```python
from astropy import units as u
from skyfield.api import Topos, load
from skyfield.api import EarthSatellite
from datetime import datetime, timedelta
import matplotlib.pyplot as plt
from matplotlib.collections import LineCollection
from matplotlib.dates import date2num
from matplotlib.patches import Patch
from pprint import pprint
from numpy import sign
from pathlib import Path
from dateutil import parser, tz
from functools import reduce
import json

# Load earth and time
planets = load("de421.bsp")
earth = planets["earth"]
ts = load.timescale()

def load_input():
    try:
        with open("./input.json") as f:
            data = json.load(f)
            now = datetime.now(tz.tzlocal())
            tomorrow = now + timedelta(days=1)
            t0 = ts.from_datetime(parser.parse(data["startTime"], default=now))
            t1 = ts.from_datetime(parser.parse(data["endTime"], default=tomorrow))
            antenna_position = Topos(data["antennaPosition"]["lat"], data["antennaPosition"]["lon"], elevation_m=data["antennaPosition"]["elevation"])
            return (t0, t1, data["satellite"], antenna_position, data["beamwidth"], data["azimuthTurnRate"], data["elevationTurnRate"], data["sampleRate"], data["bothAxis"], data["outputFile"])
    except FileNotFoundError:
        print("File \"input.json\" not found")
    except Exception as e:
        print(e)
        print("Error while parsing \"input.json\"")

(t0, t1, satellite_name, antenna_position, beamwidth, azimuth_turn_rate, elevation_turn_rate, sample_rate, both_axis, output_file) = load_input()

# granularity for turning of the antena
azimuth_turn_granularity = 1
elevation_turn_granularity = 1

# loading TLE files from celestrak
stations_url = "https://www.celestrak.com/NORAD/elements/active.txt"
satellites = load.tle_file(stations_url, reload=False)
print("Loaded", len(satellites), "satellites")

by_name = {sat.name: sat for sat in satellites}
satellite = by_name[satellite_name]

if not satellite:
    print(f"Could not find satellite {satellite_name} in the celestrak TLE file")
    exit()

print(f"Selected satellite: {satellite.name}")
```

```python

# Finding rising and setting times
t, events = satellite.find_events(antenna_position, t0, t1, altitude_degrees=0.0)
ranges = []
start = None
for ti, event in zip(t, events):
    if event == 0:
        start = ti
    elif event == 2:
        ranges.append((start, ti))

    name = ("rise above horizon", "culminate", "set below horizon")[event]

def calculate_rotation_diff(sourceAngle, targetAngle):
    diff = sourceAngle - targetAngle
    if (abs(diff) > 180):
        diff = sourceAngle - (targetAngle + sign(diff) * 360)
    return diff

def round_to_multiple(n, x):
    return x * round(n / x)

class Antenna:
    def __init__(self, position, beamwidth, azimuth_turn_rate, elevation_turn_rate, azimuth_turn_granularity, elevation_turn_granularity, both_axis):
        self.position = position
        self.beamwidth = beamwidth
        self.azimuth_turn_rate = azimuth_turn_rate
        self.elevation_turn_rate = elevation_turn_rate
        self.azimuth_turn_granularity = azimuth_turn_granularity
        self.elevation_turn_granularity = elevation_turn_granularity
        self.azimuth = 0.0
        self.elevation = 0.0
        self.both_axis = both_axis

    def turn(self, azimuth, elevation):
        azimuth_step = round_to_multiple(min(abs(azimuth), self.azimuth_turn_rate), self.azimuth_turn_granularity)
        elevation_step = round_to_multiple(min(abs(elevation), self.elevation_turn_rate), self.elevation_turn_granularity)

        if both_axis or abs(azimuth) >= abs(elevation) and abs(azimuth) >= self.azimuth_turn_granularity / 2:
            self.azimuth += sign(azimuth) * azimuth_step
            self.azimuth %= 360

        if both_axis or abs(elevation) > abs(azimuth) and abs(elevation) >= self.elevation_turn_granularity / 2:
            self.elevation += sign(elevation) * elevation_step
            self.elevation %= 360

antenna = Antenna(antenna_position, beamwidth, azimuth_turn_rate, elevation_turn_rate, azimuth_turn_granularity, elevation_turn_granularity, both_axis)
entries = []

# Simulating antenna tracking
for start, end in ranges:
    tx = start

```

```python
    print("from", start.utc_datetime().strftime("%Y-%m-%d %H:%M:%S"), "to", end.utc_datetime().strftime("%Y-%m-%d %H:%M:%S"))

    difference = satellite - antenna_position
    topocentric = difference.at(start)
    elevation_diff, azimuth_diff, distance = topocentric.altaz()

    # reset antenna azimuth and elevation
    antenna.azimuth = round_to_multiple(azimuth_diff.degrees, antenna.azimuth_turn_granularity)
    antenna.elevation = 0.0

    event_entry = []

    while tx.utc_datetime() < end.utc_datetime():
        # Calculate target azimuth and elevation
        difference = satellite - antenna_position
        topocentric = difference.at(tx)
        elevation, azimuth, distance = topocentric.altaz()

        # Calculate error and update antenna position
        azimuth_diff = calculate_rotation_diff(azimuth.degrees, antenna.azimuth)
        elevation_diff = calculate_rotation_diff(elevation.degrees, antenna.elevation)

        azimuth_in_beamwidth = bool(abs(azimuth_diff) < beamwidth / 2)
        elevation_in_beamwidth = bool(abs(elevation_diff) < beamwidth / 2)

        event_entry.append((tx.utc_datetime(), antenna.azimuth, antenna.elevation, azimuth.degrees, elevation.degrees, azimuth_diff, elevation_diff, azimuth_in_beamwidth, elevation_in_beamwidth))

        antenna.turn(azimuth_diff, elevation_diff)

        # next sample
        tx += timedelta(seconds = sample_rate)

    entries.append(event_entry)

def map_entry(entry):
    return {
        "time": entry[0].strftime("%Y-%m-%d %H:%M:%S"),
        "antennaAzimuth": entry[1],
        "antennaElevation": entry[2],
        "realAzimuth": entry[3],
        "realElevation": entry[4],
        "azimuthError": entry[5],
        "elevationError": entry[6],
        "azimuthInBeamwidth": entry[7],
        "elevationInBeamwidth": entry[8],
    }

if output_file:
    serializable_entries = [list(map(map_entry, entry)) for entry in entries]

    # dump data in json
    json_data = json.dumps(serializable_entries)
    with open('data.json', 'w') as f:
        f.write(json_data)

```

```python
# create directory for plots if not exists
path = Path("./plots")
path.mkdir(parents=True, exist_ok=True)

# plot the results
for index, entry in enumerate(entries):
    time, antenna_azimuth, antenna_elevation, real_azimuth, real_elevation, azimuth_diff, elevation_diff, azimuth_in_beamwidth, elevation_in_beamwidth = zip(*entry)
    time_outside_beamwidth = reduce(lambda acc, entry: acc + (sample_rate if not entry[7] or not entry[8] else 0), entry, 0)

    fig, (azimuth_ax1, azimuth_ax2) = plt.subplots(2, 1, sharex=True, dpi=400, figsize=(20, 10))
    plt.tight_layout(pad=4.0)

    azimuth_ax1.title.set_text("Azimuth")
    azimuth_ax1.plot(time, real_azimuth, label="ideal")
    azimuth_ax1.plot(time, antenna_azimuth, label="simulation")
    azimuth_ax1.legend()
    azimuth_ax1.set_ylabel("deg")

    azimuth_ax2.title.set_text(f"Azimuth Error (outside beamwidth: {time_outside_beamwidth}s)")
    colors = ["b" if visible else "r" for visible in azimuth_in_beamwidth]
    lines = [((date2num(x0), y0), (date2num(x1), y1)) for x0, y0, x1, y1 in zip(time[:-1], azimuth_diff[:-1], time[1:], azimuth_diff[1:])]
    line_collection = LineCollection(lines, colors = colors)
    azimuth_ax2.add_collection(line_collection)
    azimuth_ax2.autoscale_view()
    azimuth_ax2.legend(handles=[Patch(color='b', label='in beamwidth'), Patch(color='r', label='not in beamwidth')])
    azimuth_ax2.set_ylabel("deg")

    plt.setp(azimuth_ax2.get_xticklabels(), rotation=30, horizontalalignment='right')
    plt.savefig(f"./plots/azimuth-{index}.pdf")
    plt.close(fig)

    fig, (elevation_ax1, elevation_ax2) = plt.subplots(2, 1, sharex=True, dpi=400, figsize=(20, 10))
    plt.tight_layout(pad=4.0)

    elevation_ax1.title.set_text("Elevation")
    elevation_ax1.plot(time, real_elevation, label="ideal")
    elevation_ax1.plot(time, antenna_elevation, label="simulation")
    elevation_ax1.legend()
    elevation_ax1.set_ylabel("deg")

    elevation_ax2.title.set_text(f"Elevation Error (outside beamwidth: {time_outside_beamwidth}s)")
    colors = ["b" if visible else "r" for visible in elevation_in_beamwidth]
    lines = [((date2num(x0), y0), (date2num(x1), y1)) for x0, y0, x1, y1 in zip(time[:-1], elevation_diff[:-1], time[1:], elevation_diff[1:])]
    line_collection = LineCollection(lines, colors = colors)
    elevation_ax2.add_collection(line_collection)
    elevation_ax2.autoscale_view()
    elevation_ax2.legend(handles=[Patch(color='b', label='in beamwidth'), Patch(color='r', label='not in beamwidth')])
    elevation_ax2.set_ylabel("deg")
```

```
    elevation_ax2.set_xlabel("time")

    plt.setp(azimuth_ax2.get_xticklabels(), rotation=30,
horizontalalignment='right')
    plt.setp(elevation_ax2.get_xticklabels(), rotation=30,
horizontalalignment='right')
    plt.savefig(f"./plots/elevation-{index}.pdf")
    plt.close(fig)


```