#!/usr/bin/env python3

"""

Operator node simulation for Team 3:

- Propagate TLE (sgp4).

- Add random sensor bias + gaussian noise.

- \*\*\* NEW: Implement a simple Kalman Filter (EKF) to produce a local orbit estimate from noisy measurements. \*\*\*

- Compute grid-cell risk from the filtered estimate.

- 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

from astropy.coordinates import EarthLocation

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

# Configuration for Node 3

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

CONFIG = {

"node\_name": "operator\_node\_gamma", # A new name for the third node

"tle": [

"1 43226U 18017A 25305.54583333 .00001234 00000-0 24691-4 0 9993",

"2 43226 97.4442 211.2323 0008544 263.6345 96.4433 15.24453333 98765"

],

"start\_utc": "2025-10-03T00:00:00Z",

"duration\_minutes": 120,

"step\_seconds": 60,

"sensor": {

"position\_bias\_m": [-5.0, 12.0, -8.0], # Different bias for this operator

"position\_noise\_std\_m": 30.0, # Slightly different noise level

},

"grid": {

"lat\_step\_deg": 0.5,

"lon\_step\_deg": 0.5,

"extent\_km": 2000,

"risk\_kernel\_sigma\_km": 200.0

},

"anonymize\_salt": "a\_completely\_different\_salt\_for\_gamma",

"max\_grid\_cells\_returned": 200

}

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

# NEW: Kalman Filter Implementation

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

class SimpleEKF:

"""A simple Extended Kalman Filter for orbit estimation."""

def \_\_init\_\_(self, initial\_state, initial\_covariance, process\_noise\_std, measurement\_noise\_std):

# State vector [x, y, z, vx, vy, vz] in meters and m/s

self.x = initial\_state

# State covariance matrix P

self.P = initial\_covariance

# Process noise covariance Q (uncertainty in our physics model)

self.Q = np.diag([process\_noise\_std\*\*2] \* 6)

# Measurement noise covariance R

self.R = np.diag([measurement\_noise\_std\*\*2] \* 3)

# Measurement matrix H (we only measure position, not velocity)

self.H = np.zeros((3, 6))

self.H[0, 0] = 1

self.H[1, 1] = 1

self.H[2, 2] = 1

def predict(self, dt):

"""Predict the next state and covariance."""

# State transition matrix F (simple constant velocity model)

F = np.identity(6)

F[0, 3] = dt

F[1, 4] = dt

F[2, 5] = dt

# Predict state and covariance

self.x = F @ self.x

self.P = F @ self.P @ F.T + self.Q

def update(self, measurement\_xyz):

"""Update the state with a new measurement."""

# Innovation or measurement residual

y = measurement\_xyz - self.H @ self.x

# Innovation covariance

S = self.H @ self.P @ self.H.T + self.R

# Kalman gain

K = self.P @ self.H.T @ np.linalg.inv(S)

# Update state and covariance

self.x = self.x + K @ y

self.P = (np.identity(6) - K @ self.H) @ self.P

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

# Helpers (mostly unchanged)

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

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

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):

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: raise RuntimeError(f"SGP4 propagation error code {e}")

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):

t = Time(utc\_time)

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

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

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

el = EarthLocation(x=itrs.x, y=itrs.y, z=itrs.z)

return el.lat.deg, el.lon.deg, el.height.to(u.m).value

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

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

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

return measured, cov

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

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

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

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

vals, vecs = vals[order], vecs[:, order]

semi\_major = np.sqrt(vals[0]) \* 2.4477

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

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

return float(semi\_major), float(semi\_minor), np.degrees(angle\_rad)

def build\_grid(lat\_step, lon\_step):

lats = np.arange(-90, 90 + 1e-6, lat\_step)

lons = np.arange(-180, 180 + 1e-6, lon\_step)

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

return pd.DataFrame({"lat": latg.ravel(), "lon": longg.ravel()})

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

R = 6371.0

lat1r, lat2r, dlon = map(np.radians, [lat1, lat2, lon2 - lon1])

dlat = lat2r - lat1r

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

return R \* 2 \* np.arcsin(np.sqrt(a))

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

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

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

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

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

risk[mask] += kernel\_vals

if np.max(risk) > 0: risk /= np.max(risk)

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

return grid\_df

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

# Main simulation routine

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

def run\_simulation(config):

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

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

)

# Initialize the Kalman Filter with the first measurement

first\_state = raw\_states[0]

initial\_measured\_pos, initial\_cov = add\_sensor\_noise\_and\_bias(first\_state['r\_teme\_km'], config["sensor"])

initial\_vel\_ms = first\_state['v\_teme\_km\_s'] \* 1000.0 # Use true velocity for initial guess

initial\_state\_vec = np.hstack([initial\_measured\_pos, initial\_vel\_ms])

initial\_P = np.diag([config["sensor"]["position\_noise\_std\_m"]\*\*2]\*3 + [100\*\*2]\*3) # High uncertainty on initial velocity

ekf = SimpleEKF(

initial\_state=initial\_state\_vec,

initial\_covariance=initial\_P,

process\_noise\_std=1.0, # low process noise, we trust our physics model

measurement\_noise\_std=config["sensor"]["position\_noise\_std\_m"]

)

out\_states = []

subsat\_list\_filtered = []

for i, st in enumerate(raw\_states):

utc = st['utc']

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

# Simulate the noisy measurement

measured\_pos\_m, measurement\_cov\_m2 = add\_sensor\_noise\_and\_bias(r\_teme\_km, config["sensor"])

# EKF: Predict and Update

if i > 0: # Don't predict for the first step

dt = config["step\_seconds"]

ekf.predict(dt)

ekf.update(measured\_pos\_m)

# Extract filtered state

filtered\_pos\_m = ekf.x[:3]

filtered\_vel\_ms = ekf.x[3:]

filtered\_cov\_m2 = ekf.P

# Convert filtered position to lat/lon for grid risk

try:

lat, lon, alt\_m = teme\_to\_latlonalt(filtered\_pos\_m / 1000.0, utc) # use filtered pos

if not np.isnan(lat):

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

except Exception:

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

semi\_major\_m, semi\_minor\_m, angle\_deg = cov\_to\_uncertainty\_ellipse\_2d(filtered\_cov\_m2)

out\_states.append({

"utc": utc,

"ground\_truth\_teme\_km": {"r": r\_teme\_km.tolist(), "v": v\_teme\_km\_s.tolist()},

"noisy\_measurement": {"r\_m": measured\_pos\_m.tolist(), "cov\_m2": measurement\_cov\_m2.tolist()},

"filtered\_state\_estimate": {

"r\_teme\_m": filtered\_pos\_m.tolist(),

"v\_teme\_ms": filtered\_vel\_ms.tolist(),

"cov\_m2": filtered\_cov\_m2.tolist(),

"subsat\_lat\_deg": lat, "subsat\_lon\_deg": lon, "subsat\_alt\_m": alt\_m,

"uncertainty\_ellipse\_m": {"semi\_major\_m": semi\_major\_m, "semi\_minor\_m": semi\_minor\_m, "orientation\_deg": angle\_deg}

}

})

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

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

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

anon\_sat\_id = anonymize\_id(config["tle"][1], config["anonymize\_salt"])

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

summary = {

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

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

"states": out\_states,

"grid\_cells": top\_cells.to\_dict(orient='records'),

}

return summary

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

# Execute and write JSON

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

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

np.random.seed(1337) # Different seed for different results

summary = run\_simulation(CONFIG)

out\_file = "anonymized\_operator\_node\_3\_summary.json"

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

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

print(f"Wrote anonymized JSON summary for Node 3 -> {out\_file}")