Physics-informed Trajectory POI Detection Pipeline

Dated: December 29, 2025

1 Preprocessing the Flight Data

1.1 Coordinate Conversion: WGS84 Geodetic to ECEF

Given:

- latitude φ (rad)
- longitude λ (rad)
- ellipsoidal height h (m)
- WGS84 parameters:
  - semi-major axis $a = 6378137.0$
  - flattening rate $f = \frac{1}{298.257223563}$
  - first eccentricity squared $e^2 = 6.69437999014 \times 10^{-3}$

First compute the prime vertical radius of curvature:

$$N(\varphi) = \frac{a}{\sqrt{1 - e^2 \sin^2 \varphi}} \tag{1.1.1}$$

Then ECEF coordinates (x, y, z):

$$\begin{aligned} x &= (N(\varphi) + h) \cos\varphi \cos\lambda \\ y &= (N(\varphi) + h) \cos\varphi \sin\lambda \\ z &= \big(N(\varphi)(1 - e^2) + h\big) \sin\varphi \end{aligned} \tag{1.1.2}$$

Hence we get the ENU coordinates.

1.2 Coordinate Conversion: ECEF to ENU Conversion

Pick a reference point (the origin of the local ENU frame in this case) with geodetic coordinates $(\varphi_0, \lambda_0, h_0)$, and compute its ECEF coordinates (x_0, y_0, z_0) using the same equations as above.

For any point with ECEF (x, y, z), define the difference vector:

$$
\begin{bmatrix} \Delta x \\ \Delta y \\ \Delta z \end{bmatrix} = \begin{bmatrix} x - x_0 \\ y - y_0 \\ z - z_0 \end{bmatrix} \tag{1.2.1}
$$

And given the Rotation matrix and ENU coordinate at reference $(\varphi_0, \lambda_0, h_0)$:

$$
\mathbf{R} = \begin{bmatrix} \sin\varphi_0 & \cos\varphi_0 & 0 \\ \cos\varphi_0 \cdot \sin\lambda_0 & -\sin\varphi_0 \cdot \sin\lambda_0 & \cos\lambda_0 \\ \cos\varphi_0 \cdot \cos\lambda_0 & \sin\varphi_0 \cdot \cos\lambda_0 & \sin\lambda_0 \end{bmatrix} \tag{1.2.2}
$$

Therefore we have the calculation:

$$
\begin{bmatrix} E \\ N \\ U \end{bmatrix} = \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta z \end{bmatrix} \cdot \mathbf{R} \tag{1.2.3}
$$

This is the standard ECEF → ENU transformation used in geodesy and navigation.

1.3 Creating a Dictionary

To organize per-flight data extracted from each GeoJSON file, we build a dictionary where each flight ID maps to three lists:

- coords — longitude, latitude, altitude
- vel — velocity components
- dt — timestamps

The basic structure looks like this:

```
flights = dict({
    "coords": [],
    "vel": [],
    "dt": []
})
```

In practice, we use a defaultdict so each new flight_id automatically initializes this structure.

---

2 Position Prediction

To estimate future aircraft positions, I applied a **physics-based interpolation model** that blends two motion predictors:

1. **Constant-Acceleration (CA) model** — reliable for nearly straight trajectories

2. **Cubic Hermite Spline interpolation** — smooth and accurate for curved motion

The blending weight is determined by the **local curvature** of the trajectory: - Low curvature → motion is nearly straight → CA dominates
- High curvature → motion bends → spline dominates

This adaptive combination produces a more stable and realistic prediction than using either method alone.

2.1 General Prediction

For each flight:

- Convert raw coordinates into a consistent Cartesian frame

- Compute velocity and approximate acceleration

- Estimate local curvature k using

$$k = \frac{\|\mathbf{v} \times \mathbf{a}\|}{\|\mathbf{v}\|^3} \tag{2.1.1}$$

- Compute a flight-specific smoothing parameter

$$\alpha = \frac{\ln 5}{k_{95}} \tag{2.1.2}$$

where k_{95} is the 95th percentile curvature
- For each timestamp, compute: - **Spline prediction** using `CubicHermiteSpline`
- **Constant-acceleration prediction** - Blend them using $w = e^{-\alpha k}$:

$$\hat{p} = w\, p_{\mathrm{CA}} + (1 - w)\, p_{\mathrm{spline}} \tag{2.1.3}$$

This yields a smooth, curvature-aware prediction for each flight.

2.2 Loss Computation

To evaluate the quality of the predicted positions, I compute a **time-normalized Mahalanobis loss** for each flight. This metric captures not only the magnitude of prediction errors but also their **directional structure**, **covariance**, and **temporal spacing**.

The loss is computed in four main steps:

1. Extract Prediction Residuals

For each flight, I compare the predicted positions $\hat{p}_i$ with the actual converted coordinates p_i:

$$r_i = \hat{p}_i - p_i \tag{2.2.1}$$

Only interior points are used `[2 : size-2]` to avoid boundary artifacts from the spline and acceleration models.

The residuals are then centered:

$$\tilde{r}_i = r_i - \bar{r} \tag{2.2.2}$$

This removes global bias and ensures the covariance reflects *shape* rather than offset.

2. Estimate Residual Covariance

The covariance of the centered residuals is computed as:

$$\Sigma = \mathrm{Cov}(\tilde{r}) + \lambda I \tag{2.2.3}$$

A small Tikhonov regularization term $\lambda = 10^{-5}$ stabilizes the inversion of Σ, especially for nearly collinear trajectories.

The inverse covariance Σ^{-1} defines the **Mahalanobis geometry** of the error space.

3. Compute Mahalanobis Distance

For each residual vector:

$$d_i = \sqrt{\tilde{r}_i^\top \Sigma^{-1} \tilde{r}_i} \tag{2.2.4}$$

This distance penalizes errors more strongly along directions where the model is normally precise, and less along directions with naturally higher variance.

4. Normalize by Temporal Spacing

Because timestamps are not uniformly spaced, each error is scaled by a time-dependent factor:

$$t_i = \sqrt{\frac{\Delta t_i}{\bar{\Delta t}}} \tag{2.2.5}$$

The final **time-relative Mahalanobis loss** is:

$$L_i = \frac{d_i}{t_i} \tag{2.2.6}$$

This ensures that predictions made over longer time intervals are not unfairly penalized compared to short-interval predictions.

3 POI Detection

After computing the time-normalized Mahalanobis loss for each flight, the next step is to identify **Points of Interest (POIs)**—locations where the prediction error is unusually high. These points often correspond to sharp maneuvers, abnormal motion, or sensor irregularities, and they serve as valuable markers for downstream analysis.

However, a POI does not always represent an actual infrastructure feature; it simply marks a point where the motion deviates significantly.

The POI detection pipeline consists of three main stages:

3.1 Normalize the Loss Scores

For each flight, the Mahalanobis losses are rescaled to the interval `[0, 1]`:

$$s_i = \frac{L_i - \min(L)}{\max(L) - \min(L) + \varepsilon} \tag{3.1}$$

This normalization ensures that POI detection is **relative to each flight's own dynamics**, making the method robust to differences in scale, speed, or noise across flights.

3.2 Thresholding

Here, I introduced an element called POI score, which indicates how anomalous each point is relative to the rest of the flight.

A point is flagged as a POI if its normalized score exceeds a fixed threshold:

$$s_i \geq 0.75 \tag{3.2}$$

This threshold captures the upper quartile of anomalous behavior while avoiding excessive false positives.

It can be adjusted depending on the desired sensitivity of the detection process.

3.3 Export POIs to CSV

Each detected POI is stored with:

- flight ID
- point index
- longitude, latitude, altitude
- POI score

All POIs are aggregated into a Pandas DataFrame and exported as a CSV file, enabling further visualization, inspection, or integration into downstream workflows.