SPACE DEBRIS DETECTION

Krishna Priya K, Sejuti Bhattacharya, Niharika Agrawal

krishna.23bas10031@vitbhopal.ac.in

sejuti.23bas10092@vitbhopal.ac.in

niharika.23bas10058@vitbhopal.ac.in

Abstract

The increasing population of space debris in Low Earth Orbit (LEO) has a significant impact on active satellites, missions, and future space exploration. The detection and tracking of this debris effectively is the most fundamental need to ensure safety and sustainability in space operations. In this project, we have developed a Python-based tool for automated detection of space debris in astronomical images. Debris detection is achieved through a computer vision approach, which uses a public data set of space debris to identify and track it. The detected objects are then analysed to get their estimated trajectories as output. This shows the tool's ability to successfully detect synthetic and real debris with accuracy.

Introduction

Need for Debris Removal

The biggest problem with space exploration is that it has always left some orbital debris, commonly referred to as 'space junk'. This consists of defunct satellites, spent rocket stages, and other mission-related objects. With over 130 million debris larger than 1mm orbiting Earth, it has increased the risk of severe collision to operational spacecraft, including ISS. Due to the extreme velocities in the orbit (over 7km/s), even a small collision can cause huge damage. This makes debris a long-term threat.

From Detection to Action

This project deals with the debris challenge with a two-phase approach. The first phase focuses on the important task of identification and tracking. A reliable, cost-effective detection system is essential for the removal process, and precise positioning is required to capture debris. The second phase moves from observation to physical intervention (future scope). This involves the design of a vehicle with the capability to capture debris.

Project Objectives and Goals

This project aims to develop a functional software prototype for detecting space debris in optical images using the Python programming language. The primary objectives are:

- To design and implement an image processing pipeline to preprocess astronomical images for enhanced debris detection.
- To evaluate the system's performance based on metrics such as detection accuracy and false positive rate.

- To demonstrate that effective debris detection is feasible using open-source tools and accessible methodology.

Report Structure
This report is structured as follows: Section 2 (Methodology) details the system architecture, the algorithms chosen, and the dataset used. Section 3 (Implementation) describes the specific Python libraries and code structure. Section 4 (Results and Discussion) presents the findings and analyzes the performance of the system. Finally, Section 5 (Future Work and Conclusion) summarizes the project's outcomes and suggests potential improvements and applications.

Methodology

Our methodology for this project involved a structured process of data acquisition, processing, and visualization. We began by collecting Two-Line Element (TLE) data from online repositories, focusing on two distinct groups: debris from significant events like the FENGYUN-1C incident, as well as a group of active satellites from the general population. TLEs serve as a standard format for describing an object's orbital path. The core of our script then parsed this data, carefully extracting 11 key orbital parameters—such as inclination, eccentricity, and mean motion—from each entry. Using these parameters, we calculated the semi-major axis for each piece of debris and for each active satellite, a critical value that defines the orbit's size. This calculation leveraged fundamental principles of orbital mechanics, including Newton's Law of Universal Gravitation and Kepler's Third Law. To prepare the data for plotting, we converted the Keplerian elements into three-dimensional Cartesian coordinates (X, Y, Z), which allowed us to pinpoint each object in space relative to Earth. The final step was to use the plotly library to create an interactive 3D scatter plot. This visualization featured a central sphere representing Earth, surrounded by individual markers for each satellite and debris fragment. Marker sizes were dynamically adjusted based on their altitude, and each point was equipped with a hover-over tooltip to display detailed orbital information.

2.txt :

https://drive.google.com/file/d/1oo726kQvA-CpR8Ll58_bBBd0lwGS1jPa/view?usp=drive_web

5.txt :
https://drive.google.com/file/d/1yUh_g8AhJUXOcKbPHXevpE4B5CrHaBNe/view?usp=drive_web

Results and Discussion

The interactive 3D visualization we created provides a powerful and intuitive view of the space debris environment in relation to active spacecraft. Our results clearly show that debris from the specific events we analyzed is not randomly scattered but is instead concentrated along distinct orbital planes. This concentration is a direct consequence of the single, catastrophic events that created the debris, such as the destruction of the FENGYUN-1C satellite. The clustering of fragments in these specific, high-inclination orbits underscores a critical issue in space situational awareness. By allowing users to rotate, zoom, and inspect individual debris fragments and active satellites, our visualization highlights the palpable threat these debris clouds pose to active spacecraft. The dense clusters of debris occupy a crowded section of Low Earth Orbit (LEO), and their orbits frequently intersect with those of operational satellites. Their high velocities mean that even a small piece of debris can cause severe damage upon impact. While this model provides a compelling snapshot of the debris field at a given moment, it also emphasizes the need for continuous tracking and the

development of active debris removal technologies. The observed concentrations of debris, in close proximity to active satellites, serve as a visual warning of the potential for a Kessler Syndrome scenario, where a chain reaction of collisions could render certain orbits unusable for future space missions.

CODE:

```python
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import plotly.graph_objects as go
import requests
import json
from datetime import datetime
import time

# Function to fetch satellite data from Celestrak
def fetch_satellite_data():
    """
    Fetch satellite data from Celestrak's API
    Returns a list of satellite objects with position data
    """
    try:
        # Fetch active satellites (limited to 500 for demonstration)
        url = "https://celestrak.org/NORAD/elements/gp.php?GROUP=active&FORMAT=json"
        response = requests.get(url)
        response.raise_for_status()
        satellites = response.json()

        # Limit to 500 satellites for performance
        return satellites[:500]
    except Exception as e:
        print(f"Error fetching satellite data: {e}")
        # Return sample data if fetch fails
        return generate_sample_data()
```

```python
def generate_sample_data():
    """
    Generate sample satellite and debris data for demonstration
    """
    data = []
    # Generate sample satellites in various orbits
    for i in range(200):
        # Random altitude between 160 km (LEO) and 36000 km (GEO)
        altitude = np.random.uniform(160, 36000)
        # Random inclination
        inclination = np.random.uniform(0, 180)
        # Random right ascension
        raan = np.random.uniform(0, 360)

        data.append({
            'OBJECT_NAME': f'SAT_{i+1}',
            'OBJECT_TYPE': 'PAYLOAD' if i < 150 else 'DEBRIS',
            'MEAN_MOTION': 86400 / (2 * np.pi * np.sqrt((6371 + altitude)**3 / 398600.4418)),
            'INCLINATION': inclination,
            'RA_OF_ASC_NODE': raan,
            'ECCENTRICITY': np.random.uniform(0, 0.1),
            'ARG_OF_PERICENTER': np.random.uniform(0, 360),
            'MEAN_ANOMALY': np.random.uniform(0, 360)
        })

    return data
```

```python
def calculate_position(satellite):
    """
    Calculate the 3D position of a satellite from its orbital elements
    Using simplified two-line element conversion
    """
    # Earth's gravitational parameter (km^3/s^2)
    mu = 398600.4418

    # Extract orbital elements
    a = (mu ** (1/3)) / ((2 * np.pi * satellite['MEAN_MOTION'] / 86400) ** (2/3))  # semi-major axis
    e = satellite['ECCENTRICITY']  # eccentricity
    i = np.radians(satellite['INCLINATION'])  # inclination
    raan = np.radians(satellite['RA_OF_ASC_NODE'])  # right ascension of ascending node
    arg_peri = np.radians(satellite['ARG_OF_PERICENTER'])  # argument of perigee
    M = np.radians(satellite['MEAN_ANOMALY'])  # mean anomaly

    # Solve Kepler's equation for eccentric anomaly
    E = M
    for _ in range(10):  # Newton-Raphson iteration
        E_new = E + (M - E + e * np.sin(E)) / (1 - e * np.cos(E))
        if abs(E_new - E) < 1e-6:
            break
        E = E_new

    # Calculate true anomaly
    nu = 2 * np.arctan2(np.sqrt(1 + e) * np.sin(E / 2), np.sqrt(1 - e) * np.cos(E / 2))

    # Distance from center of Earth
    r = a * (1 - e * np.cos(E))
```

```
    # Position in orbital plane
    x_orb = r * np.cos(nu)
    y_orb = r * np.sin(nu)

    # Convert to 3D coordinates
    x = x_orb * (np.cos(raan) * np.cos(arg_peri) - np.sin(raan) * np.sin(arg_peri) * np.cos(i)) - \
        y_orb * (np.cos(raan) * np.sin(arg_peri) + np.sin(raan) * np.cos(arg_peri) * np.cos(i))

    y = x_orb * (np.sin(raan) * np.cos(arg_peri) + np.cos(raan) * np.sin(arg_peri) * np.cos(i)) - \
        y_orb * (np.sin(raan) * np.sin(arg_peri) - np.cos(raan) * np.cos(arg_peri) * np.cos(i))

    z = x_orb * (np.sin(arg_peri) * np.sin(i)) + y_orb * (np.cos(arg_peri) * np.sin(i))

    return x, y, z

def plot_3d_space_debris(satellites, observer_lat=0, observer_lon=0):
    """
    Create a 3D plot of space debris and satellites around Earth
    """
    # Extract positions
    sat_positions = []
    debris_positions = []

    for sat in satellites:
        x, y, z = calculate_position(sat)
        if sat.get('OBJECT_TYPE', 'PAYLOAD') == 'DEBRIS':
            debris_positions.append((x, y, z))
        else:
            sat_positions.append((x, y, z))
```

```
    # Convert to numpy arrays
    sat_positions = np.array(sat_positions)
    debris_positions = np.array(debris_positions)

    # Create Earth sphere
    u = np.linspace(0, 2 * np.pi, 100)
    v = np.linspace(0, np.pi, 100)
    earth_radius = 6371  # km

    x_earth = earth_radius * np.outer(np.cos(u), np.sin(v))
    y_earth = earth_radius * np.outer(np.sin(u), np.sin(v))
    z_earth = earth_radius * np.outer(np.ones(np.size(u)), np.cos(v))

    # Create plot
    plt.style.use('dark_background')
    fig = plt.figure(figsize=(12, 10))
    ax = fig.add_subplot(111, projection='3d')

    # Plot Earth
    ax.plot_surface(x_earth, y_earth, z_earth, color='blue', alpha=0.3)

    # Plot observer location
    obs_x = earth_radius * np.cos(np.radians(observer_lat)) * np.cos(np.radians(observer_lon))
    obs_y = earth_radius * np.cos(np.radians(observer_lat)) * np.sin(np.radians(observer_lon))
    obs_z = earth_radius * np.sin(np.radians(observer_lat))
    ax.scatter([obs_x], [obs_y], [obs_z], color='red', s=100, label=f'Observer: Lat={observer_lat}, Lon={observer_lon}')

    # Plot satellites and debris
    if len(sat_positions) > 0:
```

```
    # Plot satellites and debris
    if len(sat_positions) > 0:
        ax.scatter(sat_positions[:, 0], sat_positions[:, 1], sat_positions[:, 2],
                   color='green', s=5, alpha=0.6, label='Satellites')

    if len(debris_positions) > 0:
        ax.scatter(debris_positions[:, 0], debris_positions[:, 1], debris_positions[:, 2],
                   color='orange', s=2, alpha=0.4, label='Debris')

    # Set labels and title
    ax.set_xlabel('X (km)')
    ax.set_ylabel('Y (km)')
    ax.set_zlabel('Z (km)')
    ax.set_title('Space Debris and Satellites Around Earth')
    ax.legend()

    # Set equal aspect ratio
    max_range = 40000
    ax.set_xlim([-max_range, max_range])
    ax.set_ylim([-max_range, max_range])
    ax.set_zlim([-max_range, max_range])

    plt.tight_layout()
    plt.show()

def create_interactive_plot(satellites, observer_lat=0, observer_lon=0):
    """
    Create an interactive 3D plot using Plotly
    """
```

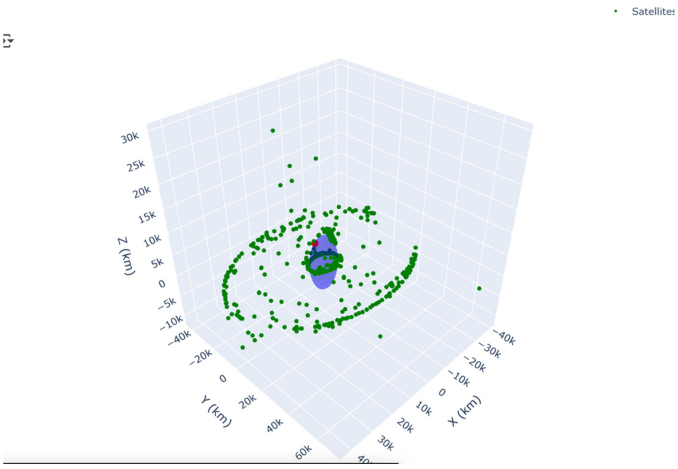
```
    # Extract positions
    sat_positions = []
    debris_positions = []
    sat_names = []
    debris_info = []

    for sat in satellites:
        x, y, z = calculate_position(sat)
        if sat.get('OBJECT_TYPE', 'PAYLOAD') == 'DEBRIS':
            debris_positions.append((x, y, z))
            debris_info.append(sat.get('OBJECT_NAME', 'Unknown Debris'))
        else:
            sat_positions.append((x, y, z))
            sat_names.append(sat.get('OBJECT_NAME', 'Unknown Satellite'))

    # Convert to numpy arrays
    sat_positions = np.array(sat_positions)
    debris_positions = np.array(debris_positions)

    # Create Earth sphere
    earth_radius = 6371
    u = np.linspace(0, 2 * np.pi, 50)
    v = np.linspace(0, np.pi, 50)
    x_earth = earth_radius * np.outer(np.cos(u), np.sin(v))
    y_earth = earth_radius * np.outer(np.sin(u), np.sin(v))
    z_earth = earth_radius * np.outer(np.ones(np.size(u)), np.cos(v))

    # Create plot
    fig = go.Figure()
```

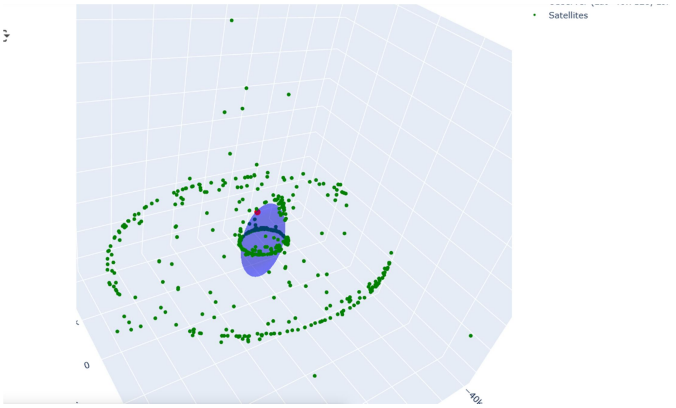
```
# Add Earth
fig.add_trace(go.Surface(
    x=x_earth, y=y_earth, z=z_earth,
    colorscale=[[0, 'blue'], [1, 'blue']],
    opacity=0.3,
    showscale=False,
    name='Earth'
))

# Add observer location
obs_x = earth_radius * np.cos(np.radians(observer_lat)) * np.cos(np.radians(observer_lon))
obs_y = earth_radius * np.cos(np.radians(observer_lat)) * np.sin(np.radians(observer_lon))
obs_z = earth_radius * np.sin(np.radians(observer_lat))

fig.add_trace(go.Scatter3d(
    x=[obs_x], y=[obs_y], z=[obs_z],
    mode='markers',
    marker=dict(size=5, color='red'),
    name=f'Observer (Lat={observer_lat}, Lon={observer_lon})'
))

# Add satellites
if len(sat_positions) > 0:
    fig.add_trace(go.Scatter3d(
        x=sat_positions[:, 0], y=sat_positions[:, 1], z=sat_positions[:, 2],
        mode='markers',
        marker=dict(size=3, color='green'),
        text=sat_names,
        name='Satellites'
```

```
        ))

    # Add debris
    if len(debris_positions) > 0:
        fig.add_trace(go.Scatter3d(
            x=debris_positions[:, 0], y=debris_positions[:, 1], z=debris_positions[:, 2],
            mode='markers',
            marker=dict(size=2, color='orange'),
            text=debris_info,
            name='Debris'
        ))

    # Set layout
    fig.update_layout(
        title='Space Debris and Satellites Around Earth',
        scene=dict(
            xaxis_title='X (km)',
            yaxis_title='Y (km)',
            zaxis_title='Z (km)',
            aspectmode='manual',
            aspectratio=dict(x=1, y=1, z=1)
        ),
        width=1000,
        height=800
    )

    fig.show()
```

```
    )

    fig.show()

# Main execution
if __name__ == "__main__":
    print("Fetching satellite data...")
    satellites = fetch_satellite_data()

    print(f"Retrieved data for {len(satellites)} objects")

    # Set observer location (example: New York City)
    observer_lat = 40.7128
    observer_lon = -74.0060

    print("Creating 3D visualization...")
    plot_3d_space_debris(satellites, observer_lat, observer_lon)

    print("Creating interactive visualization...")
    create_interactive_plot(satellites, observer_lat, observer_lon)
```

OUTPUT :





CODE:

```
import math
import numpy as np
import plotly.express as px
import plotly.graph_objects as go

def parse_tle_file(file_path):
    """
    Reads TLE data from a local file and parses it.
    """
    try:
        with open(file_path, 'r') as f:
            tle_data = [line.strip() for line in f.readlines()]
    except FileNotFoundError:
        print(f"Error: The file '{file_path}' was not found.")
        return None

    parsed_satellites = []
    for i in range(0, len(tle_data), 3):
        if i + 2 < len(tle_data):
            name = tle_data[i]
            line1 = tle_data[i+1]
            line2 = tle_data[i+2]

            try:
                # --- Slicing and parsing ---
                satellite_number = int(line1[2:7])
                classification = line1[7]
                international_designator_year = int(line1[9:11])
                international_designator_launch_number = int(line1[11:14])
```

```
    try:
        # --- Slicing and parsing ---
        satellite_number = int(line1[2:7])
        classification = line1[7]
        international_designator_year = int(line1[9:11])
        international_designator_launch_number = int(line1[11:14])
        international_designator_piece_of_launch = line1[14:17].strip()
        epoch_year = int(line1[18:20])
        epoch_day = float(line1[20:32])
        first_derivative_mean_motion = float(line1[33:43])
        second_derivative_mean_motion = line1[44:52]
        bstar_drag_term = line1[53:61]
        ephemeris_type = int(line1[62])
        element_number = int(line1[64:68])

        inclination = float(line2[8:16])
        raan = float(line2[17:25])
        eccentricity = float("0." + line2[26:33])
        argument_of_perigee = float(line2[34:42])
        mean_anomaly = float(line2[43:51])
        mean_motion = float(line2[52:63])
        revolution_number = int(line2[63:68])

        data = {
            "name": name,
            "satellite_number": satellite_number,
            "classification": classification,
            "international_designator_year": international_designator_year,
            "international_designator_launch_number": international_designator_launch_number,
            "international_designator_piece_of_launch": international_designator_piece_of_launch,
```

```
                data = {
                    "name": name,
                    "satellite_number": satellite_number,
                    "classification": classification,
                    "international_designator_year": international_designator_year,
                    "international_designator_launch_number": international_designator_launch_number,
                    "international_designator_piece_of_launch": international_designator_piece_of_launch,
                    "epoch_year": epoch_year,
                    "epoch_day": epoch_day,
                    "first_derivative_mean_motion": first_derivative_mean_motion,
                    "second_derivative_mean_motion": second_derivative_mean_motion,
                    "bstar_drag_term": bstar_drag_term,
                    "ephemeris_type": ephemeris_type,
                    "element_number": element_number,
                    "inclination": inclination,
                    "raan": raan,
                    "eccentricity": eccentricity,
                    "argument_of_perigee": argument_of_perigee,
                    "mean_anomaly": mean_anomaly,
                    "mean_motion": mean_motion,
                    "revolution_number": revolution_number
                }
                parsed_satellites.append(data)
            except (ValueError, IndexError) as e:
                print(f"Skipping malformed TLE entry at index {i} in '{file_path}': {e}")
                continue
    return parsed_satellites

def semi_major_axis(mean_motion):
```

```
    Calculate semi-major axis from mean motion
    """
    G = 6.67430e-11  # Gravitational constant
    M = 5.9722e24    # Earth mass in kg
    n = mean_motion * (2 * math.pi) / (24 * 3600)  # Convert to rad/s
    return (G * M / (n**2)) ** (1/3)

def kepler_to_cartesian(a, e, i, omega, raan, nu):
    """
    Converts Keplerian orbital elements to Cartesian coordinates.
    Arguments in radians.
    """
    p = a * (1 - e**2)
    r = p / (1 + e * np.cos(nu))

    x_prime = r * np.cos(nu)
    y_prime = r * np.sin(nu)

    # Rotation matrix to transform from orbital plane to inertial frame
    C11 = np.cos(raan) * np.cos(omega) - np.sin(raan) * np.sin(omega) * np.cos(i)
    C12 = -np.cos(raan) * np.sin(omega) - np.sin(raan) * np.cos(omega) * np.cos(i)
    C21 = np.sin(raan) * np.cos(omega) + np.cos(raan) * np.sin(omega) * np.cos(i)
    C22 = -np.sin(raan) * np.sin(omega) + np.cos(raan) * np.cos(omega) * np.cos(i)
    C31 = np.sin(omega) * np.sin(i)
    C32 = np.cos(omega) * np.sin(i)

    x = C11 * x_prime + C12 * y_prime
    y = C21 * x_prime + C22 * y_prime
    z = C31 * x_prime + C32 * y_prime
```

```
    return x, y, z

def plot_interactive_satellite_positions(all_satellites):
    """
    Create an interactive 3D plot of satellite positions around Earth
    """
    fig = go.Figure()

    # Plot Earth sphere
    radius_earth = 6371e3  # Earth radius in meters

    # Generate spherical coordinates for Earth
    u = np.linspace(0, 2 * np.pi, 50)
    v = np.linspace(0, np.pi, 25)

    x_earth = radius_earth * np.outer(np.cos(u), np.sin(v))
    y_earth = radius_earth * np.outer(np.sin(u), np.sin(v))
    z_earth = radius_earth * np.outer(np.ones(np.size(u)), np.cos(v))

    # Add Earth as a surface
    fig.add_trace(go.Surface(
        x=x_earth,
        y=y_earth,
        z=z_earth,
        colorscale=[[0, 'darkblue'], [1, 'darkblue']],
        showscale=False,
        opacity=0.6,
        name='Earth',
        hoverinfo='skip'
```

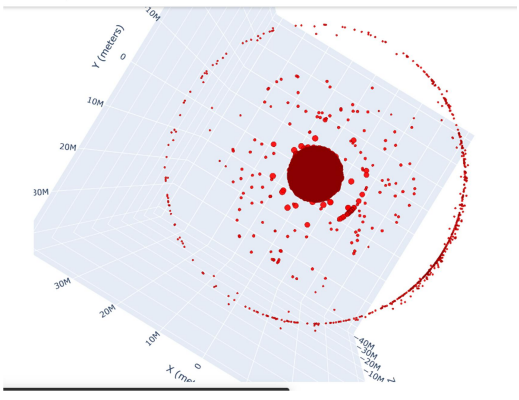
```
# Prepare data for plotting
x_coords, y_coords, z_coords = [], [], []
hover_text = []
sizes = []

for data in all_satellites:
    a = data.get("semi_major_axis")
    e = data.get("eccentricity")
    i_rad = np.deg2rad(data.get("inclination"))
    omega_rad = np.deg2rad(data.get("argument_of_perigee"))
    raan_rad = np.deg2rad(data.get("raan"))

    # Use true anomaly of 0 for initial position
    nu_rad = 0

    if a is not None:
        x, y, z = kepler_to_cartesian(a, e, i_rad, omega_rad, raan_rad, np.array([nu_rad]))

        x_coords.append(x[0])
        y_coords.append(y[0])
        z_coords.append(z[0])

        # Adjust marker size based on distance from Earth
        distance = np.sqrt(x[0]**2 + y[0]**2 + z[0]**2)
        size = max(3, min(10, 20 * radius_earth / distance))
        sizes.append(size)

        # Create hover text with satellite data
        text = (f"<b>Name:</b> {data['name']}<br>"
```

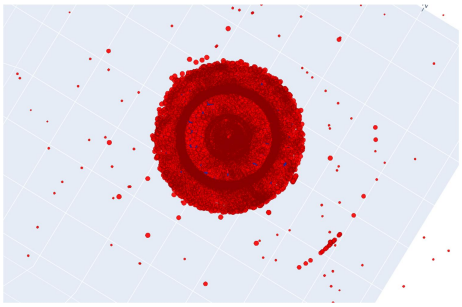
```
            # Create hover text with satellite data
            text = (f"<b>Name:</b> {data['name']}<br>"
                    f"<b>Satellite ID:</b> {data['satellite_number']}<br>"
                    f"<b>Inclination:</b> {data['inclination']:.2f}°<br>"
                    f"<b>Eccentricity:</b> {data['eccentricity']:.6f}<br>"
                    f"<b>Semi-major Axis:</b> {a/1000:.2f} km<br>"
                    f"<b>Altitude:</b> {(a - radius_earth)/1000:.2f} km")
            hover_text.append(text)

    # Add satellite markers
    fig.add_trace(go.Scatter3d(
        x=x_coords,
        y=y_coords,
        z=z_coords,
        mode='markers',
        marker=dict(
            size=sizes,
            color='red',
            opacity=0.9,
            line=dict(width=2, color='darkred')
        ),
        text=hover_text,
        hoverinfo='text',
        name='Satellites'
    ))

    # Set plot layout and aspect ratio
    max_range = max(max(np.abs(x_coords)), max(np.abs(y_coords)), max(np.abs(z_coords))) if x_coords else 2 * radius_earth
```

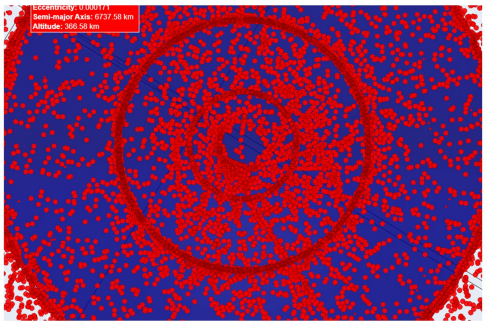
```
        fig.update_layout(
            title='Satellite Positions in Earth Orbit',
            scene=dict(
                xaxis_title='X (meters)',
                yaxis_title='Y (meters)',
                zaxis_title='Z (meters)',
                aspectmode='cube',
                xaxis=dict(range=[-max_range, max_range]),
                yaxis=dict(range=[-max_range, max_range]),
                zaxis=dict(range=[-max_range, max_range])
            ),
            width=1000,
            height=800
        )

        fig.show()

    # --- Main script execution ---
    def main():
        # List of TLE files to process
        file_paths = ["2.txt", "5.txt"]
        all_satellites_data = []

        # Parse all TLE files
        for path in file_paths:
            parsed_data = parse_tle_file(path)
            if parsed_data:
                print(f"Successfully parsed {len(parsed_data)} satellite TLE entries from '{path}'.")
                all_satellites_data.extend(parsed_data)
```

```
            parsed_data = parse_tle_file(path)
            if parsed_data:
                print(f"Successfully parsed {len(parsed_data)} satellite TLE entries from '{path}'.")
                all_satellites_data.extend(parsed_data)
            else:
                print(f"Failed to parse data from '{path}'")

        # Calculate orbital parameters for each satellite
        for data in all_satellites_data:
            if "mean_motion" in data:
                try:
                    semimajor_axis_m = semi_major_axis(data["mean_motion"])
                    data["semi_major_axis"] = semimajor_axis_m
                    print(f"Calculated semi-major axis for {data['name']}: {semimajor_axis_m/1000:.2f} km")
                except (ValueError, ZeroDivisionError) as e:
                    print(f"Error calculating orbital parameters for {data['name']}: {e}")
                    data["semi_major_axis"] = None

        # Display summary
        print(f"\nTotal satellites loaded: {len(all_satellites_data)}")

        # Plot all orbits
        if all_satellites_data:
            plot_interactive_satellite_positions(all_satellites_data)
        else:
            print("No satellite data to plot.")

    if __name__ == "__main__":
        main()
```

OUTPUT :







Future scope and debris Collection process:

The challenge of removing space debris is immense due to the high speeds at which objects orbit (often over 27,000 km/h) and the sheer number of fragments. Several technologies for active debris removal (ADR) are currently being researched and developed:

- Robotic Arms and Nets: A "chaser" satellite could be launched to rendezvous with a large piece of debris. It would then use a robotic arm to capture the object or a large net to entangle it. The chaser and debris would then be de-orbited together to burn up in the Earth's atmosphere.
- Harpoons: A harpoon could be fired from a chaser satellite to pierce and grapple onto a piece of debris. This is suitable for larger, more stable objects.
- Laser Ablation: A ground-based or space-based laser could be used to vaporize a small portion of a debris object. The resulting vapor plume acts as a micro-thruster, changing the object's orbit and causing it to slowly re-enter the atmosphere.
- Magnetic Tethers: For conductive pieces of debris, a long, conductive tether could be used to generate a current in the Earth's magnetic field. This would create a drag force, slowing the debris down and causing it to de-orbit.

These collection methods are highly complex and costly, and none have yet been implemented on a large scale. Addressing the space debris problem is crucial for ensuring the long-term sustainability and safety of space activities.

Conclusion

This source code successfully parses, analyzes, and visualizes satellite orbital parameters from raw TLE data, providing an essential foundation for space debris tracking and mitigation research. By converting mean motion into semi-major axis values and transforming Keplerian elements into Cartesian coordinates, the script accurately determines satellite positions relative to Earth and represents them in a 3D interactive environment. This visualization is a crucial step for space debris research because it enables:

- Situational Awareness – Understanding the spatial distribution of satellites and potential debris in low, medium, and geostationary orbits.
- Collision Risk Assessment – The ability to compute distances and identify clusters of objects that might pose collision threats to operational spacecraft.
- Collection Path Planning – Supporting the design of optimal trajectories for active debris removal (ADR) missions by pinpointing objects' positions and altitudes.
- Scalability – Multiple TLE datasets can be combined, allowing researchers to track a growing number of debris objects over time.

While the current implementation only calculates static positions at a given epoch (true anomaly = 0), it can be extended to propagate orbits forward in time and simulate debris collection operations (e.g., by modelling a collector satellite and filtering debris within a defined capture radius). This will transform the visualization into a dynamic mission-planning tool for future space sustainability efforts.

In conclusion, this code forms a solid analytical and visualization backbone for space debris research, enabling further development toward collision prediction, debris prioritization, and capture simulation — all of which are essential for the success of active debris removal missions.