GREMNIN - Satellite Propagator 🛰️

Problem Statement :-

Accurate prediction and visualization of satellite orbits is essential for planning Earth observation tasks, mission analysis, and satellite operations. However, tools that allow flexible, real-time orbital propagation based on Two-Line Element (TLE) sets often lack user interactivity or customizable parameters.

The task is to develop a Python-based satellite propagator using the SGP4 algorithm and user-supplied TLE data. The system should enable users to select a start date and time, propagate the orbit for a specified duration, and control the sampling resolution (from seconds to minutes). Additionally, the tool should allow the user to check whether a given geographic point (latitude, longitude) lies within the satellite's scanning footprint—a 200 km × 200 km square—at any time during the propagation window. On top of that, the system should include a 3D interactive globe to help the user visualise the path of the satellite's orbit.

The goal is to create an accurate, interactive, and user-configurable orbital analysis tool that aids in visibility assessment and satellite-ground coverage evaluation.

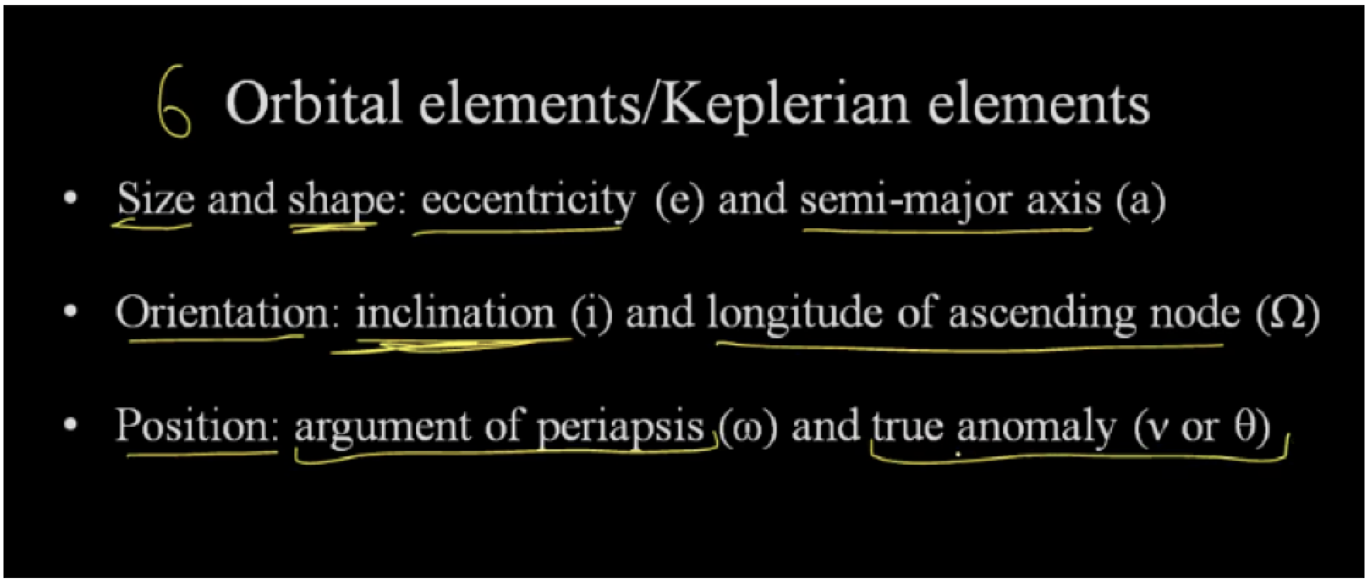
Definitions

Keplerian elements/coordinates🪐

Keplerian elements are values that are needed to define an objects orbit around one body. There are 6 Keplerian elements that are used to define the orbit.

They are :-

- Eccentricity
- Semi - major axis
- Inclination
- Right ascension of ascending node
- Argument of periapsis
- True anomaly



6 Orbital elements/Keplerian elements

- Size and shape: eccentricity (e) and semi-major axis (a)
- Orientation: inclination (i) and longitude of ascending node (Ω)
- Position: argument of periapsis (ω) and true anomaly (ν or θ)

1 - Eccentricity

Eccentricity in simple terms is how much the orbit is stretched. It is represented with the letter e. The more the value, the more it stretches. The greater the eccentricity, the further the two foci of the orbit are. This is required to define the shape of the orbit

An eccentricity of 0 will be a perfect circle. An eccentricity equal to 1 gives a parabola. An eccentricity greater than 1 gives a hyperbola. All stable orbits eccentricity lies between 0 and 1.

2 - Semi-Major Axis

The semi-major axis is half the major axis of the orbit. The major axis is the longest diameter across the orbit. This is required to define the size of the orbit.

3 - Inclination

The inclination of an orbit is the angle between the orbital plane from the reference plane of the object that the orbit is around, measured at the ascending node. This is required to define the orientation of the orbit.

4 - Right ascension of ascending node(RAAN)

The right ascension of ascending node is the angle between the vernal equinox and the ascending node of the orbit on the reference plane of the object. It helps to define the rotation of the orbit around the poles of the object its orbiting

5 - Argument of Periapsis

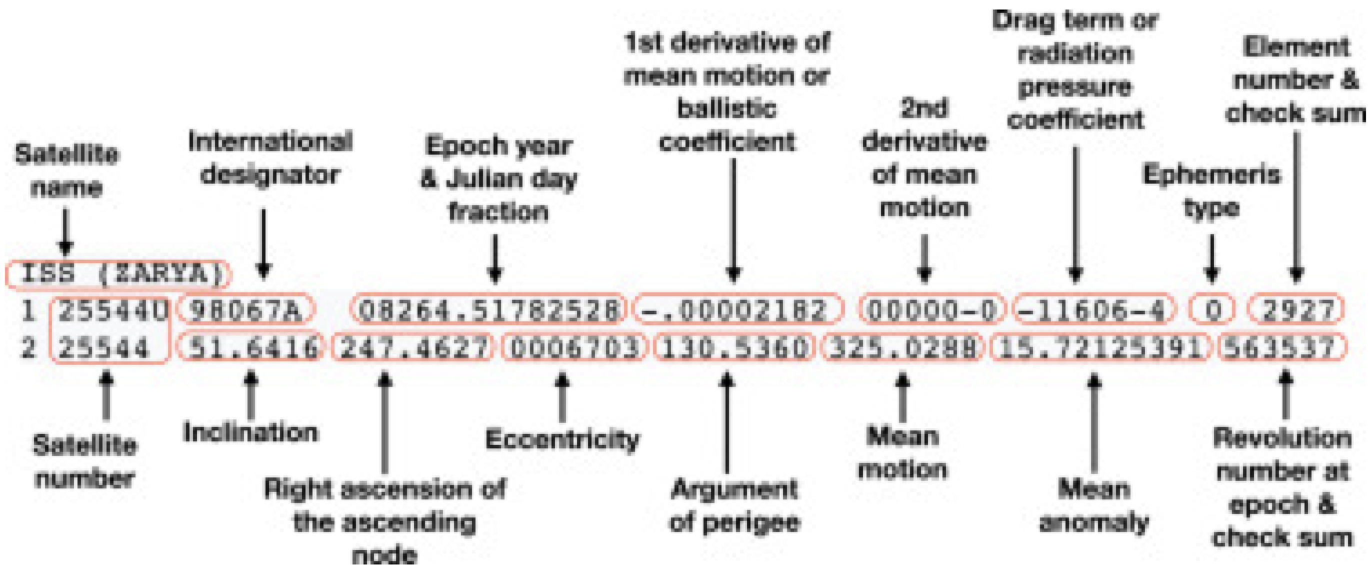
The argument of Periapsis defines the orientation of the ellipse in the orbital plane. It is the angle between the periapsis and the ascending node of the orbit

6 - True Anomaly

The true anomaly is the angle between the periapsis and the object in its orbit which is required to describe the position.

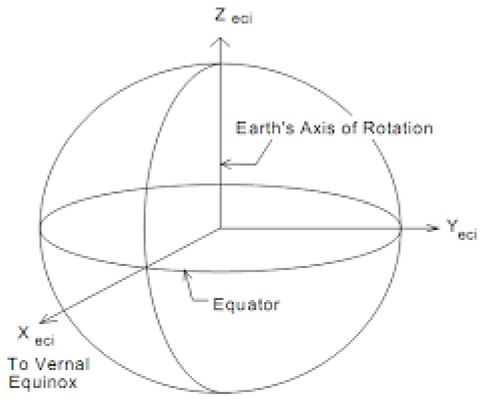
TLE

TLE or Two-Line Element is two lines of data that contains all the Keplerian elements of an object in orbit along with some more data such as velocity and acceleration. It is required to understand and visualise an orbit, and also to predict the position of an object in its orbit at any time. Using a model called SGP4, and the TLE of a satellite, you can predict its position in ECI coordinates at any time.


Satellite name
International designator
Epoch year & Julian day fraction
1st derivative of mean motion or ballistic coefficient
2nd derivative of mean motion
Drag term or radiation pressure coefficient
Ephemeris type
Element number & check sum
ISS (ZARYA)
1 25544U 98067A 08264.51782528 -.00002182 00000-0 -11606-4 0 2927
2 25544 51.6416 247.4627 0006703 130.5360 325.0288 15.72125391 563537
Satellite number
Inclination
Right ascension of the ascending node
Eccentricity
Argument of perigee
Mean motion
Mean anomaly
Revolution number at epoch & check sum

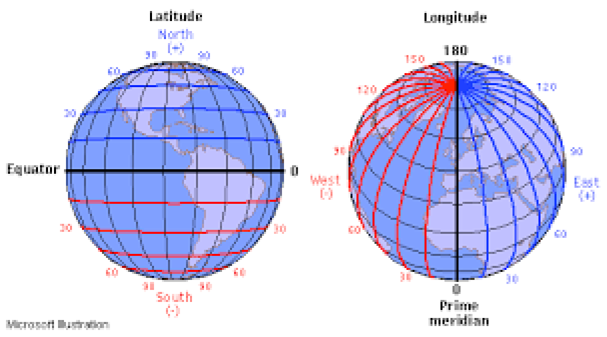
ECI

ECI is Earth Centered Inertial coordinate system. Its centre is the centre of the earth and the x and y axes are fixed based on far away stars. It doesn’t rotate with the earth



Latitude and Longitude

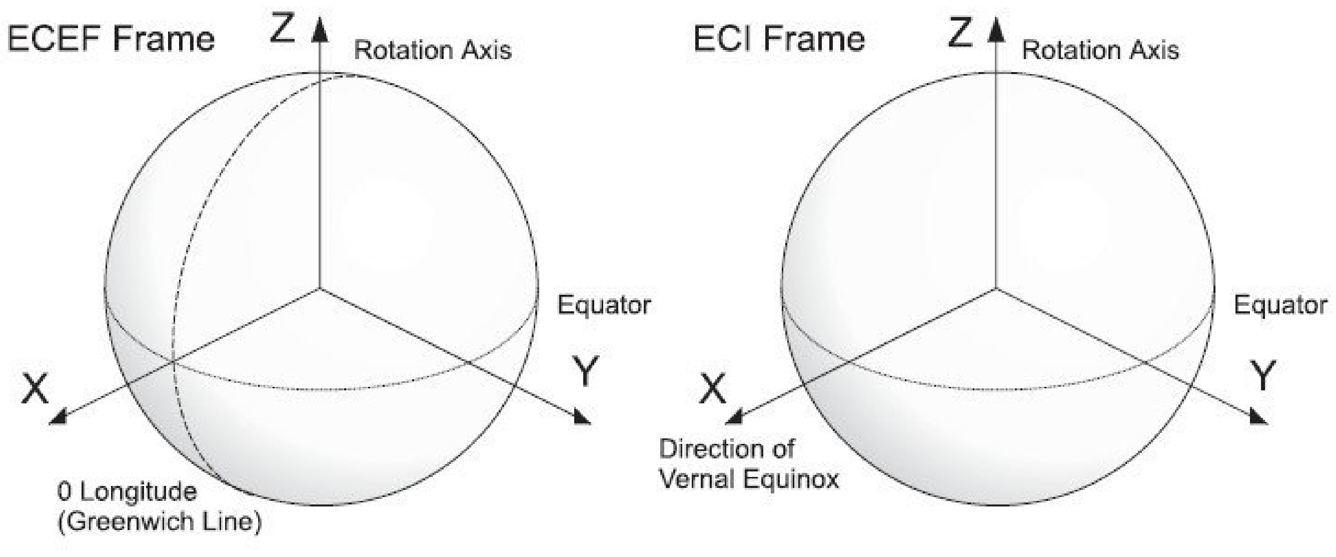
Latitude and Longitude coordinates are used on the map of the earth. ECI cannot be directly converted to Latitude and Longitude because it’s inertial so we have to convert to ECEF in between to convert it.



ECEF

ECEF is Earth Centered, Earth fixed. This means that the centre of this coordinate system is the centre of the earth, but that the x and y axes are fixed and move with the surface of the earth.

Since SGP4 outputs only in ECI coordinates, we need to convert it to ECEF so that we can convert that to Latitude and Longitude coordinates on the map. Having this intermediary step helps account for the earths rotation in the ECI output of the SGP4.



To account for the rotation of earth relative to the stars in the conversion between ECI to ECEF, the angle between the Greenwich Meridian and the vernal equinox(an axis of the ECI system) is taken. This angle is found with the Greenwich Mean Sidereal Time(GMST) because the angle changes with time as well.

The Julian date from the TLE or input is used to give the GMST angle. This angle is used to convert ECI to ECEF by changing the coordinates according to the rotation of the earth.

ECEF is then converted to Latitude and Longitude which can be used to identify the satellites location at a specific time on the world map where it is plotted.

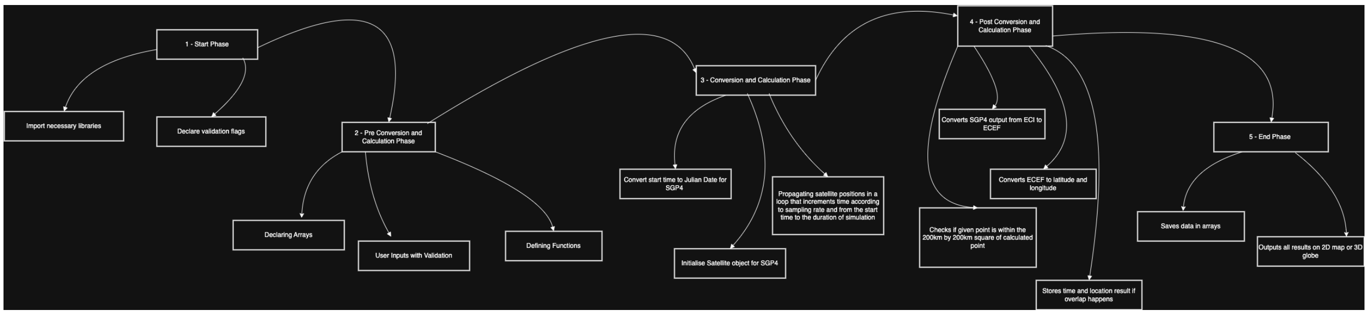
Program Functionality

The application plots a satellite's orbit with points around the world map based on the TLE, sampling rate, start time, and length of simulation, along with a 200km by 200km box about each point where the satellite can scan. It also provides a live location update of the satellite in real time.

An option is also provided whether to simulate the orbit on a 2D map or around a 3D globe. The 3D globe also has the same features as the 2D map.

Additionally, it checks whether the satellite can scan over a point that is specified by the user

Overall Flow


1 - Start Phase
Import necessary libraries
Declare validation flags
2 - Pre Conversion and Calculation Phase
Declaring Arrays
User Inputs with Validation
Defining Functions
3 - Conversion and Calculation Phase
Convert start time to Julian Date for SGP4
Initialise Satellite object for SGP4
Propagating satellite positions in a loop that increments time according to sampling rate and from the start time to the duration of simulation
4 - Post Conversion and Calculation Phase
Converts SGP4 output from ECI to ECEF
Converts ECEF to latitude and longitude
Checks if given point is within the 200km by 200km square of calculated point
Stores time and location result if overlap happens
5 - End Phase
Saves data in arrays
Outputs all results on 2D map or 3D globe

1. Start Phase :-
    - Import Libraries
    - Declare flags (For input validation)
2. Pre Conversion and Calculation Phase :-
    - User inputs (with validation)
    - Defining functions
    - Declaring arrays
3. Conversion and Calculation Phase :-
    - Converting start time to Julian Date format for SGP4
    - Initialise satellite object for SGP4
    - Propagate satellite position using SGP4 model using a loop that increments based on sampling rate and starts and ends based on user input
4. Post Conversion and Calculation Phase :-
    - Convert output coordinate format from ECI to ECEF
    - Convert ECEF to latitude/longitude
    - Check if 200km by 200km box around calculated point overlaps the specified point entered by the user
    - Store the result point and time if it overlaps.
5. End Phase :-
    - Saves data in arrays
    - Outputs all results on 2D map and/or 3D globe

Start Phase

The Start Phase imports all the libraries required to make the code work and also all the flags. Sice python does not have a post-condition loop and we need the inputs to run atleast once, a pre-condition loop is used to validate inputs, so the flags are all set to true initially in the Start Phase.

Importing Libraries

All the required libraries are in "requirements.txt"

Make sure to create a virtual environment before importing any new libraries

First run this to create the virtual environment

```
python -m venv newenv #newenv is the name of the virtual environment
```

Then run this to activate your virtual environment

On windows Command Prompt -

```
newenv\Scripts\activate.bat
```

On windows PowerShell -

```
newenv\Scripts\Activate.ps1
```

On MacOS

```
source newenv/bin/activate
```

Run this in the terminal to import all necessary libraries before running the code

```
pip install -r requirements.txt
```

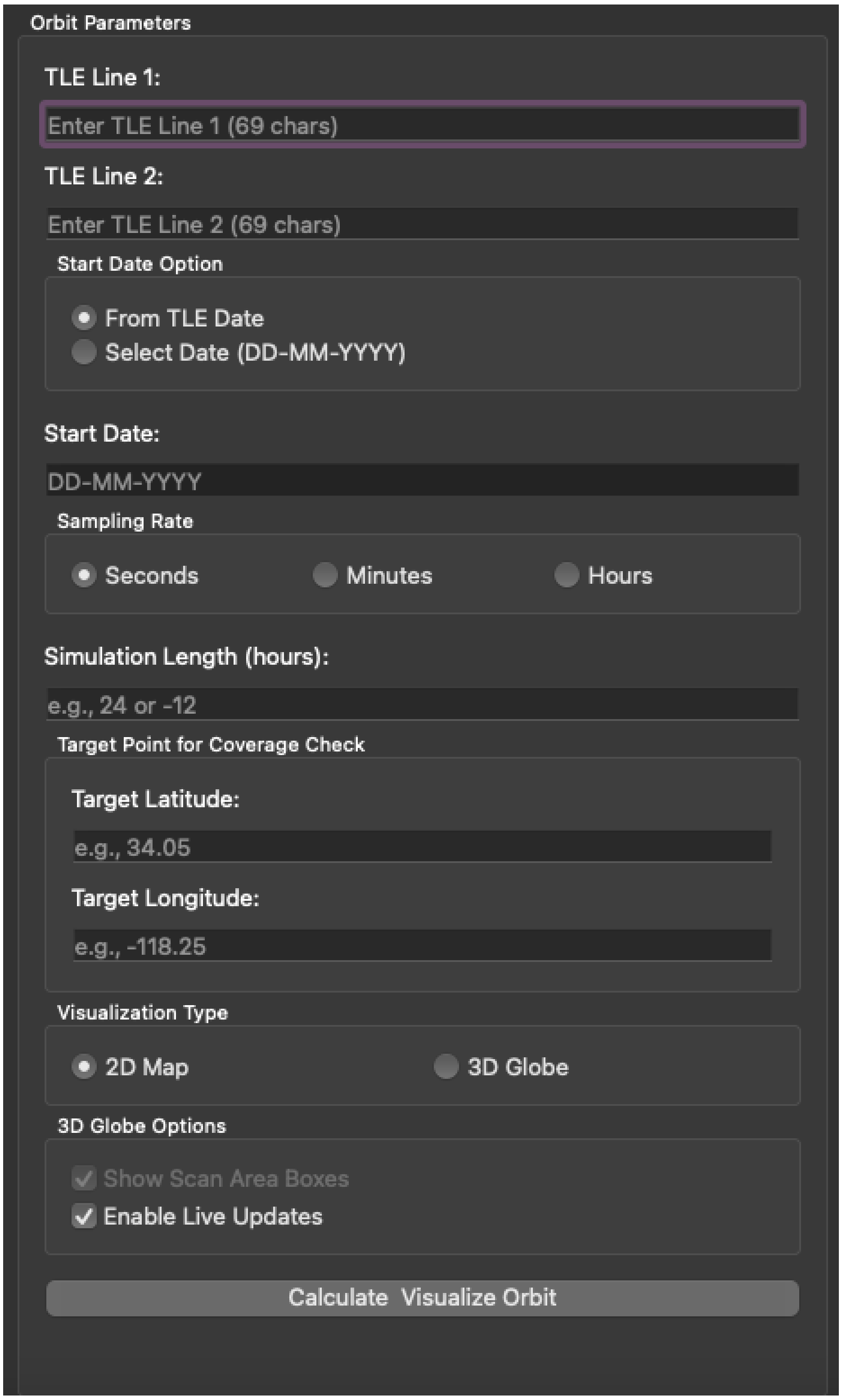
Declaring Flags

To ensure user input is valid, we need to repeat prompts until correct data is entered. Since Python lacks a built-in post-condition loop (i.e., do-while), we use a condition-controlled while loop that checks the condition before each iteration. To guarantee the loop runs at least once, all control flags must be initially set to True.

Pre Conversion and Calculation Phase

User Inputs and Validation

For the code to work, it needs data which is provided by the user based on which satellite they want to check and within which timeframe.


Orbit Parameters

TLE Line 1:

Enter TLE Line 1 (69 chars)

TLE Line 2:

Enter TLE Line 2 (69 chars)

Start Date Option

[✓ From TLE Date] Select Date (DD-MM-YYYY)

Start Date:

DD-MM-YYYY

Sampling Rate

[✓ Seconds] Minutes Hours

Simulation Length (hours):

e.g., 24 or -12

Target Point for Coverage Check

Target Latitude:

e.g., 34.05

Target Longitude:

e.g., -118.25

Visualization Type

[✓ 2D Map] 3D Globe

3D Globe Options

✓ Show Scan Area Boxes

✓ Enable Live Updates

Calculate Visualize Orbit

The inputs taken by the program are :-

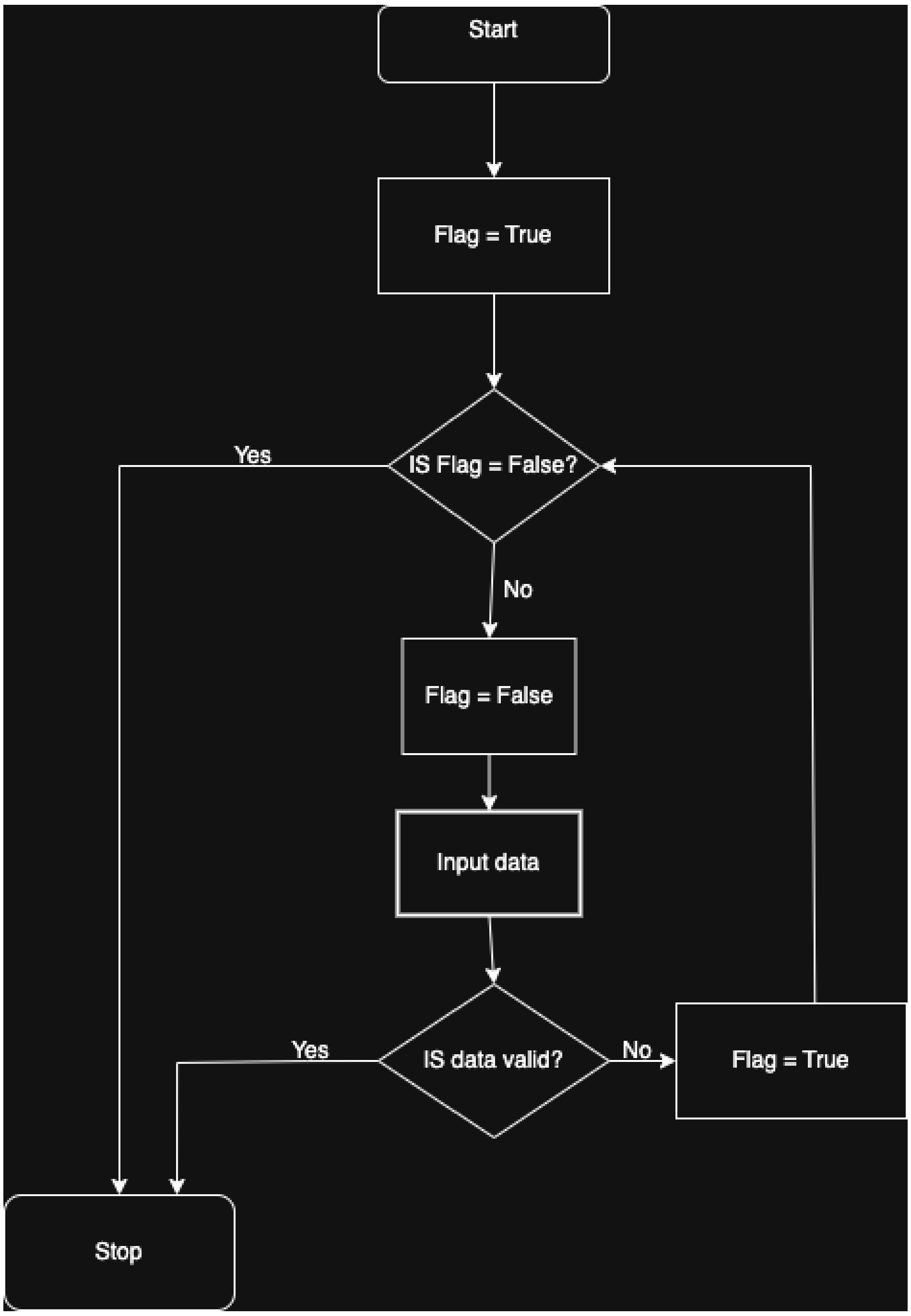
1. TLE Lines 1 and 2
2. Sampling Rate
3. A coordinate to check for Satellite overpass
4. A choice between starting from a custom date or the TLE's epoch
5. Duration of the orbit in hours

Input Validation

Input validation is critical to ensure the application handles user-provided data—like TLE entries, coordinates, or sampling rates—safely and accurately. Without validation, malformed inputs can cause crashes AND incorrect orbit predictions. For example, a wrongly formatted TLE could break the SGP4 propagator, or non-numeric sampling rates might lead to animation issues. Proper validation ensures the system remains robust, stable, and user-friendly. This is why it has been used at every step of the way when inputting data.

Each input has its own parameters/formats that need to be checked before it can be used by the program.

This is a flowchart showing how it works :-



1. TLE Input
    - User is asked to enter Line 1 and Line 2
    - Each line has to be exactly 69 characters (TLE Format)
    - If invalid, it has to be re-entered

eg.

```
while Flag is True:
    Flag = False
    print("Enter current TLE Line 1 (No Name of Satellite) - ")
    TLE1 = str(input().strip())

    print("Enter current TLE Line 2 (No Name of Satellite) - ")
    TLE2 = str(input().strip())
    if TLE1 and TLE2 and len(TLE1) == 69 and len(TLE2) == 69:
         break
    else:
        print("Please enter valid TLE lines (69 characters each).")
        Flag = True
```

2. Sampling Rate Input
    - Options for Seconds, Minutes, and Hours is given
    - Based on that, the rate of sampling is converted to steps per day

eg.

```
print("Select Sampling Rate")
while flagrate is True:
    flagrate = False
    print("Type 1 for Seconds")
    print("Type 2 for Minutes")
    print("Type 3 for Hours")
    rateselect = int(input())
    if rateselect == 1:
        rate = 24*60*60
    elif rateselect == 2:
        rate = 24*60
    elif rateselect == 3:
        rate = 24
    else:
        flagrate = True
        print("Enter only 1, 2 or 3")
```

3. Latitude/Longitude Input
    - Point to check has to be a float value
    - If not, input is rejected and asked to re-enter

eg.

```
while flagpoints is True:
    flagpoints = False
    latpoint = float(input("Please enter the latitude of the point you
want to check if the satellite goes over it"))
    lonpoint =  float(input("Please enter the longitude of the point you
want to check if the satellite goes over it"))
    if type(latpoint) != (float or int) or lonpoint != type(lonpoint) !=
(float or int):
        print("Enter only integers or decimal numbers")
        flagpoints = True
    else:
        break
```

4. Date Selection
    - User chooses to use custom date, or the one in their TLE's epoch to start from
    - If custom, user must enter in format DD-MM-YYYY, which is then checked within a range check for each value, and a format check.

eg.

```
def dateformatcheck(startdate):
    pattern = r"^\d{2}-\d{2}-\d{4}$"
    return bool(re.match(pattern, startdate))

flagchoice = True
while flagchoice is True:
    flagchoice = False
    print("Type 1 to start from date of choice")
    print("Type 2 to start from date in TLE")
    choice = float(input())
    if choice ==1:
        print("Type date to start the simulation in DD-MM-YYYY(only post
1957 and before 2056) - ")
        while flagstartdate is True:
            startdate = input()
            flagstartdate = False
            if dateformatcheck(startdate) == True:
                if int(startdate[0:2]) > 31 or int(startdate[0:2]) <= 0:
                    flagstartdate = True
                    print("Enter valid day between 1 and 31")
                elif int(startdate[3:5]) > 12 or int(startdate[3:5]) <= 0:
                    flagstartdate = True
                    print("Enter valid month between 1 and 12")
                elif int(startdate[6:10]) > 2056 or int(startdate[6:10])
<1957:
                    flagstartdate = True
                    print("Please enter a year that is after 1957 and
before 2056")
                else:
                    break
                else:
                    flagstartdate = True
                    print("Please enter exactly as DD/MM/YYYY")
        elif choice == 2:
            break
        else:
            flagchoice = True
            print("Please type only 1 or 2 as options")
```

5. Duration in hours
    - Makes sure that the time entered is not 0

eg.

```
print("Enter length of time you want check orbit for in hours - ")
while flagtime is True:
    flagtime = False
    requestedtimeh = float(input().strip())
    if requestedtimeh == 0:
        flagtime = True
        print("Please enter a non zero value")
    else:
        requestedtimed = float(requestedtimeh/24)
        break
```

Defining Functions

The recurring need of a certain conversion from one coordinate system to another or one time to another calls for the use of functions that can be used anytime.

1. latchange

    eg.

    ```
    def latchange(distance):
    return (distance / 6371) * (180/math.pi)
    ```

    This function is used to convert distances in km into latitude degrees

2. lonchange

    eg.

```
def lonchange(lat_deg, distance):
lat_rad = math.radians(lat_deg)
radius_lat = 6371 * math.cos(lat_rad)
if radius_lat == 0:
    return 0
return (distance/radius_lat) * (180/math.pi)
```

This function is used to convert distances in km into longitude degrees by using latchange and distance to find it

3. squarepoints

   eg.

```
ef squarepoints(lat, lon):
delta_lat = latchange(100)  # ≈ 0.8987°
lat_north = lat + delta_lat
lat_south = lat - delta_lat
delta_lon_north = lonchange(lat_north, 100)
delta_lon_south = lonchange(lat_south, 100)
p1 = (lat_north, lon + delta_lon_north)
p3 = (lat_south, lon - delta_lon_south)
return p1, p3
```

   This function, using the latchange and lonchange functions, finds 2 corners of a square and returns them in latitude and longitude with the square side being 200km using the a point (lat,lon) as its parameters for the centre of the square.

4. dateformatcheck

   eg.

```
def dateformatcheck(startdate):
pattern = r"^\d{2}-\d{2}-\d{4}$"
return bool(re.match(pattern, startdate))
```

   This function is used to check whether the inputted date from the user is compliant to the format DD-MM-YYYY

5. daysinyear

   eg.

```
def daysinyear(day,month,year):
if year % 4 == 0 and (year % 100 != 0 or year % 400 == 0) and month>2:
    return day + days[month] + 1
```

```
else:
    return day + days[month]
```

This function is used to count the total number of days completed in the year given the date. It makes use of an array that contains the cumulative number of days at the end of each month called days[]

6. gmst

   eg.

```
def gmst(julian_date):
d = julian_date - 2451545.0
T = d / 36525.0
gmst = 280.46061837 + 360.98564736629 * d + 0.000387933 * T**2 - T**3
/ 38710000.0
return math.radians(gmst % 360.0)
```

   This function uses the Julian Date to convert it to GMST (Greenwich Mean Sidereal Time) which helps with converting ECI to ECEF since the time determines the earths rotation in reference to the far away stars.

7. teme_to_ecef

   eg.

```
def teme_to_ecef(teme_pos, julian_date):
theta = gmst(julian_date)
R = [
    [math.cos(theta),  math.sin(theta), 0],
    [-math.sin(theta), math.cos(theta), 0],
    [0,                0,               1]
]
x, y, z = teme_pos
ecef_x = R[0][0] * x + R[0][1] * y + R[0][2] * z
ecef_y = R[1][0] * x + R[1][1] * y + R[1][2] * z
ecef_z = R[2][0] * x + R[2][1] * y + R[2][2] * z
return (ecef_x, ecef_y, ecef_z)
```

   This function uses the gmst function, Julian Date and math library to convert the ECI coordinate system to ECEF.

8. ecef_to_geodetic

   eg.

```
def ecef_to_geodetic(ecef_pos):
transformer = Transformer.from_crs(
    {"proj": "geocent", "ellps": "WGS84", "datum": "WGS84"},
    {"proj": "latlong", "ellps": "WGS84", "datum": "WGS84"},
    always_xy=True
)
x, y, z = ecef_pos
lon, lat, alt = transformer.transform(x * 1000, y * 1000, z * 1000)
return lat, lon, alt / 1000.0
```

This function uses the ecef values and converts it into latitude and longitude format using the transformer library

Declaring Arrays

Arrays are used to store multiple values of data in a table like format. This is useful as it is easy to then store and select as many values as needed.

1. Foundarr This array is made to store all the time and location values of any satellite positions that are found to be overlapping with the given point by the user. This has the size of as many points are being calculated. This is done because that is the maximum amount of points that can overlap.

```
Foundarr = np.zeros((samplenum+1,3),float)
```

2. LATLONarr This array is created to hold all the latitudes, longitudes and altitudes of the calculated points. It has as many positions as there are points being calculated.

```
LATLONarr = np.zeros((samplenum+1,3),float)
```

3. days This array holds the cumulative amount of days at the end of each month, which helps convert time.

```
days = [0, 0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334]
```

Conversion and Calculation Phase

Converting Gregorian to Julian Date

```
if choice == 2 :
    try:
        year = int(TLE1[18:20])
        if year < 57:
            year = year + 2000
        else:
            year = year + 1900
        day_of_year = float(TLE1[20:32])
```

```
            base_date = datetime(year,1,1)
            days = int(day_of_year) - 1
            Fraction = day_of_year - days
            seconds = Fraction * 86400
            epoch_date = base_date + timedelta(days=days, seconds=seconds)
            jd_epoch, fr_epoch = jday(epoch_date.year, epoch_date.month,
   epoch_date.day, epoch_date.hour, epoch_date.minute, epoch_date.second)
        except ValueError as e:
            print(f"Error parsing TLE epoch: {e}. Exiting.")
            exit()
    else:
        try:
            year = int(startdate[6:10])
            day_of_year = daysinyear(int(startdate[0:2]), int(startdate[3:5]),
   int(startdate[6:10]))
            base_date = datetime(year,1,1)
            days = int(day_of_year) - 1
            Fraction = 0
            seconds = Fraction * 86400
            epoch_date = base_date + timedelta(days=days, seconds=seconds)
            jd_epoch, fr_epoch = jday(epoch_date.year, epoch_date.month,
   epoch_date.day, epoch_date.hour, epoch_date.minute, epoch_date.second)
        except ValueError as e:
            print(f"Error parsing TLE epoch: {e}. Exiting.")
            exit()
```

This code first checks whether the user chose to start from TLE time or custom date, then accordingly takes that time and converts it to Julian Date by taking the Julian days and fraction of day using the datetime library and the daysinyear function

Initialising Satellite Object

The satellite object is created to be put into the SGP4 with which it will create an ouput. The object combines both TLE lines.

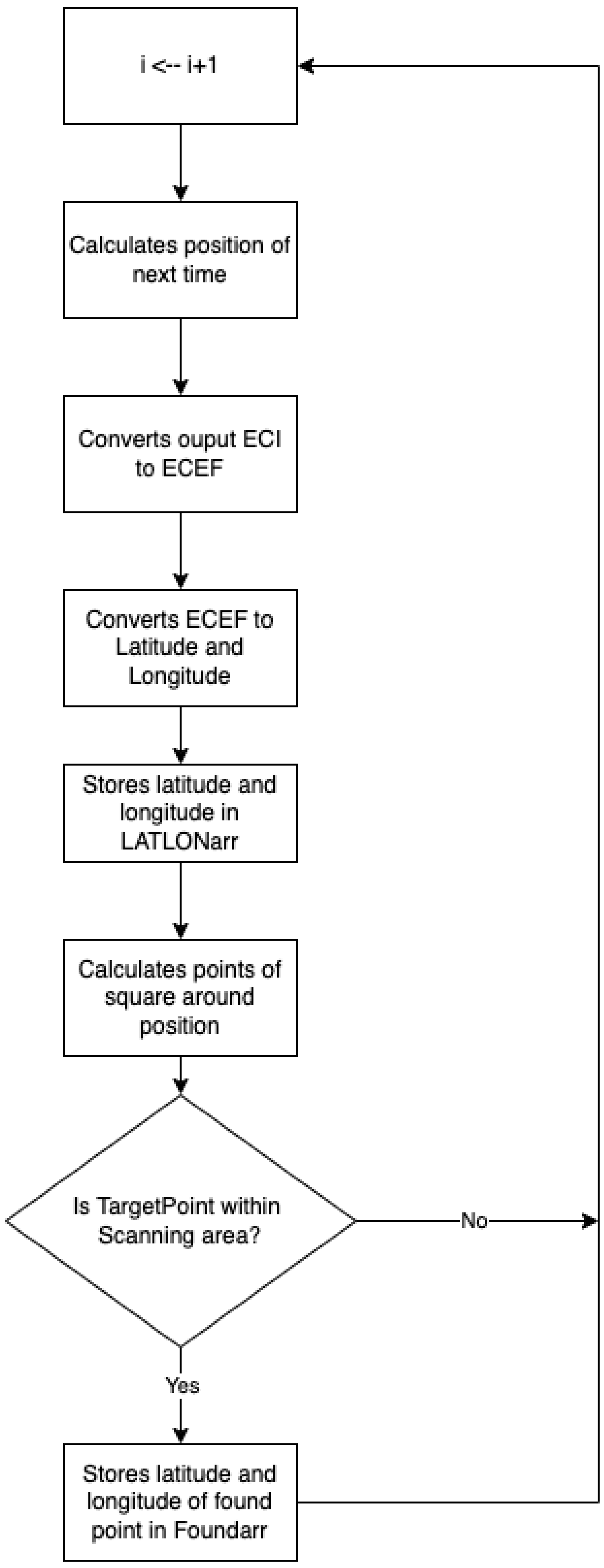
Propagation of satellite

The SGP4 model is then used along with the satellite object and the Julian Days and fraction of day to come up with an output of errorcode, position (in ECI) and velocity.

```
    error_code, position, velocity = satellite.sgp4(jd,fr)
```

This happens in a loop with an increment of time so that all the points in the orbit can be calculated.

Post Conversion and Calculation Phase


i <-- i+1
Calculates position of next time
Converts ouput ECI to ECEF
Converts ECEF to Latitude and Longitude
Stores latitude and longitude in LATLONarr
Calculates points of square around position
Is TargetPoint within Scanning area?
No
Yes
Stores latitude and longitude of found point in Foundarr

Converting ECI to ECEF

The output of ECI is then converted to ECEF using the teme_to_ecef function

```
ecef_pos = teme_to_ecef(position, jd+fr)
```

Converting ECEF to Latitude and Longitude

The ECEF output from the previous step is converted to latitude, longitude and altitude using the ecef_to_geodetic function

```
lat, lon, alt = ecef_to_geodetic(ecef_pos)
```

These values are also stored in the LATLONarr

Checking Overlap and storing values if overlap occurs

```
 p1,p3 = squarepoints(lat,lon)
 p1lat, p1lon = p1
 p3lat,p3lon = p3
 if p3lat <= latpoint <= p1lat and p3lon <= lonpoint <= p1lon:
 Foundarr[i,1] = latpoint
 Foundarr[i,2] = lonpoint
 totalfound = totalfound + 1
```

After applying the squarepoints function to the latitude and longitude calculated within the loop, this checks whether the inputted point is within the scanning area by checking if its coordinates are within the square.

If it does overlap, the latitudes and longitudes of that point are stored.

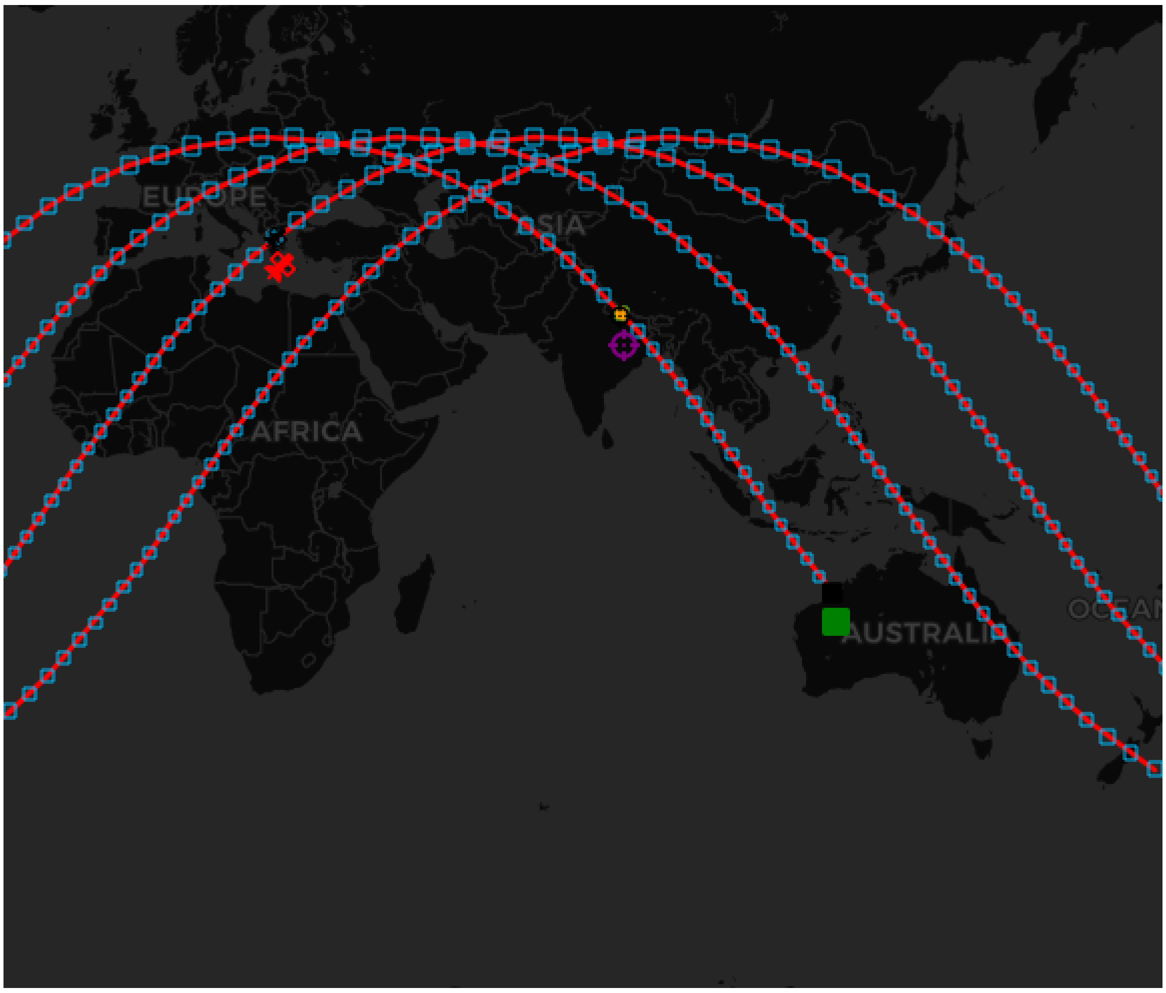
End Phase

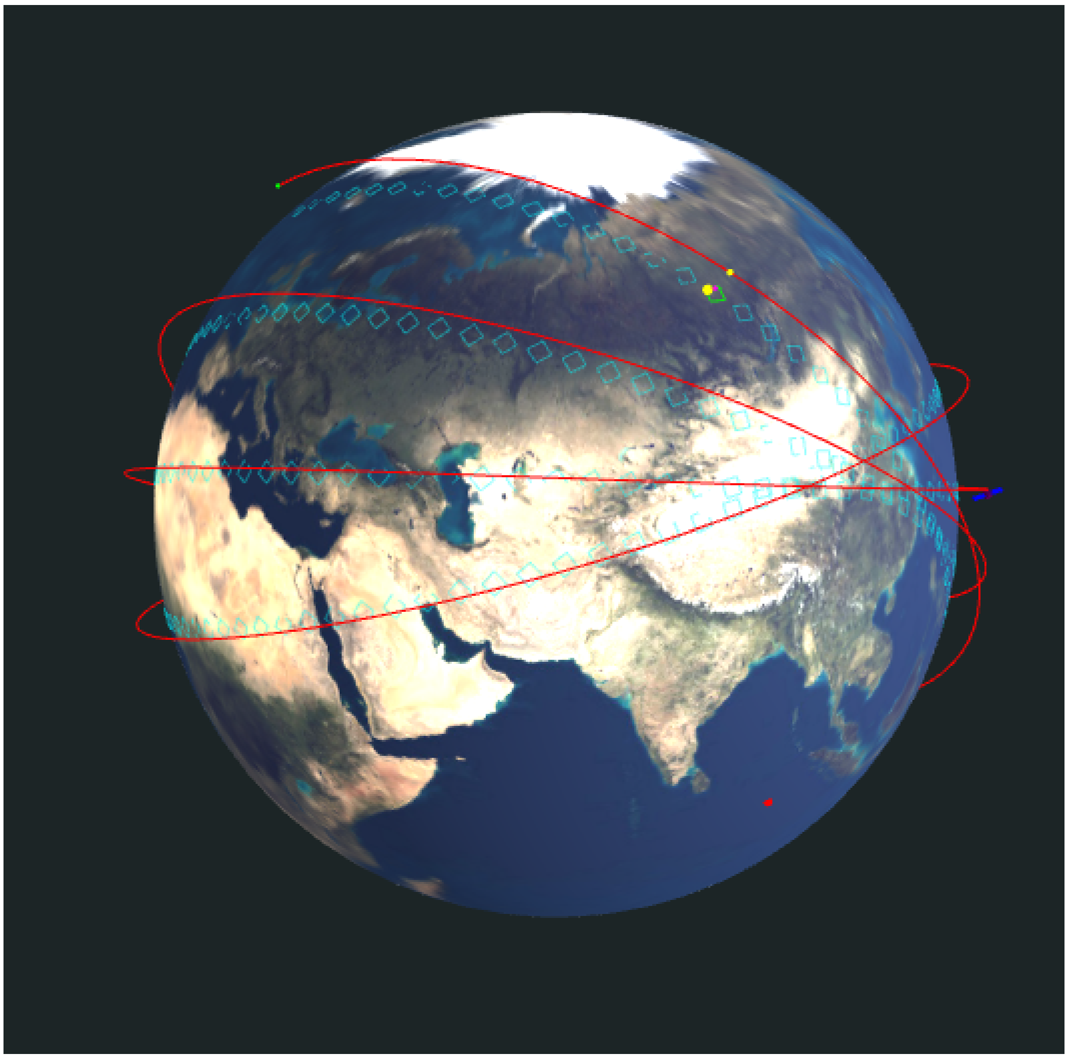
Saves data in Arrays

As previously mentioned, the latitudes and longitudes of all the points are stored in the LATLONarr array in the loop, and the points that overlap are stored in another array called Foundarr.

Outputs all results on 2D map / 3D globe

All the data (including points, squares, start and end point, and the point selected to find overlap) is then displayed onto a 2D map, or 3D globe according to the users selection


EUROPE
ASIA
AFRICA
AUSTRALIA
OCEAN



Sources to get your own TLE

You can get TLE of different satellites by searching on Celestrak, NORAD and space-track.org.