Week 7 Report

Dhanush Balusa

July 2nd, 2025

Research

Gabbard Diagram:

- Gabbard Diagram Formation: The General Theory for Elliptical Orbits: https://www.ripublication.com/aasa18/aasav8n2_01.pdf
- A Gabbard diagram plots the apogee and perigee heights of the space debris vs. the orbital periods of the space debris.
- Gabbard diagrams help in understanding the nature of the orbit, the fragmentation point, and the spread of fragments.

Conjunction analysis:

1. Spaceflight Safety Handbook For Satellite Operators: https://for-testing-only.space-track.org/documents/SFS_Handbook_For_Operators_V1.7.pdf
    - Uses Special Perturbations (SP) orbit propagation theory.
    - SP includes high-accuracy modeling of gravitational and non-gravitational forces, enabling reliable propagation of Resident Space Objects (RSOs).
    - Propagation uses data from the Space Surveillance Network (SSN) sensors worldwide. This data feeds into the High Accuracy Catalog (HAC), which is updated every 8 hours.
    - Other parameters that are important in conjunction analysis include:
        - Covariance Matrices: Quantify uncertainty in an object's position and velocity.
        - Predictive Ephemerides: Submitted by operators in formats like OEM. Predictive ephemerides are time-stamped position/velocity trajectories submitted by satellite operators, describing where their spacecraft will be in the future. These inputs allow 19 SDS to predict conjunctions involving future maneuvers and provide decision support to operators on whether an avoidance action is necessary.
        - Maneuver Intent: Operators must report maneuvers; otherwise, risk assessments may be inaccurate.
2. NASA Spacecraft Conjunction Assessment and Collision Avoidance Best Practices Handbook: https://nodis3.gsfc.nasa.gov/OCE_docs/OCE_51.pdf

- NASA advises against using TLEs, which rely on the SGP4 model, because TLEs do not include covariances and are too imprecise for collision probability calculation.
- Other parameters that are important in conjunction analysis include:
  - Predictive Ephemerides: NASA uses Orbit Ephemeris Message (OEM)-formatted data with covariances, including maneuvers. Delivered via space-track.org and NASA's internal channels.
  - Force Models Used: Drag, solar radiation pressure, and third-body perturbations are used in conjunction analysis.
  - Probability of Collision (Pc) Calculations: NASA uses 2D Pc, 3D Number of Collisions (Nc), and Monte Carlo methods.
  - Relative Geometry at TCA: TCA risk is evaluated using radial, in-track, and cross-track miss distances.

Code Changes

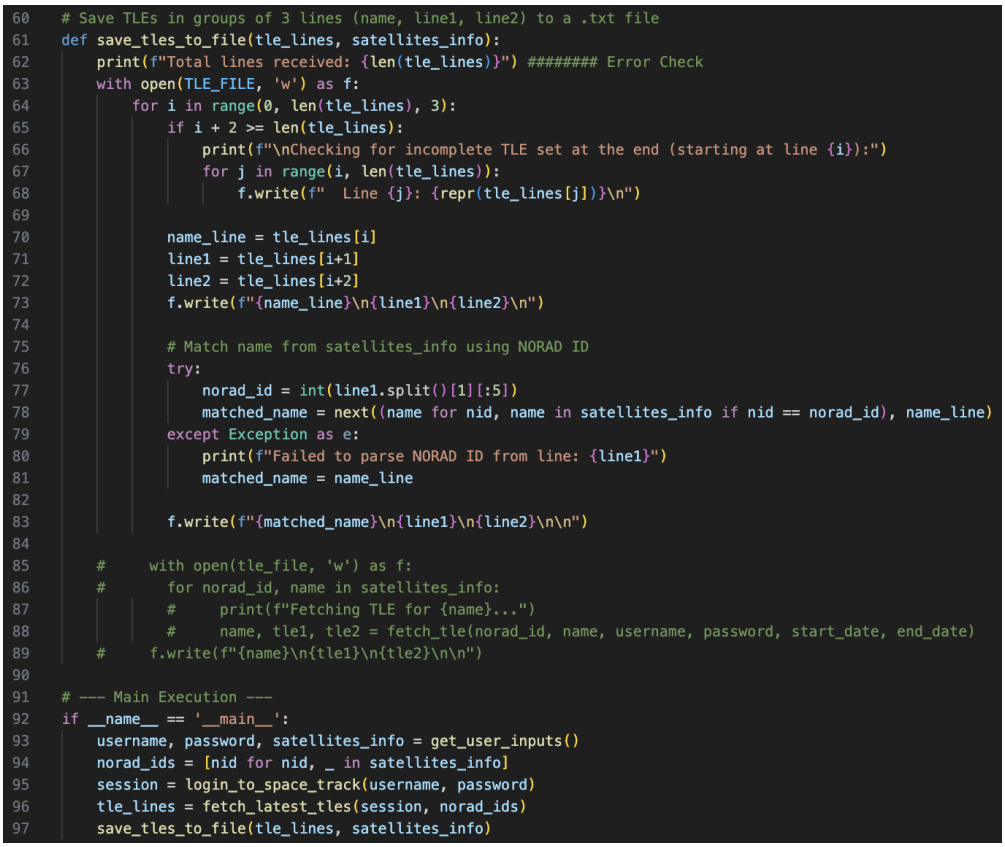
To clean up the code and based on the feedback from Space-Track.org, I decided to make the following changes:

1. **My first change was to implement my plan of action to ensure that the code is compliant with Space-Track's API rules.**

   The TLE fetching script has been updated to follow the Space-Track.org API rules.

```python
import requests
import getpass
import os

# --- Constants for Space-Track test server ---
LOGIN_URL = 'https://for-testing-only.space-track.org/ajaxauth/login'
BASE_URL = 'https://for-testing-only.space-track.org/basicspacedata/query/'
TLE_FILE = 'satellites_tle_test.txt'

# Authenticate and return a session
def login_to_space_track(username, password):
    session = requests.Session()
    login_data = {'identity': username, 'password': password}
    response = session.post(LOGIN_URL, data=login_data)
    if not response.ok:
        raise Exception(f"Login failed: {response.status_code}")
    return session

# Fetch the latest TLEs using /class/gp/endpoint
def fetch_latest_tles(session, norad_ids):
    id_string = ','.join(map(str, norad_ids))
    url = (
        f'{BASE_URL}class/gp/NORAD_CAT_ID/{id_string}'
        f'/decay_date/null-val/epoch/>now-30/format/3le'
    )
    response = session.get(url)
    if not response.ok:
        raise Exception(f"Failed to fetch TLEs: {response.status_code}")
    print(f"\nFetched raw response:\n{repr(response.text)}\n") ######## Error check
    tle_lines = [line.strip() for line in response.text.strip().split('\n') if line.strip()]
    ######## Old line: return response.text.strip().split('\n')
    return tle_lines


# Get user input
def get_user_inputs():
    username = input("Enter your Space-Track username: ")
    password = getpass.getpass("Enter your Space-Track password: ")
    num_sats = int(input("How many satellites do you want to fetch?: "))
    satellites_info = []
    for i in range(num_sats):
        norad_id = int(input(f"Enter NORAD Catalog ID for satellite #{i+1}: "))
        name = input(f"Enter name for satellite #{i+1}: ")
        satellites_info.append((norad_id, name))
    return username, password, satellites_info
```

Figure 1: Script for fetching TLE data from Space-Track.org.



```python
# Save TLEs in groups of 3 lines (name, line1, line2) to a .txt file
def save_tles_to_file(tle_lines, satellites_info):
    print(f"Total lines received: {len(tle_lines)}") ######## Error Check
    with open(TLE_FILE, 'w') as f:
        for i in range(0, len(tle_lines), 3):
            if i + 2 >= len(tle_lines):
                print(f"\nChecking for incomplete TLE set at the end (starting at line {i}):")
                for j in range(i, len(tle_lines)):
                    f.write(f"  Line {j}: {repr(tle_lines[j])}\n")

            name_line = tle_lines[i]
            line1 = tle_lines[i+1]
            line2 = tle_lines[i+2]
            f.write(f"{name_line}\n{line1}\n{line2}\n")

            # Match name from satellites_info using NORAD ID
            try:
                norad_id = int(line1.split()[1][:5])
                matched_name = next((name for nid, name in satellites_info if nid == norad_id), name_line)
            except Exception as e:
                print(f"Failed to parse NORAD ID from line: {line1}")
                matched_name = name_line

            f.write(f"{matched_name}\n{line1}\n{line2}\n\n")

    #     with open(tle_file, 'w') as f:
    #       for norad_id, name in satellites_info:
            #     print(f"Fetching TLE for {name}...")
            #     name, tle1, tle2 = fetch_tle(norad_id, name, username, password, start_date, end_date)
    #     f.write(f"{name}\n{tle1}\n{tle2}\n\n")

# --- Main Execution ---
if __name__ == '__main__':
    username, password, satellites_info = get_user_inputs()
    norad_ids = [nid for nid, _ in satellites_info]
    session = login_to_space_track(username, password)
    tle_lines = fetch_latest_tles(session, norad_ids)
    save_tles_to_file(tle_lines, satellites_info)
```

Figure 2: Script for saving TLE data to a local file.

I'm having issues with fetching the TLE data from Space-Track.org's test API. I think it's just an issue of TLE formating because I had to change that last time, but I'm not sure. I've created a lot of new functions this time, but only `fetch_latest_tles` and/or `save_tles_to_file` seem to be the problamatic ones.

Here is the output that I am getting:

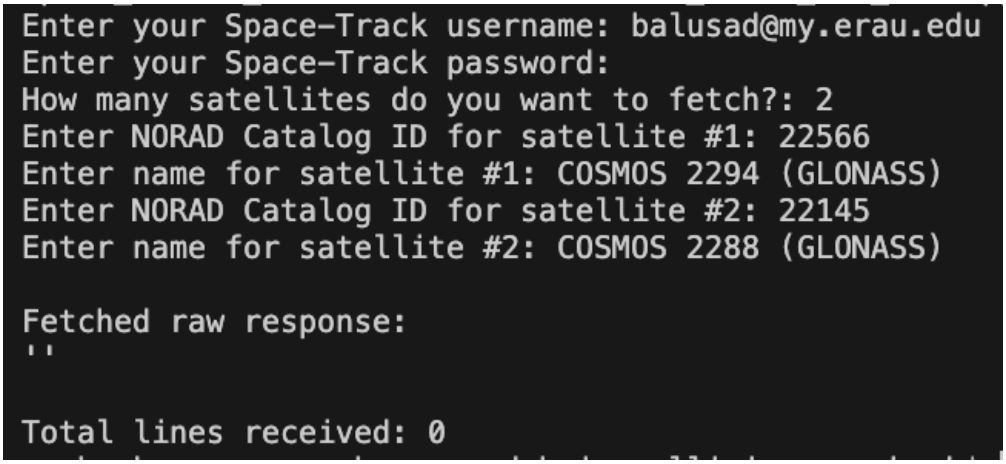


Figure 3: Output for fetching TLE conditions.

The analysis script seems to be working fine based on a fake TLE data file I created and even plots the output. It looks like it's working fine, because there is no logic change in the code.

Also, I will only be using this script for historical TLE data analysis because Space-Track's Sync.com archive already provides files of data (per year).

```python
# Load time steps
ts = load.timescale()
start_time = ts.utc(2025, 6, 19, 12, 0, 0)
hours = np.arange(0, 48, 1/1000)
time_steps = ts.utc(start_time.utc.year, start_time.utc.month, start_time.utc.day, hours)

# Calculate relative position and speed
times = []
relative_positions = {name: [] for name in names[1:]}
approach_speeds = {name: [] for name in names[1:]}

for t in time_steps:
    states = [sat.at(t) for sat in satellites]
    reference_state = states[0]
    for i, state in enumerate(states[1:], start=1):
        rel_pos = reference_state.position.km - state.position.km
        rel_vel = reference_state.velocity.km_per_s - state.velocity.km_per_s
        approach_speed = np.linalg.norm(rel_vel)
        relative_positions[names[i]].append(np.linalg.norm(rel_pos))
        approach_speeds[names[i]].append(approach_speed)
    times.append(t.utc_iso())

# Find closest approach for the first pair
target_name = names[1]
min_index = np.argmin(relative_positions[target_name])
min_distance = relative_positions[target_name][min_index]
min_speed = approach_speeds[target_name][min_index]
min_time = times[min_index]

# Calculate approach angle
closest_time = time_steps[min_index]
state_sat1 = satellites[0].at(closest_time)
state_sat2 = satellites[1].at(closest_time)

rel_pos_vector = state_sat1.position.km - state_sat2.position.km
rel_vel_vector = state_sat1.velocity.km_per_s - state_sat2.velocity.km_per_s
cos_theta = np.dot(rel_pos_vector, rel_vel_vector) / (np.linalg.norm(rel_pos_vector) * np.linalg.norm(rel_vel_vector))
approach_angle_deg = np.degrees(np.arccos(cos_theta))

# Report
print(f"\n  Near Miss Detected Between {names[0]} and {names[1]}:")
print(f"    - Closest Distance:      {min_distance:.3f} km")
print(f"    - Approach Speed:        {min_speed:.3f} km/s")
print(f"    - Approach Angle:        {approach_angle_deg:.3f} degrees")
print(f"    - Time (UTC):            {min_time}")

# Plotting
plt.figure(figsize=(10, 6))
for name, positions in relative_positions.items():
    plt.plot(times, positions, label=f"Relative Position to {name}")
plt.xlabel("Time (UTC)")
plt.ylabel("Relative Position (km)")
plt.title(f"Relative Position to {names[1]} Over 2 Days")
plt.xticks(rotation=90)
ax2 = plt.gca()
ax2.xaxis.set_major_locator(ticker.MaxNLocator(nbins=15))
plt.legend()
plt.tight_layout()
plt.show()
```

Figure 4: Script for analyzing approach conditions.

```
Enter the NORAD Catalog IDs to analyze (comma-separated): 22566,22145

  Near Miss Detected Between COSMOS 2294 and COSMOS 2288 (GLONASS):
    - Closest Distance:      18164.217 km
    - Approach Speed:        9.550 km/s
    - Approach Angle:        90.038 degrees
    - Time (UTC):            2025-06-19T13:40:23Z
```

Figure 5: Input and output for for analyzing approach conditions.

2. **My second change was to check if I find the spacecraft type (i.e. debris, rocket body, operational satellite) from the TLE data? If it's possible, I**

will plot data points coresponding to each spacecraft type (by color).

Because I was not successful in fetching the TLE data from Space-Track.org's test API, I can not surely tell. If the API provides the spacecraft name, I can try to find the spacecraft type from the name. For example, if the name contains "DEB" or "R/B", I can classify it as debris or rocket body, respectively. However, this is not a foolproof method and may not work for all cases.

Next Steps

My Next Steps:

- Debug the `fetch_latest_tles` and `save_tles_to_file` functions to ensure they function correctly. The `save_tles_to_file` function is where the code fails, especially test case `if i + 2 >= len(tle_lines)`.
- Test the new scripts to ensure they work as expected and fetch the latest TLE data correctly. Use the 4 test cases from Week 5 to verify.
- Plot the data points corresponding to each spacecraft type (by color).

Dr. Fan's feedback for next steps:

1. Read into orbit covariance, either covariance of orbital elements or states (position & velocity).
2. How to propogate covariance, analytically or numerically (Monte Carlo or sigma points).
3. How to identify collision when there is covariance information.
4. Orbit propagator using: two-body, oblateness (J2), third body (moon), drag (if possible).

End goals:

- Loop through all the TLEs in the different categories and calculate the approach conditions for each satellite if it is within a certain distance (e.g. 100 km) of the ISS.
- Incorporate covariance into space object propagation for more realistic collision identification. Also for warning time first propogate for 7 days.
- Make my code a usable function so Catherine can just call it. Some input $\rightarrow$ Output (approach speed and angle).