Satellite Tracker Navigator

This code is designed to track the positions of multiple satellites using Two-Line Element Sets (TLEs), convert their positions from Earth-Centered, Earth-Fixed

(ECEF) coordinates to Latitude, Longitude, and Altitude (LLA) format, and then find when these satellites pass over a specified geographic bounding box. The

code uses Python and several libraries for these tasks, such as sgp4 for satellite propagation, pyproj for coordinate transformations, and concurrent.futures for parallel processing.

The following steps provide a more concise and clarified explanation of each phase in the code:-

(a)Importing Libraries:-

The code imports essential libraries:

sgp4 for satellite tracking.

pyproj for coordinate transformation.

datetime for handling time.

concurrent.futures for parallel processing.

logging for error logging.

(b)Initializing Logger:-

A logger is set up to record information and errors in a file named 'satellite_tracking.log'.

Step 1: Get Satellite Location (get_satellite_positions function):-

In this step, the code is responsible for determining the locations of satellites that are specified in Two-Line Element Sets (TLEs) over a predefined period. Here's a more detailed breakdown:

Why TLEs?

TLEs are sets of data that provide information about a satellite's orbit, including its position and velocity at a specific time.

These elements are crucial for accurately tracking satellites.

Key Objectives:

1.TLE Parsing: The code begins by taking a list of TLE lines (text data) as input. Each line represents a set of TLE data for a particular satellite.

TLE data typically includes information like the satellite's name, its orbital parameters, and the epoch time (a reference time for the TLE).

2.Position Calculation: Once the TLE data is obtained, the code parses it to extract relevant information about each satellite's orbit.

This information includes the satellite's orbital elements, such as the semi-major axis, eccentricity, inclination, right ascension of the ascending node, argument of perigee, mean anomaly, and more.

3.Propagation: With the orbital elements, the code calculates the positions of the specified satellites for each minute increment over a defined duration.

This involves predicting where the satellite will be located in space at each minute within the specified time range.

4.Timestamps: For each calculated satellite position, the code also generates timestamps. These timestamps indicate when the position was calculated, typically in the format of year, month, day, hour, minute, second, and microsecond.

5.Error Handling: Throughout this process, the code monitors for any errors that may occur during satellite position calculation.

If any errors do occur, such as issues with TLE data or mathematical computations, these errors are logged for reference and troubleshooting.

Step 2: Convert Data to Lat-Long-Alt Format (transform_data function):-

In this step, the code transforms satellite positions from Earth-Centered, Earth-Fixed (ECEF) coordinates to a more human-friendly format known as

Latitude,Longitude, and Altitude (LLA). This transformation is facilitated by the pyproj library.

(*) Why ECEF to LLA Conversion?

Satellites are typically tracked and located in the ECEF coordinate system, which is a three-dimensional Cartesian coordinate system centered at the Earth's center.

While ECEF is suitable for mathematical calculations, it's less intuitive for humans to understand. Latitude, Longitude, and Altitude (LLA) coordinates are more familiar for representing locations on the Earth's surface.

How the Conversion Works:

1.Coordinate Transformation: The pyproj library is used to perform this conversion. It takes the ECEF coordinates of a satellite's position and translates them into latitude, longitude, and altitude values.

2.Result Format: The transformed data is structured into a format that includes latitude (representing North-South position), longitude (representing East-West position), and altitude (representing height above the Earth's surface).

Purpose: This conversion makes it easier for users to understand and work with satellite positions since LLA coordinates are commonly used to represent locations on Earth. It enhances the readability and usability of the data.

Step 3: Find Satellite Passes Over a Region (is_in_bounding_box function):-

(*)This step focuses on determining whether a satellite is passing over a specific geographic region, which is defined by a bounding box. The is_in_bounding_box function checks if a given coordinate (latitude and longitude) falls within this predefined bounding box.

Bounding Box Explanation:

(*)A bounding box is a rectangular area defined by its minimum and maximum latitude and longitude values.

(*)Think of it as drawing a rectangle on a map to represent an area of interest.

(*)Checking Satellite Position Against the Bounding Box:

(*) For each satellite position, the code calls the is_in_bounding_box function.

(*) The function takes the latitude and longitude of the satellite's position and compares them to the latitude and longitude ranges of the bounding box.

Step 4: Optimize Code to Reduce Computation Time:-

1.process_satellite_chunk function:

(*)This function is designed to handle smaller "chunks" of satellite positions, dividing the overall workload into manageable parts.

(*)chunk represents a portion of the satellite data to process, and bounding_box defines the geographic region of interest.

```
def process_satellite_chunk(chunk, bounding_box):
    filtered_data = []
    for (time, position) in chunk:
        lat, lon, _ = position
       if is_in_bounding_box((lat, lon), bounding_box):
           filtered_data.append(f"Time: {time}, Latitude: {lat}, Longitude: {lon}")
    return filtered_data
```

2.It iterates through each satellite position within the chunk.

(*)For each position, it checks if the latitude and longitude of the satellite fall within the specified bounding_box.

(*)If the satellite position is within the region of interest, it includes the position details (time, latitude, and longitude) in the filtered_data list.

3.compute_bounding_box function:

(*)This function calculates the bounding box for the entire dataset of satellite positions.

```
def compute_bounding_box(positions):
    latitudes, longitudes, _ = zip(*[pos[1] for pos in positions])
    lat_min = min(latitudes)
    lat_max = max(latitudes)
    lon_min = min(longitudes)
    lon_max = max(longitudes)
    return (lat_min, lon_min, lat_max, lon_max)
```

(*)It extracts latitude and longitude values from all satellite positions in the dataset.

(*)Then, it calculates the minimum and maximum latitude and longitude values among all positions.

(*)The result is a bounding box that defines the geographic region containing all satellite positions.

4.Main Function (main function):

(*)The main function coordinates the optimization process:

```
def main():
    with open('30000sats.txt', 'r') as file:
        tle_data = file.readlines()
    positions = get_satellite_positions(tle_data)
    bounding_box = compute_bounding_box(positions)
    num_processes = 4
    chunk_size = len(positions) // num_processes
    chunks = [positions[i:i + chunk_size] for i in range(0, len(positions), chunk_size)]
    with concurrent.futures.ProcessPoolExecutor() as executor:
        results = executor.map(process_satellite_chunk, chunks, [bounding_box] *
len(chunks))
    filtered_data = [result for chunk_result in results for result in chunk_result]
    for result in filtered_data:
        print(result, flush=True)
if __name__ == "__main__":
    main()
```

(*)The code reads TLE data for 30,000 satellites from a file ('30000sats.txt') and retrieves their positions.

(*)It then computes the bounding box that encompasses all satellite positions to define the region of interest.

(*)The dataset of satellite positions is divided into smaller chunks to enable parallel processing, improving computation time.

(*)Each chunk is processed concurrently using ProcessPoolExecutor, taking advantage of multiple CPU cores.

(*)Finally, the filtered results (satellite positions within the bounding box) are combined and printed to the console.

(*)This code efficiently optimizes the computation time by dividing the workload, performing parallel processing, and filtering satellite positions based on the specified bounding box, making it suitable for handling a large number of satellites.

Summary:

The provided Python code offers a comprehensive solution for tracking and analyzing the positions of a multitude of satellites using Two-Line Element Sets (TLEs). It optimizes performance through parallel processing and efficient memory usage, making it suitable for handling a large number of satellites. The code parses TLE data, transforms coordinates to Latitude, Longitude, and Altitude (LLA) format, checks satellite passes over user-defined regions, and employs distributed computing techniques to minimize computation time. Its modular structure and error handling ensure production readiness, while the use of bounding boxes and efficient processing enable real-world applications, particularly when dealing with extensive satellite datasets.