**Team 1 and 2 code**

pip install numpy pandas sgp4 astropy scipy

#!/usr/bin/env python3

"""

Operator node simulation:

- propagate TLE (sgp4)

- convert TEME -> ITRS -> lat/lon/alt (astropy)

- add random sensor bias + gaussian noise

- compute uncertainty ellipse (2D) from covariance estimate

- compute grid-cell risk (simple gaussian kernel on ground cells)

- anonymize outputs and save JSON

"""

import json

import hashlib

import numpy as np

import pandas as pd

from sgp4.api import Satrec, jday

from datetime import datetime, timedelta

from astropy.time import Time

from astropy import units as u

from astropy.coordinates import TEME, ITRS, CartesianRepresentation, CartesianDifferential, EarthLocation

from scipy.spatial.distance import cdist

# -------------------------

# Configuration

# -------------------------

CONFIG = {

"node\_name": "operator\_node\_alpha", # will be anonymized in final JSON

"tle": [

"1 25544U 98067A 25304.90585560 .00002182 00000+0 47424-4 0 9990",

"2 25544 51.6444 12.3456 0009896 123.4567 236.5433 15.50000000 12"

],

"start\_utc": "2025-10-03T00:00:00Z", # ISO-8601 UTC

"duration\_minutes": 120,

"step\_seconds": 60,

"sensor": {

"position\_bias\_m": [10.0, -7.0, 5.0], # constant bias (m) applied to x,y,z

"position\_noise\_std\_m": 25.0, # Gaussian noise std dev (m)

"attitude\_noise\_deg": 0.5, # example

},

"grid": {

"lat\_step\_deg": 0.5,

"lon\_step\_deg": 0.5,

"extent\_km": 2000, # kernel extent for risk aggregation

"risk\_kernel\_sigma\_km": 200.0

},

"anonymize\_salt": "replace\_with\_operator\_provided\_salt",

"max\_grid\_cells\_returned": 200

}

# -------------------------

# Helpers

# -------------------------

def anonymize\_id(raw: str, salt: str) -> str:

"""Return a short anonymized id for any raw string using SHA256."""

h = hashlib.sha256((salt + raw).encode("utf-8")).hexdigest()

return "sat-" + h[:12]

def propagate\_tle\_to\_states(tle\_lines, start\_utc\_iso, duration\_minutes, step\_seconds):

"""Propagate TLE using sgp4 and return list of state dicts: time, r\_teme\_km, v\_teme\_km\_s."""

sat = Satrec.twoline2rv(tle\_lines[0], tle\_lines[1])

t0 = datetime.fromisoformat(start\_utc\_iso.replace("Z", "+00:00"))

n\_steps = int((duration\_minutes \* 60) / step\_seconds) + 1

states = []

for i in range(n\_steps):

t = t0 + timedelta(seconds=i \* step\_seconds)

jd, fr = jday(t.year, t.month, t.day, t.hour, t.minute, t.second + t.microsecond\*1e-6)

e, r, v = sat.sgp4(jd, fr)

if e != 0:

# sgp4 returns error code in e; handle gracefully

raise RuntimeError(f"SGP4 propagation error code {e} at time {t.isoformat()}")

# r, v are in TEME frame km and km/s

states.append({"utc": t.isoformat() + "Z", "r\_teme\_km": np.array(r), "v\_teme\_km\_s": np.array(v)})

return states

def teme\_to\_latlonalt(state\_r\_km, utc\_time):

"""Convert TEME position vector (km) at utc\_time -> lat, lon, alt using astropy."""

# astropy expects time and TEME representation

t = Time(utc\_time) # expects ISO

# Create TEME cartesian

r = CartesianRepresentation(state\_r\_km \* u.km)

teme\_coord = TEME(r, obstime=t)

# Transform to ITRS (ECEF) then to lat/lon/height (geodetic)

itrs = teme\_coord.transform\_to(ITRS(obstime=t))

# Access geodetic lat lon alt via EarthLocation

x = itrs.x.to(u.m).value

y = itrs.y.to(u.m).value

z = itrs.z.to(u.m).value

el = EarthLocation(x=x\*u.m, y=y\*u.m, z=z\*u.m)

lat = el.lat.deg

lon = el.lon.deg

alt\_m = el.height.to(u.m).value

return lat, lon, alt\_m

def add\_sensor\_noise\_and\_bias(r\_km, config\_sensor):

"""Simulate sensor: add constant bias and gaussian noise. Returns noisy measurement in meters."""

r\_m = np.array(r\_km) \* 1000.0

bias = np.array(config\_sensor["position\_bias\_m"])

noise = np.random.normal(0.0, config\_sensor["position\_noise\_std\_m"], size=3)

measured = r\_m + bias + noise

# Simple covariance estimate: diag(noise\_std^2) in meters^2

cov = np.diag([config\_sensor["position\_noise\_std\_m"]\*\*2]\*3)

return measured, cov

def cov\_to\_uncertainty\_ellipse\_2d(cov\_3x3):

"""

Create a 2D uncertainty ellipse (east,north) from the 3x3 covariance in ECEF m^2.

We'll project to a local ENU plane by taking the top-left 2x2 of the covariance as a simplification.

Returns semi-major (m), semi-minor (m), angle\_deg (east->north).

"""

cov2 = cov\_3x3[:2, :2]

vals, vecs = np.linalg.eigh(cov2)

# sort descending

order = np.argsort(vals)[::-1]

vals = vals[order]

vecs = vecs[:, order]

semi\_major = np.sqrt(vals[0]) \* 2.4477 # 95% ~ chi2 factor ~2.4477 (sqrt of chi2(2,0.95))

semi\_minor = np.sqrt(vals[1]) \* 2.4477

# angle of major axis wrt east axis

angle\_rad = np.arctan2(vecs[1, 0], vecs[0, 0])

angle\_deg = np.degrees(angle\_rad)

return float(semi\_major), float(semi\_minor), float(angle\_deg)

def build\_grid(lat\_min=-90, lat\_max=90, lon\_min=-180, lon\_max=180, lat\_step=0.5, lon\_step=0.5):

lats = np.arange(lat\_min, lat\_max + 1e-6, lat\_step)

lons = np.arange(lon\_min, lon\_max + 1e-6, lon\_step)

latg, longg = np.meshgrid(lats, lons, indexing='ij')

flat = pd.DataFrame({

"lat": latg.ravel(),

"lon": longg.ravel()

})

return flat

def haversine\_km(lat1, lon1, lat2, lon2):

"""Great circle distance in km between two arrays (vectorized)."""

R = 6371.0

lat1r = np.radians(lat1)

lat2r = np.radians(lat2)

dlat = lat2r - lat1r

dlon = np.radians(lon2 - lon1)

a = np.sin(dlat/2.0)\*2 + np.cos(lat1r)\*np.cos(lat2r)\*np.sin(dlon/2.0)\*2

c = 2 \* np.arcsin(np.sqrt(a))

return R \* c

def compute\_grid\_risk(subsat\_points, grid\_df, kernel\_sigma\_km=200.0, extent\_km=2000.0):

"""

For each subsatellite point (lat,lon), build/add a gaussian kernel of risk to grid cells.

subsat\_points: list of (lat, lon) tuples

grid\_df: DataFrame with 'lat','lon'

Returns a DataFrame with an added 'risk' column.

"""

risk = np.zeros(len(grid\_df))

# Precompute cell centers arrays

cell\_lats = grid\_df['lat'].values

cell\_lons = grid\_df['lon'].values

# For efficiency, only consider subset within extent\_km of each point

for latc, lonc in subsat\_points:

dists = haversine\_km(latc, lonc, cell\_lats, cell\_lons)

mask = dists <= extent\_km

if not np.any(mask):

continue

# gaussian kernel

kernel\_vals = np.exp(-0.5 \* (dists[mask] / kernel\_sigma\_km)\*\*2)

risk[mask] += kernel\_vals

# Normalize risk to 0-1

if np.max(risk) > 0:

risk = risk / np.max(risk)

grid\_df = grid\_df.copy()

grid\_df['risk'] = (risk).astype(float)

return grid\_df

# -------------------------

# Main simulation routine

# -------------------------

def run\_simulation(config):

# 1) propagate

raw\_states = propagate\_tle\_to\_states(

config["tle"], config["start\_utc"], config["duration\_minutes"], config["step\_seconds"]

)

# 2) for each state compute lat/lon and sensor measurements

out\_states = []

subsat\_list = []

for st in raw\_states:

utc = st['utc']

r\_teme\_km = st['r\_teme\_km']

v\_teme\_km\_s = st['v\_teme\_km\_s']

try:

lat, lon, alt\_m = teme\_to\_latlonalt(r\_teme\_km, utc)

except Exception as e:

# fallback: mark NaN but continue

lat, lon, alt\_m = float('nan'), float('nan'), float('nan')

measured\_m, cov\_m = add\_sensor\_noise\_and\_bias(r\_teme\_km, config["sensor"])

# Compute simplified uncertainty ellipse (projected)

semi\_major\_m, semi\_minor\_m, angle\_deg = cov\_to\_uncertainty\_ellipse\_2d(cov\_m)

out\_states.append({

"utc": utc,

"r\_teme\_km": r\_teme\_km.tolist(),

"v\_teme\_km\_s": v\_teme\_km\_s.tolist(),

"subsat\_lat\_deg": float(lat),

"subsat\_lon\_deg": float(lon),

"subsat\_alt\_m": float(alt\_m),

"measured\_position\_m": measured\_m.tolist(),

"measurement\_cov\_m2": cov\_m.tolist(),

"uncertainty\_ellipse\_m": {

"semi\_major\_m": semi\_major\_m,

"semi\_minor\_m": semi\_minor\_m,

"orientation\_deg": angle\_deg

}

})

if not np.isnan(lat):

subsat\_list.append((lat, lon))

# 3) grid risk

grid = build\_grid(lat\_step=config["grid"]["lat\_step\_deg"], lon\_step=config["grid"]["lon\_step\_deg"])

grid\_with\_risk = compute\_grid\_risk(subsat\_list, grid, kernel\_sigma\_km=config["grid"]["risk\_kernel\_sigma\_km"], extent\_km=config["grid"]["extent\_km"])

# We will keep only top N grid cells by risk to avoid huge JSON

top\_cells = grid\_with\_risk.sort\_values("risk", ascending=False).head(config["max\_grid\_cells\_returned"])

grid\_cells\_out = top\_cells.to\_dict(orient='records')

# 4) anonymize metadata

raw\_id = config.get("tle", [""])[1] # use second line (contains NORAD id) as raw string to anonymize

anon\_sat\_id = anonymize\_id(raw\_id, config["anonymize\_salt"])

anon\_node\_id = anonymize\_id(config.get("node\_name", "node"), config["anonymize\_salt"])

summary = {

"node\_id": anon\_node\_id,

"sat\_id": anon\_sat\_id,

"start\_utc": config["start\_utc"],

"duration\_minutes": config["duration\_minutes"],

"step\_seconds": config["step\_seconds"],

"states": out\_states,

"grid\_cells": grid\_cells\_out,

"metadata": {

"sensor\_model": {

"position\_bias\_m": None, # explicitly remove operator actual bias to preserve anonymization

"position\_noise\_std\_m": config["sensor"]["position\_noise\_std\_m"]

},

"grid": {

"lat\_step\_deg": config["grid"]["lat\_step\_deg"],

"lon\_step\_deg": config["grid"]["lon\_step\_deg"]

}

}

}

return summary

# -------------------------

# Execute and write JSON

# -------------------------

if \_name\_ == "\_main\_":

np.random.seed(42) # deterministic example

summary = run\_simulation(CONFIG)

out\_file = "anonymized\_operator\_node\_summary.json"

with open(out\_file, "w") as fh:

json.dump(summary, fh, indent=2)

print(f"Wrote anonymized JSON summary -> {out\_file}")