ChatGPT

Satellite Tracker Project Guide

A **Satellite Tracker** is an application that fetches real orbital data (Two-Line Element sets) and uses them to compute and display the real-time position and ground track of satellites. This project will use Skyfield (a Python astronomy library) to load NORAD TLE data, compute satellite positions, convert them to latitude/longitude, and visualize them on a map (e.g. using Leaflet or Mapbox). It can also predict future passes (rise/set times) and visibility (sunlit vs. dark) for a given location. This makes for an impressive but manageable project (suitable for a beginner/intermediate CS student), involving real-time data, orbital math, and interactive mapping.

Skyfield is well-suited for this purpose: it **"predict(s) the position of an Earth satellite by downloading the satellite's standard SGP4 orbital elements"** [1]. In practice, you will fetch TLE data (from sources like CelesTrak or Space-Track), load it into a Skyfield `EarthSatellite` object, and then use its built-in methods to compute the satellite's location over time. The core pipeline is:

1. **Fetch and parse TLE data** (NORAD two-line elements, from CelesTrak or similar).
2. **Compute satellite positions** (use Skyfield's SGP4 propagation).
3. **Convert ECI to geographic coordinates** (latitude/longitude) using `wgs84.subpoint()`.
4. **Plot the ground track** on an interactive map (Leaflet/Mapbox) with a moving marker or polyline.
5. **Predict visible passes** (find times when the satellite is above the horizon and in sunlight for a given location).

Below is a detailed guide to each part of the project, plus ideas for extensions and improvements. Every step is essential for a working tracker, and we reference authoritative sources for guidance on Skyfield and orbital data.

1. Fetching TLE Data (Satellite Orbital Elements)

- **Source of TLEs:** Most satellite trackers use **CelesTrak** or the official Space-Track portal to get up-to-date TLE files. For example, CelesTrak provides pre-packaged groups of satellites (like "Space Stations" or "Weather") in plain text (TLE) or in modern formats (JSON/CSV) [2]. The classic TLE format consists of two lines of fixed-width ASCII per satellite (as shown for the ISS in the Skyfield docs [2]).
- **Downloading TLEs:** In Python you can use `requests` or Skyfield's own loader to fetch these files. Skyfield's `load.download()` function can retrieve a URL and save it locally, which helps with caching. For example, one documented pattern is to download CelesTrak's "Space Stations" list once per week (using `load.download(url, filename=name)`) to avoid hammering the servers [3]. The Skyfield docs also advise saving the data file on first run and only re-downloading when it gets old (a few days) [4]. In code, you might do something like:

```
from skyfield.api import load
max_age = 7.0           # days before re-download
filename = 'stations.tle'
url = 'https://celestrak.com/NORAD/elements/gp.php?GROUP=stations&FORMAT=TLE'
if not load.exists(filename) or load.days_old(filename) >= max_age:
    load.download(url, filename=filename)
```

```
with load.open(filename) as f:
    lines = f.readlines()
```

Skyfield also offers `parse_tle_file()` to parse a TLE text file into `EarthSatellite` objects [5].

- **TLE Accuracy:** Keep in mind that **TLE data ages quickly**. A set of elements is most accurate near its epoch; after a week or two its accuracy degrades. Skyfield's documentation warns that "satellite orbital elements go rapidly out of date" [6]. Thus your app should periodically refresh the TLE data (e.g. daily or weekly) to maintain accuracy.

- **Storage:** Store the downloaded TLE text (or JSON/CSV) on disk or cache it in memory, so you don't refetch on every update. As the docs note, saving the file prevents unnecessary server load [4]. You could also version your data if you want reproducibility.

2. Computing Satellite Positions with Skyfield

- **Creating EarthSatellite:** Once you have the two-line element data for a satellite, create a Skyfield object. For example:

```
from skyfield.api import EarthSatellite, load
ts = load.timescale()
sat = EarthSatellite(tle_line1, tle_line2, 'ISS', ts)
```

  This object can propagate the orbit using SGP4.

- **Selecting Times:** Decide if you need just the current position or a series of positions (e.g. to draw a track). You can get the current time via `ts.now()` or create a series of times (e.g. every second or minute).

- **Calculating Position:** To get the satellite's geocentric position at a given time, use `sat.at(time)`. This returns a position vector (`Geocentric`) in Earth-centered coordinates. For example:

```
t = ts.now()
geocentric = sat.at(t)
```

  This gives the position of the satellite relative to Earth's center at time `t`.

- **Latitude/Longitude Conversion:** To map the satellite on a 2D Earth map, convert this vector to geographic coordinates. Skyfield provides `wgs84.subpoint()` for this: it takes the geocentric vector and returns the point on Earth (latitude, longitude, and elevation) directly below the satellite [7]. In code:

```
from skyfield.api import wgs84
subpoint = wgs84.subpoint(geocentric)
```

```
lat = subpoint.latitude.degrees
lon = subpoint.longitude.degrees
```

This yields the satellite's ground track position (the sub-satellite point). The docs note: "You can pass [the position vector] to the subpoint() method of a standard geoid to compute latitude and longitude" [7]. (If you need the actual ground elevation, you'd have to query a DEM, but for the map a zero-elevation subpoint suffices [8].)

- **Observer-Centric Angles:** (Optional) To know where the satellite appears in your sky, you can compute topocentric altitude/azimuth. For an observer location (use `wgs84.latlon(observer_lat, observer_lon)` to create it), subtract to get a vector difference, then call `.altaz()` on that difference. This gives altitude above horizon and azimuth. Skyfield's examples show how to do this to check if the ISS is above the horizon [9].

- **Real-Time Updating:** In a running app, you can loop (or use a timer) to repeatedly compute `sat.at(now)` and update the position. Even doing this every second or few seconds gives a smooth moving marker. Because Skyfield calculations are relatively fast, a simple loop can keep the track real-time.

3. Plotting the Ground Track on a Map

- **Choose a Map Library:** For interactive mapping, popular choices are Leaflet (open-source) or Mapbox GL JS. Both support adding markers and polylines on top of a world map. The project can be implemented either entirely in Python (using libraries like folium that combine Leaflet with Python) or with a Python back-end (e.g. Flask/Django) and a JavaScript front-end that handles the map.

- **Displaying a Marker:** Add a map to your page and create a marker for the satellite. Optionally use a custom icon (e.g. a satellite image). For example, in Leaflet: `L.marker([lat, lon], {icon: satelliteIcon}).addTo(map)`. Then update that marker's position on each time tick. (An ISS tracker tutorial uses Leaflet and an image of ISS as a marker [10].)

- **Drawing the Orbit Path:** You can draw the ground track path by adding a polyline through the past (or future) lat/lon points. For example:

```
L.polyline(coordinatesArray, {color: 'red'}).addTo(map);
```

where `coordinatesArray` is an array of `[lat, lon]` pairs. Update it as new positions come in (or pre-compute a full orbit's worth of points and draw it once). Be careful around the dateline: Leaflet can handle longitudes past ±180° but you may need to split the line if it crosses the map edge.

- **Mapview Updates:** Implement a timer or callback so that every second (or 5 seconds) the app fetches the latest satellite lat/lon from Python and moves the marker. In pure JS you might call an API endpoint (which runs Skyfield) or, in a Python app, update via a callback from a scheduler. The Leaflet/Mapbox documentation and examples show how to update markers and polylines dynamically.

- **Additional Map Features:** You can add more info on the map: e.g. draw the satellite's current **footprint** (area on Earth where it can see, as a circle around the subpoint), show multiple satellites (e.g. all Starlink or active spacecraft), or let the user click to get details (latitude, longitude, altitude, velocity). Interactivity like zoom/pan is handled by the map library.

4. Predicting Satellite Passes

- **Definition:** A *pass* is when a satellite rises above the horizon (visible) and then sets. We often care only about passes when the satellite is sunlit (so it's visible) and above some altitude.

- **Finding Rise/Set Times:** Skyfield's `almanac` module can compute rise/set events. Using an observer `topos = wgs84.latlon(lat, lon)` and the satellite's position, you can use `almanac.risings_and_settings(ephemeris, sat, topos)` and then `find_discrete()` to get timestamps of rises and sets. (The Skyfield examples show using `risings_and_settings` for any celestial body [11].) Alternatively, you can simply step through time and check when the topocentric altitude crosses zero.

- **Visibility (Sunlight):** Not all passes are visible – a satellite in Earth's shadow won't be lit by the Sun. Skyfield provides `is_sunlit()` to check this: it returns `True` when the satellite is in sunlight [12]. In other words, after computing the satellite's position at a time, call `sat.at(t).is_sunlit(eph)` (using an ephemeris like `de421`) to see if it's illuminated. This lets you filter out nighttime passes.

- **Using Skyfield for Passes:** As one user noted, "the best way to predict satellite passes" is to use Skyfield, which can download TLEs, propagate with SGP4, and compute pass details for any observer [13]. For example, you could find the next pass by incrementing time until altitude > 0 for a few minutes. Or use `find_discrete` on a 24-hour span to list all rise/set times, then pick those periods that coincide with `is_sunlit == True` to get visible passes.

- **Output:** Display the next pass time(s) in the UI, or draw a segment of track on the map for the upcoming minutes. You might also compute maximum elevation and duration of each pass. With Skyfield, this is all straightforward once you have the observer location and satellite object.

5. Technologies and Tools

- **Languages:** Python will run Skyfield and handle calculations. The front-end map can be done in JavaScript (with Leaflet/Mapbox). You could also use a Python web framework (Flask, Django, FastAPI) to serve data and HTML, or a desktop Python GUI toolkit (though web is more common). Some people use `folium` or `ipyleaflet` to display maps from Python directly, but JS gives more flexibility for real-time updates.

- **APIs:** CelesTrak (for TLE data). Optionally, if you want alternative data sources: Space-Track.org (requires login) or celestrak/TLE of TLE groups.

- **Libraries:** Aside from Skyfield, you might use `requests` or `urllib` for HTTP, and standard math/NumPy if needed. For mapping, Leaflet and its plugins (or Mapbox GL JS). For time handling, use Python's `datetime` or Skyfield's `Timescale`.

- **Orbital Math:** For basic understanding, recall that TLEs use the SGP4 model (not simple Keplerian elements), but Skyfield handles that detail. You just feed it the TLE and let it compute the orbit.

6. Additional Features and Improvements

Once the basic tracker is working, you can enhance it in many ways:

- **Multiple Satellites:** Let the user select one or more satellites (e.g. ISS, Starlink cluster, weather sats). You could fetch a whole group of TLEs and animate several markers.
- **Dynamic TLE Updates:** Automatically refresh TLE data on schedule (e.g. daily) to keep orbits accurate.
- **Historical Track:** Show the past orbit trail of the satellite (e.g. last 10 minutes) with a fading tail.
- **User-Specified Location:** Allow the user to enter their latitude/longitude (or use geolocation) to compute passes for their location.
- **Orbit Visualization:** In addition to 2D maps, consider a 3D globe view (e.g. with Cesium or Three.js) for a richer visualization.
- **Ground Station View:** Plot the observer's location and the line of sight to the satellite.
- **Improvements to Map:** Use satellite imagery or Mapbox's styles for nicer appearance, add layers (grid, satellite footprints), or integrate weather overlays.
- **Error Handling:** Check for TLE parsing errors or propagation failures (SGP4 can sometimes fail if TLE is corrupt).
- **Performance:** Skyfield is fast, but if tracking many objects or doing fine time resolution, consider optimizing (batch propagate times, reuse `Timescale` object).
- **User Interface:** Add labels (satellite names, time stamps), controls to pause/resume animation, and user settings for update interval.

Skyfield's documentation and community (e.g. Space.SE discussions [13]) can provide guidance on advanced topics like eclipse transitions and orbital perturbations. Overall, a Satellite Tracker ties together real data (TLE), orbital algorithms (SGP4 via Skyfield), and interactive maps, making it an excellent project that's "not too complex" yet yields impressive real-time visualization.

Sources: Information on using Skyfield for satellite orbits and passes is drawn from the Skyfield documentation and related discussions [1] [13] [7] [12]. These explain how to load TLEs, compute positions, convert to latitude/longitude, and determine visibility.

---

[1] [2] [3] [4] [5] [6] [8] [9] [12] Earth Satellites — Skyfield documentation
https://rhodesmill.org/skyfield/earth-satellites.html

[7] [11] Examples — Skyfield documentation
https://rhodesmill.org/skyfield/examples.html

[10] Build a Real-Time ISS Tracker using Javascript | by thecoding pie | Medium
https://thecodingpie.medium.com/build-a-real-time-iss-tracker-using-javascript-f3809e54ba70

[13] communication satellite - Minimum data required to predict passes - Space Exploration Stack Exchange
https://space.stackexchange.com/questions/55351/minimum-data-required-to-predict-passes