indicates that the coordinate of the satellite is still before the latitude of the arc. A negative d value indicates that the satellite coordinate is past the latitude of the arc. This function then fills longB4Af and latB4Af variable with the coordinates of longitude and latitude before and after crossing the latitude of the arc. These are in radians. The first element of each variable corresponds to the coordinate before crossing while the second element corresponds to the coordinate after crossing. This is done by inputting the corresponding index of the latitudes and longitudes from the indexPosNeg array into the j values of longCoord and latCoord variables.

- This function accepts dd. dd is a list of the d values for each satellite given by the dispArc function, longCoord, and latCoord.
- This function returns longB4Af, latB4Af, and t_B4Af. These are the times, latitudes, and longitudes of the satellites before and after crossing.

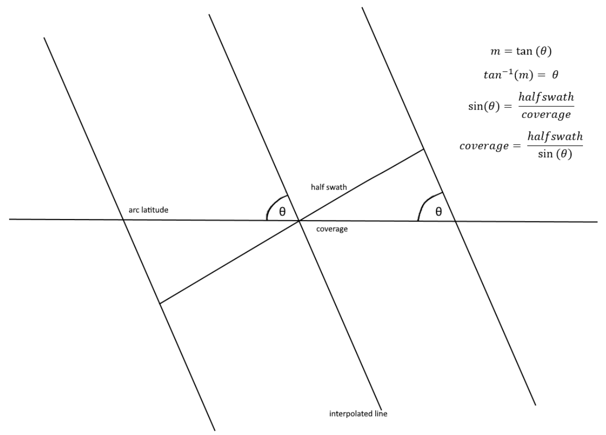
```
#arcCrossing function checks all the d values that come out of the dispArc function
#looking via for loop for a sign change in the d values
def arcCrossing(dd,longCoord,latCoord):
    #creating variable that will store the index of the index of the longitude and lat
    #indexPosNeg stores the index of the longitude and latitude
    #as the andgle from the arc to the point of crossing changes sign,signaling an arc
    indexPosNeg = ['']*satellites
    for i in range(0,satellites):
        indexPosNeg[i]=[];
        #checking for sign change
        #if there is a sign change, the index
        #of the latitude before crossing is stored in
        #indexPosNeg
        for j in range(0,np.size(dd[i])-1):
            if( dd[i][j] > 0 and dd[i][j+1] < 0 ) or ( dd[i][j] < 0 and dd[i][j+1] > (
                indexPosNeg[i].append(j)
#creating arrays of 2d vectors latB4Af and longB4Af
#first element of latB4Af is the latitude before crossing of arc
#second element of latB4Af is the latitude after crossing of arc
#first element of longB4Af is the longitude before crossing of arc
#second element of longB4Af is the longitude after crossing of arc
    longB4Af = ['']*satellites
    latB4Af = ['']*satellites
    t_B4Af = ['']*satellites
    for i in range(0,satellites):
        longB4Af[i] = [];
        latB4Af[i] = [];
        t_B4Af[i]=[];
        for j in range(0,np.size(indexPosNeg[i])-1):
        #appending before after latitude and longitude points
        #the j index of longCoord and latCoord variables are given by the index value
        #the second element being appended are the latitude and longitude coordinates
        #the index before crossing, again the index of latCoord and longCoord before c
            longB4Af[i].append([longCoord[i][indexPosNeg[i][j]],longCoord[i][indexPosN
            latB4Af[i].append([latCoord[i][indexPosNeg[i][j]],latCoord[i][indexPosNeg[
            t_B4Af[i].append([t[i][indexPosNeg[i][j]],t[i][indexPosNeg[i][j]+1]])

    return longB4Af, latB4Af, t_B4Af
```

Arc Coverage Function

This function returns the longitude points at which the satellites cross the arc as well as the coverage of each satellite as it crosses over the arc. The longitude points of the crossing of the arc are calculated with an interpolated line. The swath intervals are calculated by adding and subtracting swath width from the longitudinal points of crossing. Since the projection along the track to the arc is θ-dependent, in general, the projection is not perpendicular to the arc. The outputs are in radians.

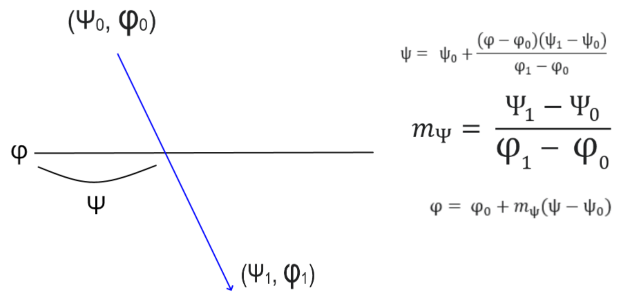
- This function accepts longB4Af, latB4Af, and halfSwath. longB4Af and latB4Af are lists of 2d vectors where the first element is the coordinate before crossing and the second element is the coordinate after crossing.

- This function returns the interLine variable (longitudinal coordinate of each satellite crossing)



- "The swath approaches the equator at an angle, so we incorporate this thought when calculating the swath projection. To do this, we first take two points straddling the equator and interpolate a line crossing the equator. The swath is imposed onto the equator its center at a point on the interpolated line crossing the equator. The line that intersects the equator at angle theta allows us to impose the additional measurement across the equator. From this, we can construct a triangle on both sides that give us the two endpoints of the swath coverage." - Fall 2019 Math Clinic Ball Final Write Up

Sources: [A Study into Satellite Coverage and Orbital Equations]()


(Ψ_0, φ_0)
$\psi = \psi_0 + \frac{(\varphi - \varphi_0)(\psi_1 - \psi_0)}{\varphi_1 - \varphi_0}$
φ
$m_\Psi = \frac{\Psi_1 - \Psi_0}{\varphi_1 - \varphi_0}$
Ψ
$\varphi = \varphi_0 + m_\psi(\psi - \psi_0)$
(Ψ_1, φ_1)

We are trying to find the linear interpolation using this method, shown in the picture. Which will give us interSlope.

"If the two known points are given by the coordinates (φ_0, ψ_0) and (φ_1, ψ_1) the linear interpolant is the straight line between these points." - Linear Interpolation (listed in Sources).

- φ - phi (latitude)
- ψ- psi (longitude)
- interSlope is m_ψ

Sources: Linear Interpolation

```
def arcCoverage(longB4Af,latB4Af,halfSwath,t_B4Af,y1):
#creating array to hold the slope at which the satellite
#crosses the arc stored in interSlope
#this is calculated by dividing latitude difference over longitude distance
    interSlope = ['']*satellites
    interLine = ['']*satellites
    interLat = ['']*satellites
    inter_t = ['']*satellites
    for i in range(0,satellites):
        interSlope[i] = [];
        interLine[i] = [];
        interLat[i] = [];
        inter_t[i] = [];
        for j in range(0,len(latB4Af[i])-1):
            m = ((latB4Af[i][j][1]-latB4Af[i][j][0])/(longB4Af[i][j][1]-longB4Af[i][j]
            interSlope[i].append(m)
            interLine[i].append((longB4Af[i][j][0]+((longB4Af[i][j][1]-longB4Af[i][j]
            interLat[i].append( arc_latitude )
        for j in range(0,len(latB4Af[i])-1):
            inter_t[i].append((t_B4Af[i][j][0]+((t_B4Af[i][j][1]-t_B4Af[i][j][0]) / (

    swathWidth= ['']*satellites;
    for i in range(0,satellites):
        swathWidth[i]=[];
        for j in range(0,np.size(interSlope[i])-1):
        #calculating the width of the swaths of each satellites path crossing the arc
            swathWidth[i].append(m2r*(halfSwath/np.sin(np.arctan(interSlope[i][j]))))
#creating variable that will hold the swath intervals of every satellite crossing of 
    swathInterval = ['']*satellites;
    for i in range(0, satellites):
        swathInterval[i]=[];
        for j in range(0,len(interLine[i])-1):
        #adding and subtracting swath width to the longitude coordinate of the crossi
        #in order to create interval
            swathInterval[i].append([interLine[i][j]+swathWidth[i][j],interLine[i][j]-

    return swathInterval, interLine, interLat, inter_t
```

Arc Interval Function

This function sorts through the interLine variable (longitudinal crossings of arc latitude) and checks if the interLine value falls between the longitudes of interest. The outputs are in radians for the variables that do not deal with time. The timeCrossings is in seconds.

- This function accepts interLine, $x1$, and $x2$. The variable interLine is an array of longitudinal crossings of the arc, and $x1$ and $x2$ represent the longitudes of interest for our arc.
- This function returns the variable arcCrossings. arcCrossings are the longitudinal crossings of the arc latitude that fall between $x1$ and $x2$.

```
# x1 must be greater value in radians
# x2 must be lesser value in radians
def arcLongInterval(interLine, x1, x2, inter_t, swathInterval):
  #creating variable for the longitudinal crossings of arc
  #that fall between x1 and x2
  arcCrossings = ['']*satellites
  latCrossings = ['']*satellites
  timeCrossings = ['']*satellites
  swathInArc= ['']*satellites
  for i in range(0,satellites):
    arcCrossings[i] = [];
    latCrossings[i] = [];
    timeCrossings[i] = [];
    swathInArc[i]= [];
    for j in range(0,np.size(interLine[i])-1):
      if(interLine[i][j] < x1 and interLine[i][j] > x2):
        # appending values that fall in our interval
        arcCrossings[i].append(interLine[i][j])
        latCrossings[i].append(arc_latitude)
        timeCrossings[i].append(inter_t[i][j])
        swathInArc[i].append(swathInterval[i][j])
  return arcCrossings, latCrossings, timeCrossings, swathInArc
```

Time to Date & Time

This function takes in the time array (t) in seconds and outputs an array of date & time in seconds since the epoch (01-01-1970 00:00:00).

```
def timeSinceStart(t):
  dtInSec = []
  for i in range(0, satellites):
    columnArray = []
    for j in range(0, len(t[i])):
      addTime = satellite_start.timestamp() + t[i][j]
                                  #adds time at satellite i and index j to the start time
      columnArray.append(addTime)     #appends sum to array for one satellite
    dtInSec.append(columnArray)       #appends array for one satellite to list of arra
  return dtInSec
```

This function converts an input of date & time in seconds since the epoch into time in hours since the first satellite started.

```
def convertDateToSecSinceStart(timeInput):
  timeResult = (timeInput - float_sat_start) / hours2seconds
  return timeResult
```

Daytime

This function determines whether the crossing of equator/arc happens during the daytime and returns only the daytime coordinates.

Daytime is determined by the angle of the sun. If the sun is more than 6 degrees below the horizon from the longitude/latitude pair, it is twilight. (Civil definition of twilight). https://www.timeanddate.com/astronomy/different-types-twilight.html

(Astronomical definition: If the sun is more than 18 degrees below the horizon from the longitude/latitude pair, it is twilight.)

Uses vernal equinox in 2020 for satellite start date (assigned as a global variable), as the equinox ensures both northern and southern hemispheres get the same amount of sunlight. Hence this is a good model for an average amount of sunlight over the year.

Utilizes Ephem package (https://pypi.org/project/ephem/) to calculate the angle of the sun. Accepts time in UTC based off of a pre-determined date, adds time t calculated by t2q function to determine the time of crossing in UTC ("absolute time").

Accepts inputs phi (latitude), psi (longitude), time, and altitude of satellite Returns coordinates of equatorial/arc coverage during the daytime.

Source for the backbone of function: https://stackoverflow.com/questions/43299500/pandas-convert-datetime-timestamp-to-whether-its-day-or-night

```
def daytimeFunction(longCoord, latCoord, dt):

  earthCoordinates = [longCoord, latCoord, dt]
  sun = ephem.Sun()
  location = ephem.Observer()     #sets the location of the observation point
  location.lon = ephem.degrees(longCoord)    #Pulls latitude coordinate, changes from r
  location.lat = ephem.degrees(latCoord)     #Pulls longitude coordinate, changes from
  location.elevation = h     #sets the elevation of the observation point to the altit
  #takes in time in seconds, will be supplied by time array
```

```
  datetimeInSec = datetime.datetime.utcfromtimestamp(dt)
  #changes the datetime object to a format read by ephem (YYYY/MM/DD HH:MM:SS)
  location.date = datetime.datetime.strftime(datetimeInSec, "%Y/%m/%d %H:%M:%S")
  sun.compute(location)     #Determines location of sun at the observation point
  altitudeOfSun = sun.alt   #Calculates altitude of sun with respect to the observatio
                            #if below a certain point, it is nighttime
                            #Note that ephem presents angles in radians
  sunDegrees = (altitudeOfSun*180/np.pi)      #Transforms angle of sun in radians into
  if (sunDegrees >= sun_day):
    daytimeCoordinates = earthCoordinates
    return daytimeCoordinates
```

Helper Function to call Daytime function

Daytime function accepts & returns a single set of points; the helper function accepts and returns arrays of points.

```
def callingDaytime(longCoord, latCoord, dtInSec):
  filledArray = []

  for i in range(0, satellites):
    columnArray = []
    for j in range(0, len(longCoord[i])):
      #call daytimeFunction (input to daytimeFunction is current ordered pair set)
      daytimeCoords = daytimeFunction(longCoord[i][j],latCoord[i][j], dtInSec[i][j])
      if not daytimeCoords:     # don't append 'None' values to array
        continue
      columnArray.append(daytimeCoords)
    filledArray.append(columnArray)   #append returned values from daytimeFunction ont

  return filledArray      #return array of daytime ordered pairs
```

Returns result of helper function with time in hours

This function accepts an ordered triplet of longitude, latitude, and date & time values and returns the same triplet, but with time changed to hours since the first satellite started.

```
def daytimeArcCrossings(longitude, latitude, dateTimeInSeconds):
  # assigns result of calling callingDaytime (returns array of ordered triples)
  daytimeCrossings = callingDaytime(longitude,latitude, dateTimeInSeconds)

  for i in range(0, satellites):
    for j in range(0, len(daytimeCrossings[i])):
      # assigns the time in hours since first satellite start to the third value in th
      daytimeCrossings[i][j][2] = convertDateToSecSinceStart(daytimeCrossings[i][j][2
  return daytimeCrossings   # returns altered daytimeCrossings array
```

Daytime Swath

This function works the same as Daytime function, but only returns swath. Two separate functions are needed because swath intervals are initiated later in the code than arc crossing points, so the Daytime function will not work if swath intervals are taken as an argument.

This function determines whether the crossing of equator/arc happens during daytime and returns only the swath interval around each daytime arc crossing point.

Daytime is determined by the angle of the sun. If the sun is more than 6 degrees below the horizon from the longitude/latitude pair, it is twilight. (Civil definition of twilight) https://www.timeanddate.com/astronomy/different-types-twilight.html

Utilizes Ephem package to calculate the angle of the sun. Accepts time in UTC based off of a pre-determined date, adds time t calculated by t2q function to determine the time of crossing in UTC ("absolute time").

- Accepts inputs latitude, longitude, date & time, and swath interval
- Returns swath interval if satellite crossed the arc at that point during the day

Source for the backbone of function: https://stackoverflow.com/questions/43299500/pandas-convert-datetime-timestamp-to-whether-its-day-or-night

```
def daySwath(long_point, lat_point, dt_point, swathInterval_point):

  earthCoordinates = [long_point, lat_point, dt_point, swathInterval_point]
  sun = ephem.Sun()
  location = ephem.Observer()        #sets the location of the observation point
  location.lon = ephem.degrees(long_point)   #Pulls latitude coordinate, changes from
  location.lat = ephem.degrees(lat_point)    #Pulls longitude coordinate, changes fror
  location.elevation = h    #sets the elevation of the observation point to the altitu
  ##takes in time in seconds, will be supplied by time array
  datetimeInSec = datetime.datetime.utcfromtimestamp(dt_point)
  # #changes the datetime object to a format read by ephem (YYYY/MM/DD HH:MM:SS)
  location.date = datetime.datetime.strftime(datetimeInSec, "%Y/%m/%d %H:%M:%S")
  sun.compute(location)     #Determine location of sun at the observation point
  altitudeOfSun = sun.alt  #Calculates altitude of sun with respect to the observatior
                            #if below a certain point, it is nighttime
                            #Note that ephem presents angles in radians
  sunDegrees = (altitudeOfSun*180/np.pi)      #Transforms angle of sun in radians intc
  if (sunDegrees >= sun_day):
    # If the angle of the sun is greater than the angle for nighttime
    # Assign the swath Interval to the variable daytimeSwath
    daytimeSwath = earthCoordinates[3]
    return daytimeSwath
```

Helper Function to call Day Swath

Daytime function accepts a single set of points and returns a single swath interval; the helper function accepts arrays of points and returns an array of swath intervals.

```
def callDaySwath(long_array, lat_array, dt_array, swath_array):
  filledArray = []

  #increment through pairs
  for i in range(0, satellites):
    columnArray = []
    for j in range(0, len(long_array[i])):
      #call daySwath (input is current pair set)
      daytimeCoords = daySwath(long_array[i][j],lat_array[i][j], dt_array[i][j], swatl
      if not daytimeCoords:     # don't append 'None' values to array
        continue
      columnArray.append(daytimeCoords)
    #append returned values from daySwath onto array
    filledArray.append(columnArray)
  #return array of daytime pairs
  return filledArray
```

Functions for Swath Coverage of the Arc

Converting 3D array to 2D Array

This function was necessary as our swathInArc was a multidimensional array, which contained an array of satellites as well as an array of order pairs of longitudes (radians) associated with each satellite. In order to perform the calculation of arc coverage, the array of satellites was not needed only the pairs of longitudes thus they were appended to a new array. An example of Global_swathInArc prior to and after the function is provided in Appendix A. The longitudinal orders are retained just at a new index.

```
# converting 3d array to 2d array (Original array was a list of arrays that contained
#pairs of longitude points associated with each satellite. Now contains only pairs of
def c3d2(input_array):
  empty=[]
  for i in range(0, len(input_array)):
    for j in range(0, len(input_array[i])):
      empty.append(input_array[i][j])
  return empty
```

Helper Function to Call Merge Interval Function

This function takes the array of pairs of longitudes (radians) and ensures that they are all ordered from smallest to largest in all cases, i.e. (3,1) converts to (1,3) and (-1, -4) converts to (-4, -1). An example of Global_swathInArc prior to and after the function is provided in Appendix A. This was to ensure that our "lower" limit is not greater than our "upper limit."

```
#Function makes sure longitude intervals are ordered from smallest to largest in all c
def merge_intervals_order_helper(input_array):
  for i in range(0, len(input_array)):
    if input_array[i][0]>input_array[i][1]:
      temp=input_array[i][0]
      input_array[i][0]=input_array[i][1]
      input_array[i][1]=temp
  return input_array
```

Merged Intervals of Longitudinal Coverage Function

This function takes the sorted array of ordered pairs of longitudes and finds the unions for the entire set of points. And creates a new array which contains ordered pairs of the unions, i.e. ((1,3), (3,6)) convert to (1,6). It also outputs those ordered pairs that lay outside the union, i.e. (8,9).

This code was taken from an example on stack overflow: https://stackoverflow.com/questions/49071081/merging-overlapping-intervals-in-python/49076157

```
#merges intervals of longitudes to calculate swath overlap between satellites
def merge_intervals(intervals):
    intervals=merge_intervals_order_helper(intervals)
    sorted_intervals = sorted(intervals, key=lambda x: x[0])
    interval_index = 0

    for  i in sorted_intervals:


        #modified from original stack overflow
        if i[0] > sorted_intervals[interval_index][1]:
            interval_index += 1
            sorted_intervals[interval_index] = i

        else:
            sorted_intervals[interval_index] = [sorted_intervals[interval_index][0],i[

    return sorted_intervals[:interval_index+1]
```

Summation of Distance Covered by Satellites

The function takes the array of merged intervals and calculates the distance between each ordered pair in a for loop and outputs the total distance in radians covered by all the satellites. Note that the

distance calculated ignores the sphericity of the Earth.

```
#summation of merged arc intervals
def sumarc(interval_array):
  sum=0
  for i in range(0, len(interval_array)):
    sum+=abs(interval_array[i][0]-interval_array[i][1])
  return sum
```

Concatenation of Swath Interval Array

This function takes an input array of swath intervals and returns the first $i + 1$ swath intervals from the input array to index i of the returned array.

```
def dayCoverageConcat(input_array):
    array_of_inputs = []
    coverage_ratio_array = []

    for i in range(0, len(input_array)):
      # append the first (i+1) intervals to the coverage ratio array
      coverage_ratio_array.append(merge_intervals(input_array[:i+1]))
    return coverage_ratio_array
```

Ratio of Coverage of the Arc

This function takes as input the concatenated array and returns an array of ratios for each input.

```
def ratioDaySwath(concat_array):
  totarc = abs(x1-x2)   # assigns the absolute value of the endpoints of the arc (leng
  return_array=[]
  for i in range(0, len(concat_array)):
    # takes the sum of the coverage of the index i of the concatenated array
    # and divides by the total length of the arc to determine coverage with i crossing
    sat_cov = (sumarc(concat_array[i]) / totarc)
    return_array.append(sat_cov)
  return return_array
```

Graphing

Ground Tracking Visualization

This figure shows the ground tracking generated as (longitude, latitude) points in radians by each satellite mapped on a flat representation of the Earth. Here the solid black line at approximately 0.69 radians represents the latitude of the arc through Denver, Colorado which is a rough approximation of the latitude of the geographic center of the contiguous United States. Thus points on either side of the line represent the satellite crossing over the arc.

Source for the backbone of this code: Spring 2020 Ball Aerospace final Jupyter Notebook

Source for changing colormap to assign colors for satellites: https://matplotlib.org/3.1.0/tutorials/colors/colormaps.html

```
def groundtrackingViz(longCoord, latCoord, satellites):
  arcLine = [arc_latitude, arc_latitude]

  f = plt.figure(figsize=(10,8))
  ax = f.add_subplot(1,1,1)

  colors = cm.viridis(np.linspace(0, 1, satellites))  # sets colormap to a value equal
  for i in range(0,satellites):
    title = "sat" + str(i+1)
    # plots logitude values with respect to latitude values
    plt.plot(longCoord[i], latCoord[i],'.', label=title, c=colors[i])
  plt.plot([-np.pi,np.pi], arcLine,'-',label = "Arc Latitude", c='k')        # plots t
  #plt.axis([x2, x1, 0.5, 1.0])       #plt.axis([min_x, max_x, min_y, max_y]) sets mir
  plt.title('Ground Tracking Projection For ' + str(i+1) + ' satellites in 24 hours')
  plt.legend(loc='lower right')
  plt.xlabel('Longitude of Earth in radians (-π to +π)')
  plt.ylabel('Latitude of Earth in radians (-π/2 to +π/2)')
  txt=" This figure shows the ground tracking generated as (longitude, latitude) point
  plt.figtext(.5, .001, txt, ha='center')

  return
```

Daytime Ground Tracking Visualization

This figure shows the ground tracking generated as (longitude, latitude) points in radians by each satellite mapped on a flat representation of the Earth during **daytime** hours. Here the solid black line at approximately 0.69 radians represents the latitude of the arc through Denver, Colorado which is a rough approximation of the latitude of the geographic center of the contiguous United States. Thus points on either side of the line represent the satellite crossing over the arc. Note the date chosen was an equinox to ensure equal amounts of sunlight on the northern and southern hemispheres.

Source for the backbone of this code: Spring 2020 Ball Aerospace final Jupyter Notebook

Source for code to prevent repeated labels in legend: https://stackoverflow.com/questions/19385639/duplicate-items-in-legend-in-matplotlib

```
def dayGroundtrackingViz(coordinates, satellites):
  arcLine = [arc_latitude, arc_latitude]
```

```
  f = plt.figure(figsize=(10,8))
  ax = f.add_subplot(1,1,1)

  colors = cm.viridis(np.linspace(0, 1, satellites))
  for i in range(0,satellites):
    for j in range(0, len(coordinates[i])):
      title = "sat" + str(i+1) if j == 0 else "_nolegend_"
      # plots longitude values with respect to latitude values
      plt.plot(coordinates[i][j][0], coordinates[i][j][1],'.', label = title, c=colors
  plt.plot([-np.pi,np.pi], arcLine,'-', label = "Arc Latitude", c='k')       #plots tl
  #plt.axis([x2, x1, 0.5, 1])       #plt.axis([min_x, max_x, min_y, max_y]) sets min/r
  plt.title('Daytime Ground Tracking Projection For ' + str(i+1) + ' satellites in 24
  plt.legend(loc='lower right')
  plt.xlabel('Longitude of Earth in radians (-π to +π)')
  plt.ylabel('Latitude of Earth in radians (-π/2 to +π/2)')
  txt=" This figure shows the ground tracking generated as (longitude, latitude) point
  plt.figtext(.5, .00000001, txt, ha='center')
  return
```

Location of Daytime Arc Crossing Visualization

This figure shows the arc longitude at which each satellite crosses the arc during daytime as related to the time since the first satellite was launched. The orange line represents sunrise and the blue represents sunset. The distance between the black dashed horizontal lines represents the length of the arc.

Source for the backbone of this code and the sunrise/sunset lines: Spring 2020 Ball Aerospace final Jupyter Notebook

```
def locationOfDaytimeCrossings(coordinates_time, satellites):
  l_i = 0 #Initial longitude location
  t_i = HOURS*hours2seconds   #Time that the satellites start in hours, transformed tc
  psi_r = (l_i + ((np.pi/2)-(deg2rad(abs(sun_day))))-t_i*radm)%(2*np.pi)    #initial s
  psi_s = (l_i + ((3*np.pi/2)+(deg2rad(abs(sun_day))))-t_i*radm)%(2*np.pi)  #initial s

  f = plt.figure(figsize=(10,8))
  ax = f.add_subplot(1,1,1)
  for i in range(0,satellites):
    plt.plot(t[i]/3600, (psi_r-t[i]*radm)%(2*np.pi), '.', label="sunrise" if i == 0 el
    plt.plot(t[i]/3600, (psi_s-t[i]*radm)%(2*np.pi), '.', label="sunset" if i == 0 els

  colors = cm.viridis(np.linspace(0, 1, satellites))
  k=0
  for i in range(0,satellites):
    for j in range(0, len(coordinates_time[i])):
      title = "sat" + str(i+1) if j == 0 else "_nolegend_"
      # plots time values with respect to longitude values (from 0 to 2*pi)
      plt.plot(coordinates_time[i][j][2], (coordinates_time[i][j][0] % (2*np.pi)), '.'
      k+=k
  eastern_boundary = (x1 % (2*np.pi))
  western_boundary = (x2 % (2*np.pi))

  title1="Boundary of Arc Longitudes"
  #https://matplotlib.org/3.1.0/gallery/subplots_axes_and_figures/axhspan_demo.html#sp
  plt.axhline(y=eastern_boundary, linestyle='--', color='black', label=title1)
  plt.axhline(y=western_boundary, linestyle='--', color='black')

  plt.xlim([0, 24])
  plt.ylim([0, 2*np.pi])
  plt.xlabel('Time since Satellite 1 started in hours')
  plt.ylabel('Arc Longitude in radians (0 to 2π)')
  ax.set_title('Location of Daytime Arc Crossings for ' + str(i+1) + ' Satellites')
  ax.legend(loc='lower right')
  txt=" This figure shows the arc longitude at which each satellite crosses the arc du
  plt.figtext(.5, .0000001, txt, ha='center')
  return
```

Location of Daytime Arc Crossing Visualization, Bounded by Arc Endpoint Longitudes

This is the same code as the function locationOfDaytimeCrossings, but the graph is bounded by the endpoint longitudes of the arc to better show the coverage of the arc.

This figure shows the arc longitude at which each satellite crosses the arc during daytime as related to the time since the first satellite was launched. The black error bars show the swath coverage at each crossing point. The orange line represents sunrise and the blue represents sunset.

Source for the backbone of this code as well as the sunrise/sunset lines: Spring 2020 Ball Aerospace final Jupyter Notebook

```
def zoomedInlocationOfDaytimeCrossings(coordinates_time, satellites):
  l_i = 0                     #Initial longitude location
  t_i = HOURS*hours2seconds   #Time that the satellites start in hours, transformed to
  psi_r = (l_i + ((np.pi/2)-(deg2rad(abs(sun_day))))-t_i*radm)%(2*np.pi)    #initial s
  psi_s = (l_i + ((3*np.pi/2)+(deg2rad(abs(sun_day))))-t_i*radm)%(2*np.pi)  #initial s

  f = plt.figure(figsize=(10,8))      #Sets size of figure
  ax = f.add_subplot(1,1,1)
  for i in range(0,satellites):       #Iterates through satellites
    plt.plot(t[i]/3600, (psi_r-t[i]*radm)%(2*np.pi),'.', label="sunrise" if i == 0 els
    plt.plot(t[i]/3600, (psi_s-t[i]*radm)%(2*np.pi),'.', label="sunset" if i == 0 else
  colors = cm.viridis(np.linspace(0, 1, satellites))
  k=0
  for i in range(0,satellites):
    for j in range(0, len(coordinates_time[i])):
```

```
      title = "sat" + str(i+1) if j == 0 else "_nolegend_"
      # plots swath coverage of arc
      plt.errorbar(coordinates_time[i][j][2], (coordinates_time[i][j][0] % (2*np.pi)),
      # plots time with respect to longitude (from 0 to 2*pi)
      plt.plot(coordinates_time[i][j][2], (coordinates_time[i][j][0] % (2*np.pi)), 'o'
      k += k
  plt.xlim([0, 24])
  plt.ylim([x2%(2*np.pi), x1%(2*np.pi)]) # graph is bounded by arc endpoint longitudes
  plt.xlabel('Time since Satellite 1 started in hours')
  plt.ylabel('Arc Longitude in radians (0 to 2π) \n Bounded by Arc Endpoints')
  ax.set_title('Location of Daytime Arc Crossings for ' + str(i+1) + ' Satellites\n Bo
  ax.legend(loc='lower right')
  txt=" This figure shows the longitude (bounded by arc) at which each satellite cross
  plt.figtext(.5, .0000001, txt, ha='center')
  return
```

Visualization for Coverage Ratio

This figure shows the arc longitude at which each satellite crosses the arc during daytime as related to the time since the first satellite was launched. The right axis shows the accumulated coverage of the arc by the swath intervals (as a percentage) over the 24 hour period. The black error bars show the swath coverage at each crossing point.

The purple dashed line in the upper right corner represents full coverage of the arc by the swath intervals.

Source for the backbone of this code: Spring 2020 Ball Aerospace final Jupyter Notebook

```
def coverageRatioViz(coordinates_time, satellites, ratio_array):
  f = plt.figure(figsize=(10,8))
  ax1 = f.add_subplot(1,1,1)
  k=0
  colors = cm.viridis(np.linspace(0, 1, satellites))
  x_array = []
  for i in range(0,satellites):
    #title = "sat" + str(i+1) + "-orbit"
    for j in range(0, len(coordinates_time[i])):
      x_array.append(coordinates_time[i][j][2])
      #only prints the first entry for each satellite in the legend
      title = "sat" + str(i+1) if j == 0 else "_nolegend_"
      # plots swath coverage of arc
      plt.errorbar(coordinates_time[i][j][2], (coordinates_time[i][j][0] % (2*np.pi)),
      # plots time values with respect to longitude (from 0 to 2*pi)
      plt.plot(coordinates_time[i][j][2], (coordinates_time[i][j][0] % (2*np.pi)), 'o'
      k +=k
  x_array.sort()
  ax1.legend(loc= 'lower right')
  plt.xlabel('Time since Satellite 1 started in hours')
  plt.ylabel('Arc Longitude in radians (0 to 2π) \n Bounded by Arc Endpoints')
```

```
    ax2 = ax1.twinx()
    title1 = "Cumulative Coverage of All Satellites"
    title2 = "Full Arc Coverage during Daytime (righthand y-axis)"
    title3 = "Cumulative coverage"

    # plots the ratio of coverage for each successive swath interval
    ax2.plot(x_array, ratio_array, color='blue', label = title3)
    ax2.yaxis.label.set_color('blue')
    ax2.hlines(y=(1), xmin=10, xmax=24, color='purple', linestyles='--', lw=2, label =ti
    ax2.legend(loc='upper right')

    plt.ylim((0, 1.2))
    plt.ylabel('Cumulative Arc coverage in percentage')
    ax1.set_title('Daytime Coverage of Arc Crossings within 24 hours with ' + str(i+1) +
    ax2.spines['right'].set_color('blue')
    txt=" This figure shows the arc longitude at which each satellite crosses the arc du
    plt.figtext(.5,.0000001, txt, ha='center')
    return
```

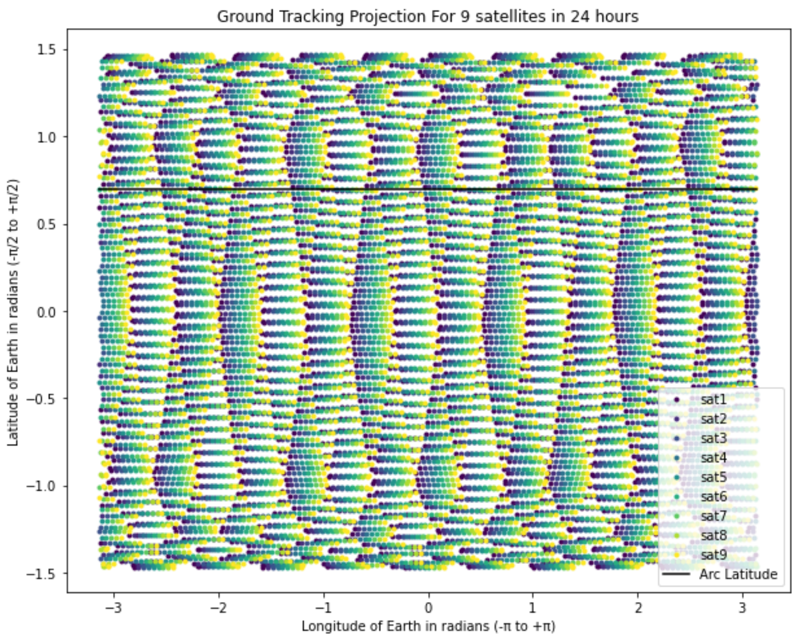
Results & Discussion

Main

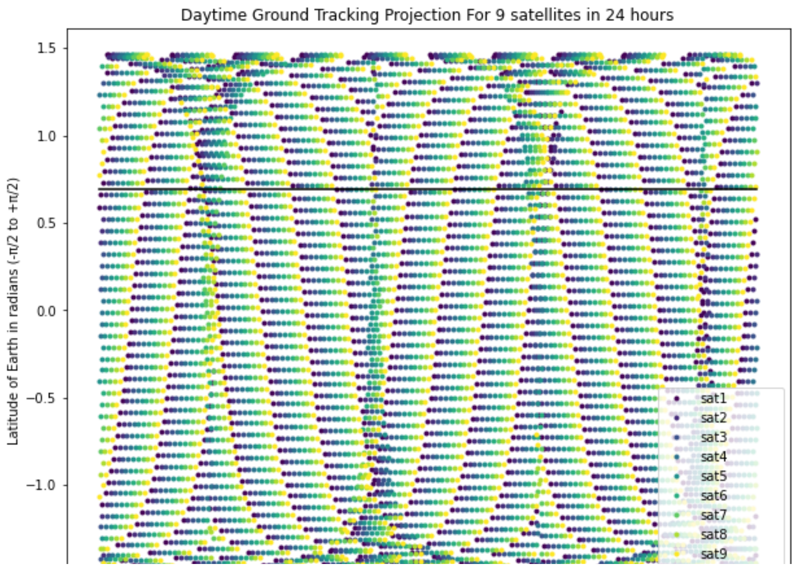
Contains some global variables and function calls.

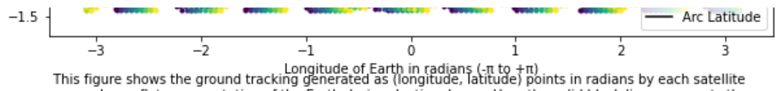
```
## calling code
Global_orbit3D = [0]*satellites                    # Creating empty lists for the x,y,
t = [0]*satellites
E = [0]*satellites
# Ω, the longitude of the ascending node will rotate by one degree every day.
#    This ensures that the orbital plane passes through the same time on each
#                                                   day at each latitude.

Ω = (np.pi/4) - 0.25
ω = 0

#spacer=.1021569                      #value used in Spring 2020
spacer = 1 / satellites

for i in range(0,satellites):         # Creating individual times lists for each of th
    t[i]= np.linspace((i*(spacer*P)), ((orbits*P)+(i*(spacer*P))), 4*N) #***in linspac
    E[i] = t2E(e, H*t[0]/a/b)
    Global_orbit3D[i] = sat3D(t[i], inclination, ω, Ω, a, b, H, E[i])


# displays all graphs without scroll bar
#https://stackoverflow.com/questions/55546869/google-colaboratory-is-there-any-way-to-
display(Javascript('''google.colab.output.setIframeHeight(0, true, {maxHeight: 5000})'
```

```
longCoord, latCoord = angularEarthCoord(Global_orbit3D)
# r is an array of the long and lat Coordinates of the satellites, r will be used in t
r = ['']*satellites
for i in range(0, satellites):
    r[i]=[]
    for j in range(0, np.size(latCoord[i])-1):
        r[i].append([longCoord[i][j],latCoord[i][j]])

#creating variable to hold values of s, d
ss = ['']*satellites
dd = ['']*satellites
# running a for loop in order to call dispArc function
for i in range(0,satellites):
    ss[i]=[]
    dd[i]=[]
    for j in range(0, np.size(latCoord[i])-1):
        d, s, e = dispArc([x1,y1],[x2,y1],r[i][j])
        # appending d values to dd
        # appending s values to ss
        dd[i].append(d)
        ss[i].append(s)
        #arcLength = c

# Function calls assigned to variables
Global_longB4Af, Global_latB4Af, Global_t_B4Af = arcCrossing(dd,longCoord,latCoord)
Global_swathInterval, Global_interLine, Global_interLat, Global_inter_t = arcCoverage(
Global_arcCrossings, Global_latCrossings, Global_timeCrossings, Global_swathInArc = ar
Global_ratio_array = ratioDaySwath(dayCoverageConcat(c3d2(callDaySwath(Global_arcCross
Global_SortedSwath = c3d2(callDaySwath(Global_arcCrossings, Global_latCrossings, timeS
Global_SortedSwathDiff = [qqq[1]-qqq[0] for qqq in Global_SortedSwath]

# Plots
groundtrackingViz(longCoord, latCoord, satellites)
dayGroundtrackingViz(callingDaytime(longCoord, latCoord, timeSinceStart(t)), satellite
locationOfDaytimeCrossings(daytimeArcCrossings(Global_arcCrossings, Global_latCrossing
zoomedInlocationOfDaytimeCrossings(daytimeArcCrossings(Global_arcCrossings, Global_lat
coverageRatioViz(daytimeArcCrossings(Global_arcCrossings, Global_latCrossings, timeSin
```

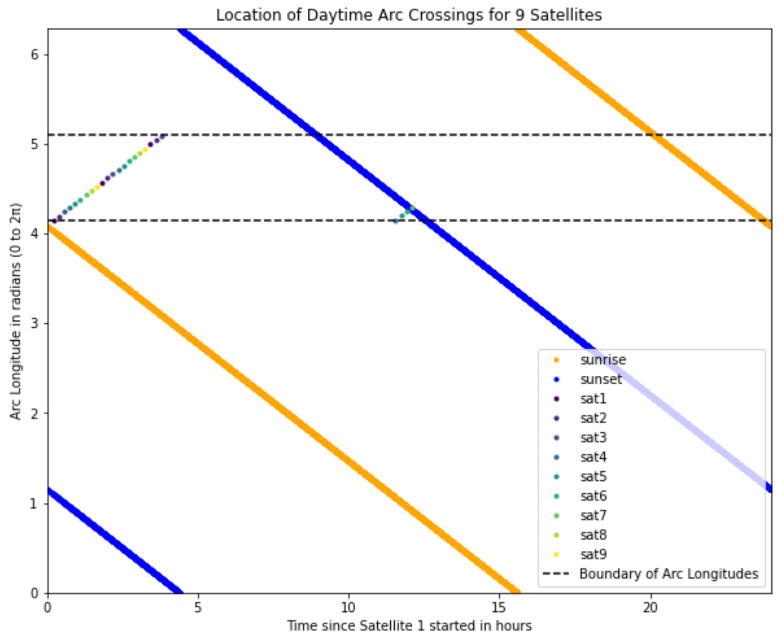


This figure shows the ground tracking generated as (longitude, latitude) points in radians by each satellite mapped on a flat representation of the Earth. Here the solid black line represents the latitude of the arc. Thus points on either side of the line represent the satellite crossing over the arc.

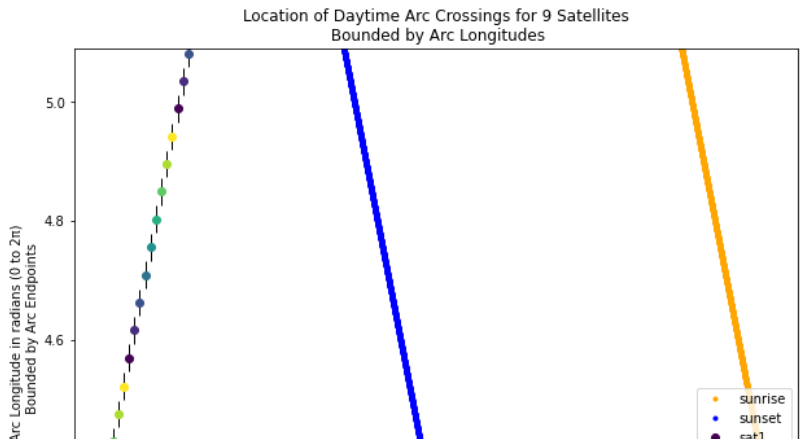


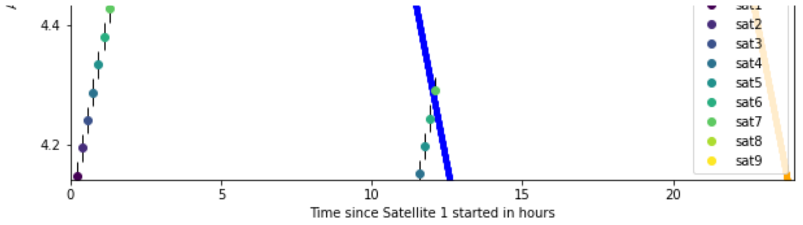


This figure shows the ground tracking generated as (longitude, latitude) points in radians by each satellite mapped on a flat representation of the Earth during daytime hours. Here the solid black line represents the latitude of the arc. Thus points on either side of the line represent the satellite crossing over the arc. Note the date chosen was an equinox to ensure equal amounts of sunlight on the northern and southern hemisphere.

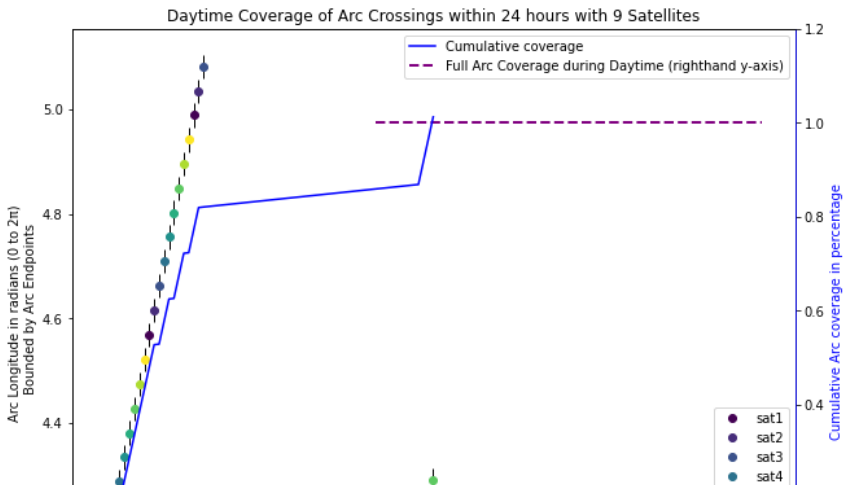


This figure shows the arc longitude at which each satellite crosses the arc during daytime as related to the time since the first satellite was launched. The orange line represents sunrise and the blue sunset. The distance between the black dashed horizontal lines represents the length of the arc.





This figure shows the longitude (bounded by arc) at which each satellite crosses the arc during daytime as related to the time since the first satellite was launched. The error bars show the swath coverage at each crossing point.



Total Percentage of Swath Coverage Over Arc

Total percentage swath coverage over arc with 5-9 satellites (note that satellites are equidistant): Generated with the last value from Global_ratio_array, which sums up the total coverage from all daytime swath intervals over the arc.

- 5 satellites: 0.641035857722568
- 6 satellites: 0.6812359836317422
- 7 satellites: 0.8137610859271983
- 8 satellites: 0.8860558939558417
- 9 satellites: 1.0126931051673647

Note that the global ratio array does not calculate the amount of overlap, so it is not a good quantification for coverage over 100%.

```
Global_ratio_array

    [0.04791394792106863,
     0.09602351701016776,
     0.1440362485777575,
     0.1919501964988261,
     0.24005976558792572,
     0.28782797014170736,
     0.335741918062776,
     0.3838514871518751,
     0.4316196917056567,
     0.47953363962672535,
     0.5276432087158249,
     0.5289443609045849,
     0.5768583088256535,
     0.6249678779147531,
     0.6262690301035125,
     0.6741829780245812,
     0.7222925471136803,
     0.7235936993024398,
     0.7715076472235084,
     0.8196172163126075,
     0.8209183685013675,
     0.8688323164224361,
     0.9169418855115352,
     0.9648558334326038,
     1.0126931051673647]
```

Results

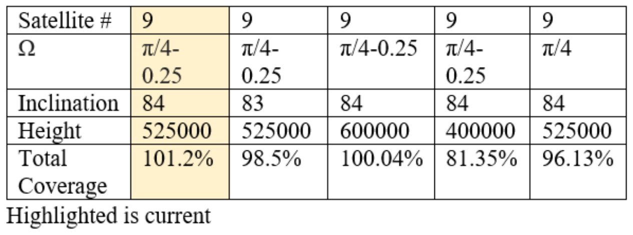
Our graphs display that 9 satellites have full coverage of the contiguous United States within 24 hours during an equinox. Since during half the year there will be more sun on the northern hemisphere than on the day of the equinox, this full-coverage case holds for a minimum of half the year.

The prior cell shows that the total percentage of swath coverage over the arc is approximately 88% for 8 satellites, but jumps up to 101% for 9 satellites. Since the function does not test for overlap, any values greater than 100% lose accuracy. We chose to merge intervals together, but recommend that prior semesters calculate the gaps & overlap in the coverage. The additional 1% could be due to swath intervals that start within the bounds of the arc but have sections outside of the arc as well. In future semesters, it would be beneficial to determine the amount of coverage versus the amount of overlap and determine the most optimal amount of satellites. Because while 9 satellites may offer full coverage, 88% may be acceptable for Ball Aerospace and have a significant impact on budget or other restrictions.

The code this semester has been encapsulated into functions. Encapsulation helps with maintainability, it makes it much easier to read and understand with minimum time investment. Future semesters will be able to take key functions from our code that they would like to use without having to recode it themselves. Encapsulation also reduces the complexity of the code and inconsistencies in variable names.

Choice of Variables

The table below shows that changes in the longitude of the ascending node (Ω), inclination (i), and altitude (h) affect total coverage of the arc during the daytime. The yellow highlighted column includes the variables used in the notebook prior to this point.



Satellite #	9	9	9	9	9
Ω	π/4-0.25	π/4-0.25	π/4-0.25	π/4-0.25	π/4
Inclination	84	83	84	84	84
Height	525000	525000	600000	400000	525000
Total Coverage	101.2%	98.5%	100.04%	81.35%	96.13%

Highlighted is current

The longitude of the ascending node (Ω) was chosen to be $\frac{\pi}{4} - 0.25$ so that the first satellite crosses the arc during daylight at approximately the time the satellites appear in the sky. This ensures the maximum number of arc crossings during an uninterrupted daytime period. When Ω is changed to $\frac{\pi}{4}$ (the 5th column), total coverage is 5% lower due to fewer satellite crossings during the day.

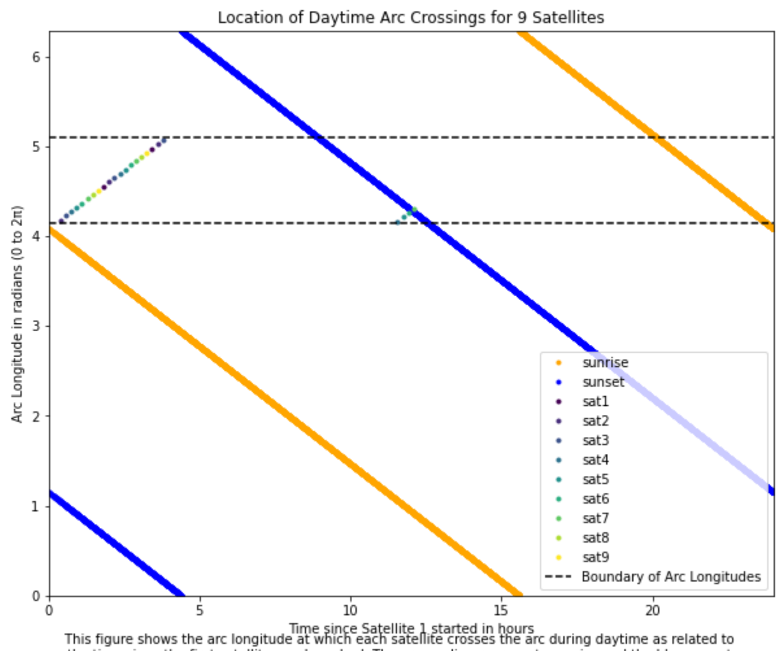
The inclination was chosen to be 84° to maximize coverage in the satellite's near-polar orbit. Polar orbits are around 90°, so we tested various inclinations around that point and found that 84° provided the most coverage. When the inclination is changed to 83° (the 2nd column), the amount of coverage decreases by several percent.

The altitude was chosen to be 525km to give approximately a 96 minute orbit (15 revolutions in 24 hours) while providing full coverage with 9 satellites. The altitude was restricted by the CIRiS instruments on the satellites which require an altitude between 400km to 600km. At the minimum altitude for the satellite, there is far less arc coverage at each crossing point, so even though there are more daytime crossings, there is less arc coverage. At the maximum altitude for the satellite, there is more arc coverage at each crossing point, but there is less coverage due to the longer period.

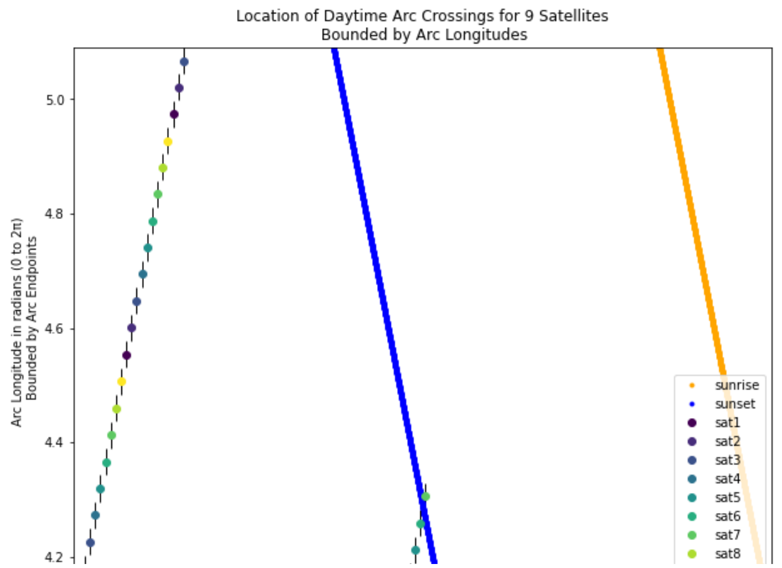
Inclination ($i = 83°$)

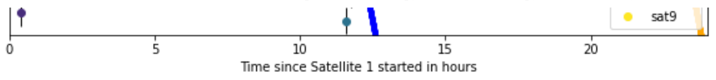
Here is the graph for the first simulation with 9 satellites, Ω is $\frac{\pi}{4} - 0.25$, inclination is 83°, altitude is 525 km:

This simulation gave about 98% arc coverage due to fewer arc crossings during the daytime. There are 24 crossings with 83° inclination and 25 crossings with 84° inclination.

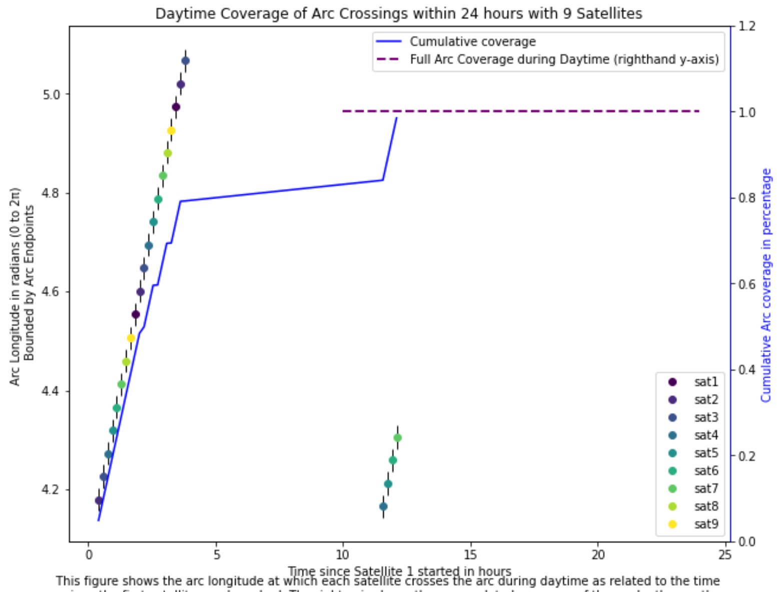


This figure shows the arc longitude at which each satellite crosses the arc during daytime as related to the time since the first satellite was launched. The orange line represents sunrise and the blue sunset. The distance between the black dashed horizontal lines represents the length of the arc. Tested with inclination of 83, height of 525 km, Ω = (π/4)-0.25.





This figure shows the longitude (bounded by arc) at which each satellite crosses the arc during daytime as related to the time since the first satellite was launched. The error bars show the swath coverage at each crossing point. Tested with inclination of 83, height of 525 km, Ω = (π/4)-0.25.

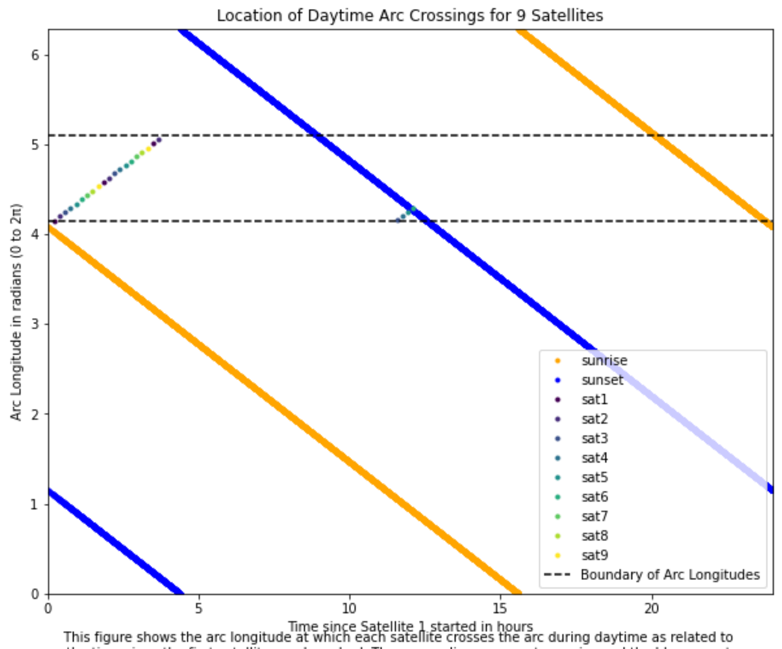


This figure shows the arc longitude at which each satellite crosses the arc during daytime as related to the time since the first satellite was launched. The right axis shows the accumulated coverage of the arc by the swath intervals (as a percentage) over the 24 hour period. The error bars show the swath coverage at each crossing point. Tested with inclination of 83, height of 525 km, Ω = (π/4)-0.25.

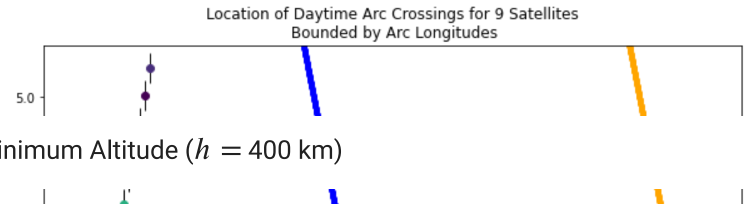
Maximum Altitude (h = 600 km)

See below for the graphs for the second simulation with 9 satellites. Ω is $\frac{\pi}{4} - 0.25$, the inclination is 84°, and the altitude is 600 km, which is the maximum for the CIRiS instruments on the satellite.

This simulation gave a little over 100% coverage. This simulation was tested with the maximum altitude of the satellite. Due to the higher altitude, there was more coverage of the arc with each crossing. However, there was less overall coverage due to the longer period of the satellite and thus fewer instances where the satellite crossed over the arc during the daytime.



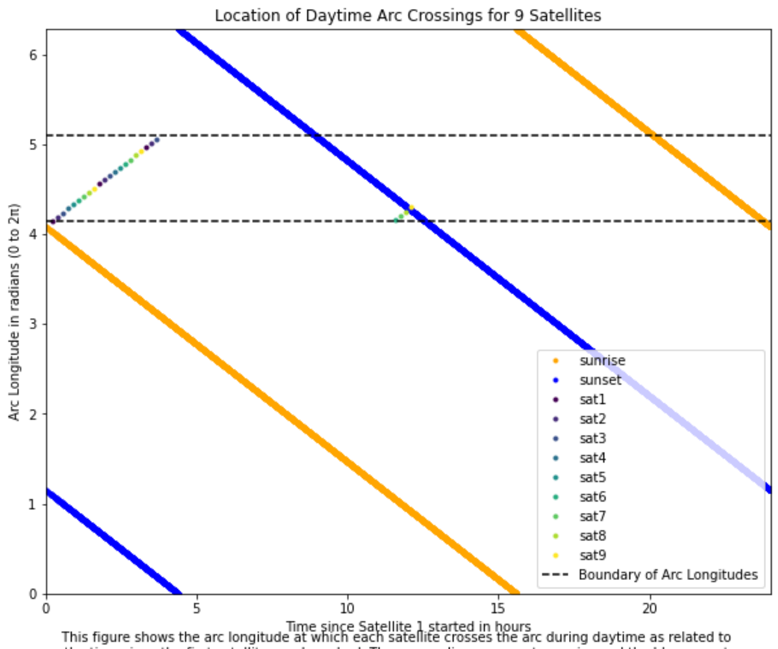
This figure shows the arc longitude at which each satellite crosses the arc during daytime as related to the time since the first satellite was launched. The orange line represents sunrise and the blue sunset. The distance between the black dashed horizontal lines represents the length of the arc.
Tested with inclination of 84, height of 600 km, Ω = (π/4)-0.25.



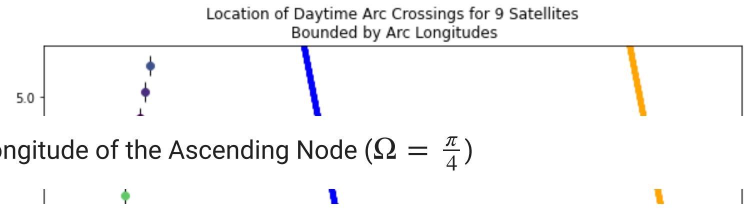
Minimum Altitude (h = 400 km)

See below for graphs for the third simulation with 9 satellites. Ω is $\frac{\pi}{4} - 0.25$, the inclination is 84°, and the altitude is 400km, which is the minimum possible for the CIRiS instruments on the satellite.

At an altitude of 400km, there is only about 81% coverage because there is less coverage of the arc at each crossing point. Although there was also a shorter period and thus more instances of the satellite crossing over the arc, the additional crossings did not make up for the smaller ground tracking projection.



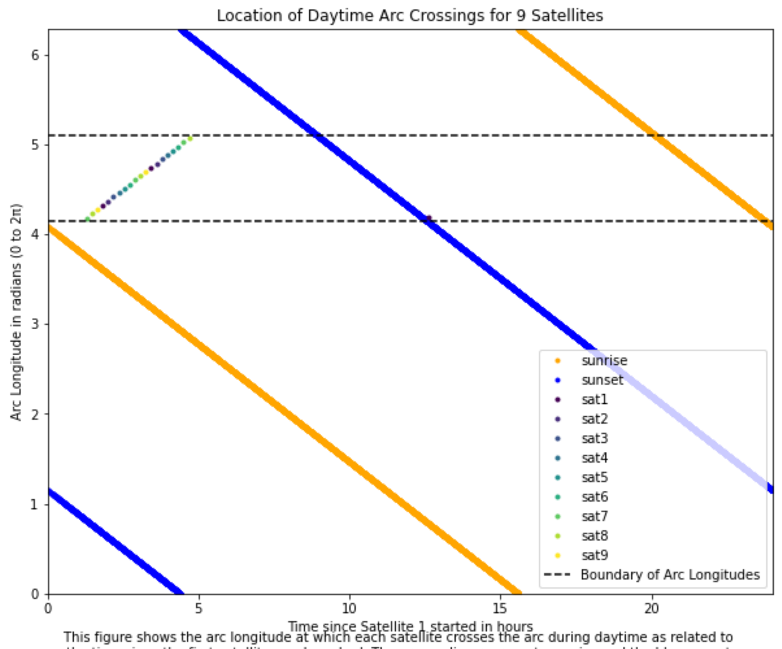
This figure shows the arc longitude at which each satellite crosses the arc during daytime as related to the time since the first satellite was launched. The orange line represents sunrise and the blue sunset. The distance between the black dashed horizontal lines represents the length of the arc. Tested with inclination of 84, height of 400 km, Ω = (π/4)-0.25.



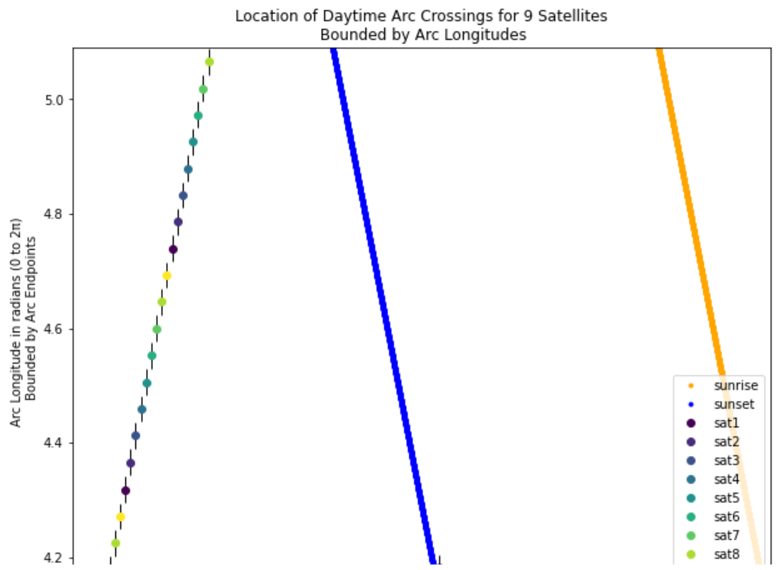
Longitude of the Ascending Node ($\Omega = \frac{\pi}{4}$)

Here is the graph for the last simulation with 9 satellites. Ω is $\frac{\pi}{4}$, the inclination is 84° and the altitude is 525 km:

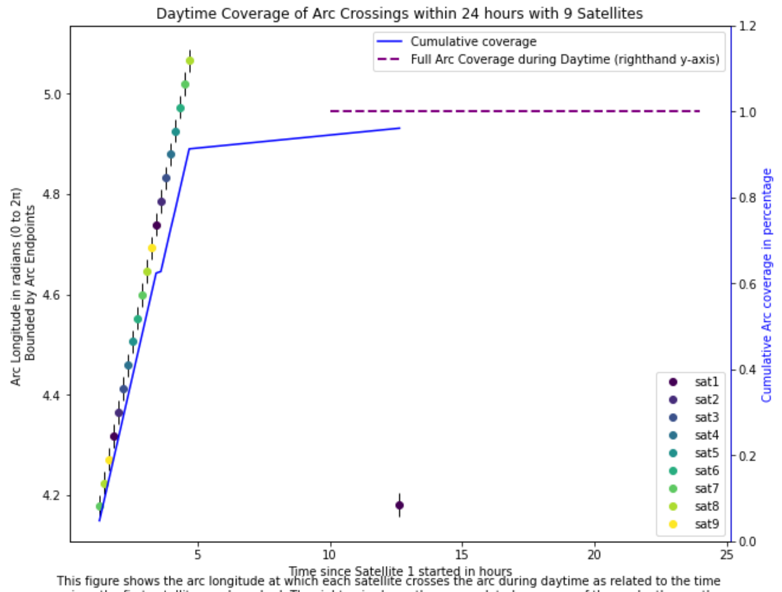
This simulation gives about 96% coverage because there are fewer daytime arc crossings. There are 21 crossings with the changed Ω, whereas with the original Ω ($\frac{\pi}{4} - 0.25$), there are 25 crossings. In the first plot "Location of Daytime Arc Crossings for 9 Satellites," it is clear that the first satellite does not cross the arc at sunrise (when time in hours is 0) and there are also fewer crossing points before sunset, which is why there is less coverage.



This figure shows the arc longitude at which each satellite crosses the arc during daytime as related to the time since the first satellite was launched. The orange line represents sunrise and the blue sunset. The distance between the black dashed horizontal lines represents the length of the arc. Tested with inclination of 84, height of 525 km, Ω = (π/4).



This figure shows the longitude (bounded by arc) at which each satellite crosses the arc during daytime as related to the time since the first satellite was launched. The error bars show the swath coverage at each crossing point. Tested with inclination of 84, height of 525 km, Ω = (π/4).



This figure shows the arc longitude at which each satellite crosses the arc during daytime as related to the time since the first satellite was launched. The right axis shows the accumulated coverage of the arc by the swath intervals (as a percentage) over the 24 hour period. The error bars show the swath coverage at each crossing point. Tested with inclination of 84, height of 525 km, Ω = (π/4).

Conclusion

Our notebook produces the groundwork to allow future students to optimize coverage of satellites for defined areas of the world such as the United States. It builds upon the work of previous semesters which demonstrated that the minimum number of satellites need to achieve full daytime coverage within 24 hours was 10 satellites. According to our calculations, which were centered on producing an arc as a means of calculating coverage as opposed to the equator itself, optimal coverage is achieved from 9 satellites. Ball Aerospace anticipated that 8 satellites would be ideal, but we are unsure of the root cause of the discrepancy. We chose our arc to be the furthest most eastern and western points of the contiguous United States which also could lead to more areas being covered than necessary to achieve satisfactory coverage of the U.S. For future students, we would suggest initially considering equidistant satellites, then optimized satellite spacing. Proof-of-

concept with graphs that show global coverage are needed to validate bounded sections of the world. Finally, we would suggest improving further on our concepts of optimization by exploring packages that may assist with the parameters that may have caused our over-coverage. Much like the previous semester, our takeaway is a deeper understanding of orbital mechanics and the two-body problem, along with enhanced coding skills and interdisciplinary collaboration.

Appendix A

Additional Coding Examples

Examples of c3d2 and merge_interval_helper functions

```
#This shows the Swath in Arc prior to being converted to a 2d array
Global_swathInArc

    [[[-2.158590465858506, -2.1130768904547903],
      [-1.7371320696178643, -1.6914326731919198],
      [-1.316827193573757, -1.2712197833444914],
      [-1.8299151356525913, -1.8755724159133302],
      [-1.4086319586908271, -1.454112240349865]],
     [[-2.111840922833162, -2.0663273474294463],
      [-1.6903825265925203, -1.6446831301665754],
      [-1.2703377558623064, -1.2249626224368544],
      [-1.783165592627248, -1.828822872887987],
      [-1.3618824156654832, -1.407362697324521]],
     [[-2.0650913798078183, -2.019577804404103],
      [-1.6436329835671766, -1.5979335871412321],
      [-1.2235882128369626, -1.1782130794115107],
      [-1.736416049601904, -1.782073329862643],
      [-1.3151328726401392, -1.360613154299177]],
     [[-2.018341836782475, -1.9728282613787593],
      [-1.5968834405418328, -1.551184044115888],
      [-2.1553522846940707, -2.110004194500619],
      [-1.6896665065765601, -1.735323786837299],
      [-1.2683833296147955, -1.3138636112738333]],
     [[-1.9715922937571309, -1.9260787183534154],
      [-1.5501338975164891, -1.5044345010905442],
      [-2.1088804189377477, -2.063310697755375],
      [-1.6429169635512162, -1.688574243811955],
      [-1.2216337865894522, -1.26711406824849]],
     [[-1.9248427507317871, -1.8793291753280716],
      [-1.5033843544911454, -1.457684958065201],
      [-2.0621308759124037, -2.016561154730031],
      [-1.5961674205258722, -1.6418247007866111],
      [-1.1748842435641083, -1.220364525223146]],
     [[-1.8780932077064434, -1.832579632302728],
      [-1.4566348114658016, -1.4109354150398572],
       [-2.01538133288706, -1.9698116117046869],
       [-1.549417877500529, -1.5950751577612678]],
      [[-1.8313436646810997, -1.7858300892773842],
       [-1.409885268440458, -1.3641858720145135],
       [-1.9230620686793443, -1.9686317898617176],
       [-1.502668334475185, -1.548325614735924]],
      [[-1.784594121655756, -1.7390805462520404],
       [-1.363911047649211, -1.3184703071409405],
       [-1.8763125256540003, -1.9218822468363737],
       [-1.4559187914498408, -1.5015760717105797]]]
```

```
#This shows the converted array
print("The following is the converted array")
c3d2(Global_swathInArc)
#Please note that the array contains the same longitudinal pairs just
#at a new index.
#Also note that the fourth ordered pair
#(-1.8299151356525913, -1.8755724159133302) which has a "lower" limit
#greater than its "upper limit"
```

```
The following is the converted array
[[-2.158590465858506, -2.1130768904547903],
 [-1.7371320696178643, -1.6914326731919198],
 [-1.316827193573757, -1.2712197833444914],
 [-1.8299151356525913, -1.8755724159133302],
 [-1.4086319586908271, -1.454112240349865],
 [-2.111840922833162, -2.0663273474294463],
 [-1.6903825265925203, -1.6446831301665754],
 [-1.2703377558623064, -1.2249626224368544],
 [-1.783165592627248, -1.828822872887987],
 [-1.3618824156654832, -1.407362697324521],
 [-2.0650913798078183, -2.019577804404103],
 [-1.6436329835671766, -1.5979335871412321],
 [-1.2235882128369626, -1.1782130794115107],
 [-1.736416049601904, -1.782073329862643],
 [-1.3151328726401392, -1.360613154299177],
 [-2.018341836782475, -1.9728282613787593],
 [-1.5968834405418328, -1.551184044115888],
 [-2.1553522846940707, -2.110004194500619],
 [-1.6896665065765601, -1.735323786837299],
 [-1.2683833296147955, -1.3138636112738333],
 [-1.9715922937571309, -1.9260787183534154],
 [-1.5501338975164891, -1.5044345010905442],
 [-2.1088804189377477, -2.063310697755375],
 [-1.6429169635512162, -1.688574243811955],
 [-1.2216337865894522, -1.26711406824849],
 [-1.9248427507317871, -1.8793291753280716],
 [-1.5033843544911454, -1.457684958065201],
 [-2.0621308759124037, -2.016561154730031],
 [-1.5961674205258722, -1.6418247007866111],
 [-1.1748842435641083, -1.220364525223146],
 [-1.8780932077064434, -1.832579632302728],
 [-1.4566348114658016, -1.4109354150398572],
 [-2.01538133288706, -1.9698116117046869],
 [-1.549417877500529, -1.5950751577612678],
```

```
 [-1.8313436646810997, -1.7858300892773842],
 [-1.409885268440458, -1.3641858720145135],
 [-1.9230620686793443, -1.9686317898617176],
 [-1.502668334475185, -1.548325614735924],
 [-1.784594121655756, -1.7390805462520404],
 [-1.363911047649211, -1.3184703071409405],
 [-1.8763125256540003, -1.9218822468363737],
 [-1.4559187914498408, -1.5015760717105797]]
```

```
#This shows the sorted array
print("The following is the sorted array")
sorted_swathInArc =merge_intervals_order_helper(c3d2(Global_swathInArc))
#note that the "lower" limit and "upper" limits have been corrected
#in the fourth ordered pair.
sorted_swathInArc
```

```
The following is the sorted array
[[-2.158590465858506, -2.1130768904547903],
 [-1.7371320696178643, -1.6914326731919198],
 [-1.316827193573757, -1.2712197833444914],
 [-1.8755724159133302, -1.8299151356525913],
 [-1.454112240349865, -1.4086319586908271],
 [-2.111840922833162, -2.0663273474294463],
 [-1.6903825265925203, -1.6446831301665754],
 [-1.2703377558623064, -1.2249626224368544],
 [-1.828822872887987, -1.783165592627248],
 [-1.407362697324521, -1.3618824156654832],
 [-2.0650913798078183, -2.019577804404103],
 [-1.6436329835671766, -1.5979335871412321],
 [-1.2235882128369626, -1.1782130794115107],
 [-1.782073329862643, -1.736416049601904],
 [-1.360613154299177, -1.3151328726401392],
 [-2.018341836782475, -1.9728282613787593],
 [-1.5968834405418328, -1.551184044115888],
 [-2.1553522846940707, -2.110004194500619],
 [-1.735323786837299, -1.6896665065765601],
 [-1.3138636112738333, -1.2683833296147955],
 [-1.9715922937571309, -1.9260787183534154],
 [-1.5501338975164891, -1.5044345010905442],
 [-2.1088804189377477, -2.063310697755375],
 [-1.688574243811955, -1.6429169635512162],
 [-1.26711406824849, -1.2216337865894522],
 [-1.9248427507317871, -1.8793291753280716],
 [-1.5033843544911454, -1.457684958065201],
 [-2.0621308759124037, -2.016561154730031],
 [-1.6418247007866111, -1.5961674205258722],
 [-1.220364525223146, -1.1748842435641083],
 [-1.8780932077064434, -1.832579632302728],
 [-1.4566348114658016, -1.4109354150398572],
 [-2.01538133288706, -1.9698116117046869],
 [-1.5950751577612678, -1.549417877500529],
 [-1.8313436646810997, -1.7858300892773842],
 [-1.409885268440458, -1.3641858720145135],
 [-1.9686317898617176, -1.9230620686793443],
 [-1.548325614735924, -1.502668334475185],
 [-1.784594121655756, -1.7390805462520404],
[-1.363911047649211, -1.3184703071409405],
[-1.9218822468363737, -1.8763125256540003],
[-1.5015760717105797, -1.4559187914498408]]
```

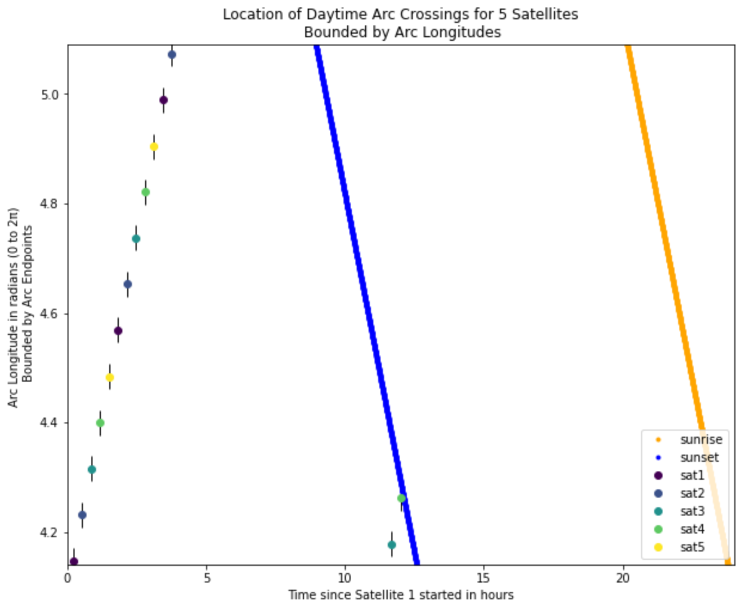
Appendix B (5 satellites)

The sponsor asked for the coverage of the contiguous United States with 5 satellites.

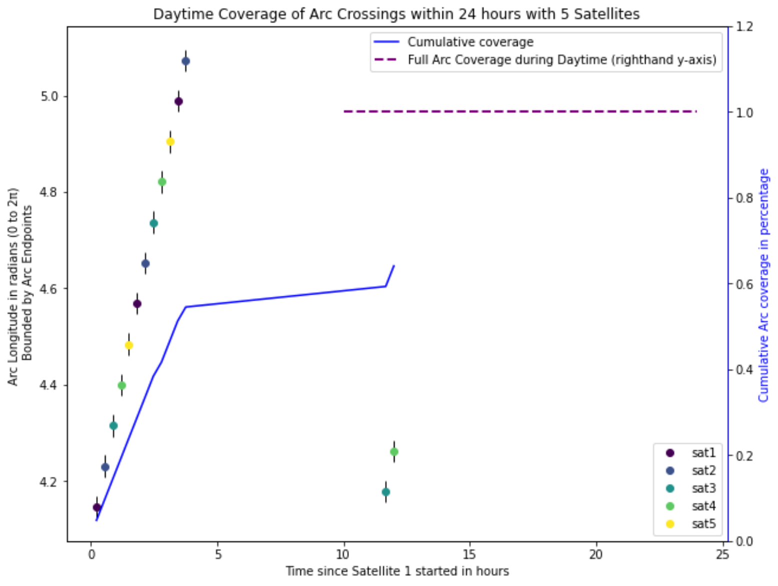
Cumulative percent coverage over the arc with 5 satellites: 0.641035837722568

(Note that the satellites are equally spaced to maximize coverage and minimize overlap)

The graphs below show the locations of daytime arc crossings and the cumulative coverage of the arc for daytime arc crossings with 5 satellites.



This figure shows the longitude (bounded by arc) at which each satellite crosses the arc during daytime as related to the time since the first satellite was launched.



This figure shows the arc longitude at which each satellite crosses the arc during daytime as related to the time since the first satellite was launched.
The right axis shows the accumulated coverage of the arc by the swath intervals (as a percentage) over the 24 hour period.

Appendix C (West of the Mississippi River)

The sponsor asked for the coverage for the western US (west of the Mississippi River). Wickliffe, Kentucky (36.966600˚N, 89.086822˚W) is on the eastern bank of the Mississippi River at what appears to be the eastern-most point of the river. Thus the longitude of Wickliffe (-89.086822˚) is used as the eastern longitude (variable x1).

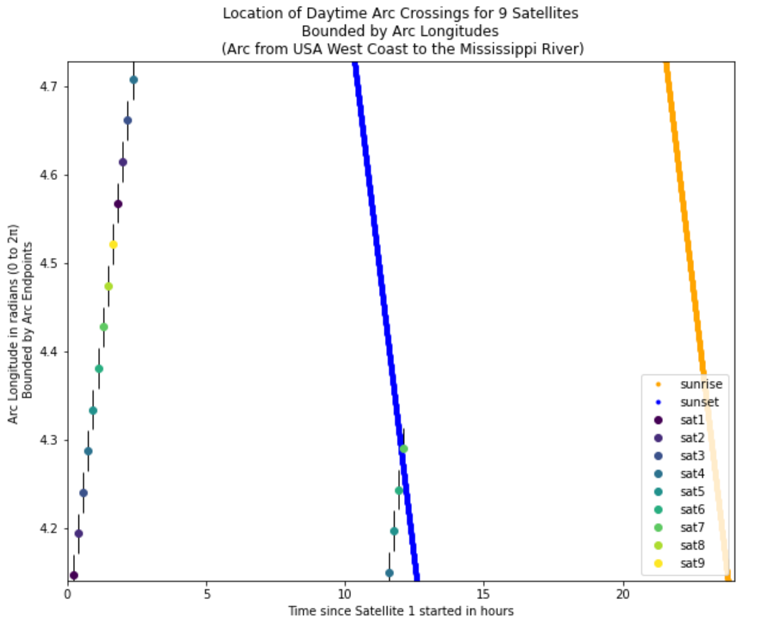
Cumulative percent coverage over arc with different numbers of satellites:

(Note that satellites are equally spaced to maximize coverage and minimize overlap)

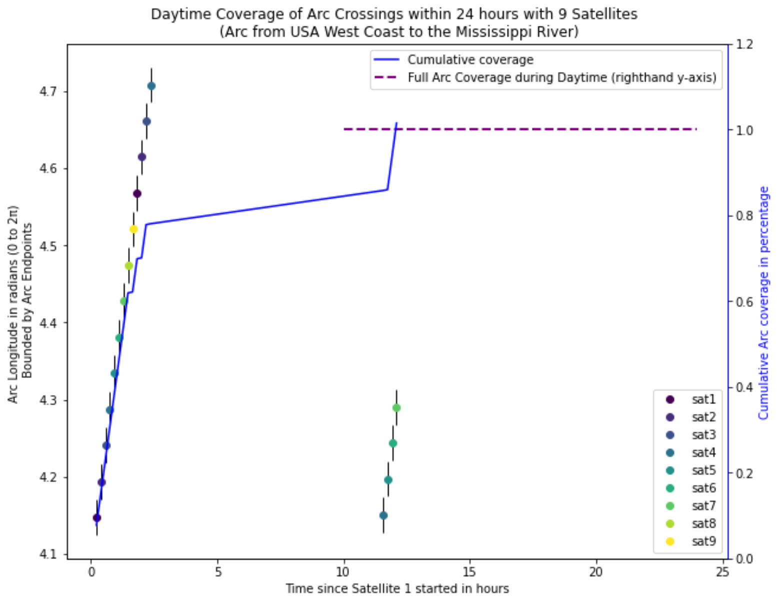
- 5 satellites: 0.647959611858685
- 6 satellites: 0.7124282184392872
- 7 satellites: 0.8489295795155646
- 8 satellites: 0.9653912111816317
- 9 satellites: 1.0151297478535097

Thus 9 satellites are still needed for full coverage of the arc west of the Mississippi River. However, if 96.5% coverage is sufficient, then 8 satellites could be used.

The graphs below show the locations of daytime arc crossings and the cumulative coverage of the arc for daytime arc crossings with 9 satellites.



This figure shows the longitude (bounded by arc) at which each satellite crosses the arc during daytime as related to the time since the first satellite was launched.



This figure shows the arc longitude at which each satellite crosses the arc during daytime as related to the time since the first satellite was launched.
The right axis shows the accumulated coverage of the arc by the swath intervals (as a percentage) over the 24 hour period.