CP 1 Numerical Programming

Motion Extraction and Derivative-Aware Clustering of Autobahn Traffic

Murman Gorgadze

Instructor: Ramaz Botchorishvili

November 19, 2025

Abstract

This report documents a complete pipeline for extracting motion information from a 10-second traffic video of moving vehicles on a highway (Autobahn) and performing derivative-aware clustering on the tracked objects. The approach follows the CP 1 constraints: no pretrained models are used, all key operations (detection, tracking, numerical differentiation, smoothing, and clustering) employ classical numerical and algorithmic methods. We describe the mathematical foundations behind the Savitzky–Golay (SG) smoothing and differentiation, robust finite-difference fallback, construction of per-object motion features (speed, acceleration, jerk, jounce), path straightness, and derivative-weighted norms for clustering. We also discuss the centroid + Hungarian assignment tracker, background subtraction with morphological cleanup, and (optionally) pixel-to-meter conversion. Experiments include a working case and a failure case to analyze limitations.

1 Introduction

The aim is to extract from video (i) the number of moving vehicles and (ii) their kinematic profiles: speed (first derivative), acceleration (second), jerk (third), and jounce/snap (fourth). These are computed in pixel-related units and, when a spatial scale is available, converted to physical units. The pipeline must also apply clustering "with norms that incorporate derivatives" to reveal groups of vehicles with similar motion behaviors (e.g., steady cruisers vs. abrupt accelerators).

The four main stages are: (1) foreground-based detection (no deep models), (2) multi-object tracking via centroid association and Hungarian assignment, (3) smoothing and differentiating trajectories using Savitzky–Golay filters (with finite-difference fallback), and (4) feature construction and clustering with derivative-aware weights.

2 Video Processing and Detection

2.1 Background Subtraction

Given a frame $I_t \in \mathbb{R}^{H\times W}$, we compute a foreground mask M_t using a running Gaussian mixture background model (OpenCV's MOG2):

$$M_t(x,y) = \begin{cases} 1, & \text{if pixel is foreground at time } t,\ 0, \\ \text{otherwise.} \end{cases} \tag{1}$$

We then threshold to remove shadows and apply morphological opening/dilation to suppress noise and close small gaps. Contours in the cleaned mask are converted into axis-aligned bounding boxes (detections) $\mathcal{D} * t = (x, y, w, h)$ filtered by a minimum area $A * \min$ to reject spurious blobs.

2.2 Detection Function

Listing 1: Detection via background subtraction and morphology.

```
def detect_vehicles(frame, fgbg, min_area=400, dilate_iters=2):
fgmask = fgbg.apply(frame)
_, th = cv2.threshold(fgmask, 200, 255, cv2.THRESH_BINARY) # remove shadows
kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (3,3))
th = cv2.morphologyEx(th, cv2.MORPH_OPEN, kernel, iterations=1)
th = cv2.dilate(th, kernel, iterations=dilate_iters)
cnts, _ = cv2.findContours(th, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
detections = []
for c in cnts:
area = cv2.contourArea(c)
if area < min_area:
continue
x, y, w, h = cv2.boundingRect(c)
```

```
if w*h < min_area: # reject thin noise
continue
detections.append((x, y, w, h))
return detections, th
```

Notes. *(i)* Thresholding at 200 converts gray foreground/shadow intensities into a binary mask. *(ii)* Morphological opening removes isolated pixels; dilation reconnects fragmented blobs. *(iii)*

Multi-Object Tracking

3.1 Centroid Tracking + Hungarian Assignment

Each detection's centroid is $c = (x + \frac{w}{2}, , y + \frac{h}{2})$. We maintain a set of active tracks T_i, where each track stores the history of centroids and sizes. For association, we build a cost matrix $C \in \mathbb{R}^{m \times n}$ between m existing tracks and n current detections using Euclidean distances:

$C_{ij} = \|p_i - d_j\|_2, \quad p_i = \text{last centroid of track } i, \quad d_j = \text{centroid of detection } j. Matches are accepted only if$

$C_{ij} \leq d_{\text{th}}$ (`dist_thresh`); unmatched tracks increase a "missed" counter and are pruned if it exceeds `max_age` frames.

Listing 2: Core data structure for a single tracked object.

```
class Track:
_next_id = 1
def **init**(self, cx, cy, bbox, frame_idx):
self.id = Track._next_id; Track._next_id += 1
self.history = [] # (frame_idx, cx, cy, w, h)
self.add(frame_idx, cx, cy, bbox)
self.missed = 0
def add(self, frame_idx, cx, cy, bbox):
x, y, w, h = bbox
self.history.append((frame_idx, float(cx), float(cy), float(w), float(h)))
def last_centroid(self):
_, cx, cy, _, _ = self.history[-1]
return np.array([cx, cy])
```

Listing 3: Centroid Tracker with Hungarian association and gating.

```
from scipy.spatial.distance import cdist
from scipy.optimize import linear_sum_assignment

class CentroidTracker:
def **init**(self, max_age=15, dist_thresh=80.0):
self.max_age = max_age
self.dist_thresh = dist_thresh
self.tracks = []
def update(self, detections, frame_idx):
det_centroids = np.array([[x+w/2, y+h/2] for (x,y,w,h) in detections], dtype=np.float32)
track_centroids = np.array([t.last_centroid() for t in self.tracks], dtype=np.float32)
if self.tracks else np.zeros((0,2), np.float32)
if len(self.tracks) == 0:
for bbox in detections:
cx, cy = bbox[0]+bbox[2]/2, bbox[1]+bbox[3]/2
self.tracks.append(Track(cx, cy, bbox, frame_idx))
return
if len(detections) == 0:
```

```
for t in self.tracks: t.missed += 1
else:
C = cdist(track_centroids, det_centroids) if len(track_centroids) and len(det_centroids)
    ↪ else np.empty((0,0))
assigned_tracks, assigned_dets = set(), set()
if C.size:
r_idx, c_idx = linear_sum_assignment(C)
for r, c in zip(r_idx, c_idx):
if C[r, c] <= self.dist_thresh:
t = self.tracks[r]
x, y, w, h = detections[c]
cx, cy = x+w/2, y+h/2
t.add(frame_idx, cx, cy, (x,y,w,h))
t.missed = 0
assigned_tracks.add(r); assigned_dets.add(c)
for i, t in enumerate(self.tracks):
if i not in assigned_tracks: t.missed += 1
for j, bbox in enumerate(detections):
if j not in assigned_dets:
cx, cy = bbox[0]+bbox[2]/2, bbox[1]+bbox[3]/2
self.tracks.append(Track(cx, cy, bbox, frame_idx))
self.tracks = [t for t in self.tracks if t.missed <= self.max_age]
```

Mathematical Objective. Hungarian assignment solves $\min_\pi \sum_i C_{i,\pi(i)}$, where π is a permutation (or partial matching) between tracks and detections. The gating condition $C_{ij} \le d_{\text{th}}$ enforces a physical prior that objects cannot teleport across frames.

4 Trajectory Smoothing and Numerical Derivatives

Let $x(t)$ and $y(t)$ be pixel coordinates over time with sampling period $\Delta t = 1/\text{fps}$. Direct finite differences amplify tracking noise, especially for higher derivatives. We therefore use Savitzky–Golay filtering to both smooth and differentiate.

4.1 Savitzky–Golay (SG) Filter

SG filtering fits a local polynomial of order p over a symmetric window of w samples, minimizing least-squares error, and evaluates the polynomial (or its derivatives) at the window center. For samples $f_{t-k}, \dots, f_t, \dots, f_{t+k}$, we solve

$$\min_{a_0,\dots,a_p} \sum_{i=-k}^{k} \left[f_{t+i} - P(i)\right]^2, \quad P(i) = a_0 + a_1 i + a_2 i^2 + \cdots + a_p i^p. \tag{3}$$

The r-th derivative estimate at the center is then given by a fixed convolution of the window samples with precomputed SG coefficients; with sample period Δt this yields correct units via delta $= \Delta t$.

4.2 Implemented Derivative Chain

We compute components

$$\mathbf{v}(t) = \left[\dot{x}(t)\ \dot{y}(t)\right], \quad \mathbf{a}(t) = \left[\ddot{x}(t)\ \ddot{y}(t)\right], \quad \mathbf{j}(t) = \left[x^{(3)}(t)\ y^{(3)}(t)\right], \quad \mathbf{s}(t) = \left[x^{(4)}(t)\ y^{(4)}(t)\right], \tag{4}$$

using SG with window $w \approx \text{fps} \cdot T_w$ (here $T_w = 0.5$ s) and polynomial order $p = 3$. For short tracks ($n < 7$) we fall back to central gradients:

$$\dot{x}(t_i) \approx \frac{x_{i+1} - x_{i-1}}{2\Delta t}, \quad \ddot{x}(t_i) \approx \frac{x_{i+1} - 2x_i + x_{i-1}}{\Delta t^2}, \text{ etc.} \tag{5}$$

Listing 4: Savitzky–Golay differentiation with finite-difference fallback.

```
from scipy.signal import savgol_filter

def compute_derivatives(times, xs, ys, fps, sg_window_sec=0.5, polyorder=3):
dt = 1.0 / fps
n = len(times)
if n < 7:
vx = np.gradient(xs, dt); vy = np.gradient(ys, dt)
ax = np.gradient(vx, dt); ay = np.gradient(vy, dt)
jx = np.gradient(ax, dt); jy = np.gradient(ay, dt)
sx = np.gradient(jx, dt); sy = np.gradient(jy, dt)
return vx, vy, ax, ay, jx, jy, sx, sy
w = max(5, int(round(sg_window_sec * fps)))
if w % 2 == 0: w += 1
w = min(w, n - (1 - n % 2))
if w < 5: w = 5
vx = savgol_filter(xs, w, polyorder, deriv=1, delta=dt)
vy = savgol_filter(ys, w, polyorder, deriv=1, delta=dt)
ax = savgol_filter(xs, w, polyorder, deriv=2, delta=dt)
ay = savgol_filter(ys, w, polyorder, deriv=2, delta=dt)
jx = savgol_filter(xs, w, polyorder, deriv=3, delta=dt)
jy = savgol_filter(ys, w, polyorder, deriv=3, delta=dt)
sx = savgol_filter(xs, w, polyorder, deriv=4, delta=dt)
sy = savgol_filter(ys, w, polyorder, deriv=4, delta=dt)
return vx, vy, ax, ay, jx, jy, sx, sy
```

Units. If a pixel-to-meter scale κ is known (e.g., from lane width), we convert $x \mapsto \kappa x$ and derivatives scale as $\dot{x} \mapsto \kappa\dot{x}$, $\ddot{x} \mapsto \kappa\ddot{x}$, etc., preserving physical units (m/s, m/s^2, ...).

5 Feature Engineering for Clustering

For each track we compute scalar magnitudes $v = \|\mathbf{v}\|_2$, $a = \|\mathbf{a}\|_2$, $j = \|\mathbf{j}\|_2$, $s = \|\mathbf{s}\|_2$ framewise, then summarize with means and standard deviations. We also compute a geometric straightness measure

$$\text{straightness} = \frac{\|(x_{\text{end}} - x_{\text{start}},\ y_{\text{end}} - y_{\text{start}})\|_2}{\sum_i \|(x_{i+1} - x_i,\ y_{i+1} - y_i)\|_2 + \varepsilon}, \quad 0 \le \text{straightness} \le 1. \tag{6}$$

Duration T and frame count N are also included.

Listing 5: Per-track feature summarization.

```
def summarize_features(df_one):
v = np.hypot(df_one.vx, df_one.vy)
a = np.hypot(df_one.ax, df_one.ay)
j = np.hypot(df_one.jx, df_one.jy)
s = np.hypot(df_one.sx, df_one.sy)
dx = df_one.x.iloc[-1] - df_one.x.iloc[0]
dy = df_one.y.iloc[-1] - df_one.y.iloc[0]
disp = math.hypot(dx, dy)
path = np.sum(np.hypot(np.diff(df_one.x), np.diff(df_one.y))) + 1e-9
straightness = disp / path
feats = {
"v_mean": float(np.mean(v)), "v_std": float(np.std(v)),
"a_mean": float(np.mean(a)), "a_std": float(np.std(a)),
"j_mean": float(np.mean(j)), "j_std": float(np.std(j)),
"s_mean": float(np.mean(s)), "s_std": float(np.std(s)),
"straightness": float(straightness),
"duration": float(df_one.time.iloc[-1] - df_one.time.iloc[0]),
```

```
"n_frames": int(len(df_one)),
}
return feats
```

5.1 Derivative-Aware Feature Vectors

To incorporate derivatives into the clustering norm, we scale feature blocks with weights (w_v, w_a, w_j, w_s) prior to distance computations. For track i we form a vector

$$\mathbf{z}_i = [w_v\mu(v),\ w_v\sigma(v),\ w_a\mu(a),\ w_a\sigma(a),\ w_j\mu(j),\ w_j\sigma(j),\ w_s\mu(s),\ w_s\sigma(s),\ \text{straightness},\ T], \tag{7}$$

where $\mu(\cdot)$ and $\sigma(\cdot)$ are mean and standard deviation. The weight tuple adjusts the relative importance of velocity vs. higher derivatives in the similarity metric.

Listing 6: Constructing the derivative-weighted feature matrix.

```
def build_feature_matrix(tracks_df, use_weights=(1.0, 1.0, 1.0, 1.0)):
rows, ids, meta = [], [], []
for tid, grp in tracks_df.groupby("track_id"):
feats = summarize_features(grp)
wv, wa, wj, ws = use_weights
vec = [wv*feats["v_mean"], wv*feats["v_std"],
wa*feats["a_mean"], wa*feats["a_std"],
wj*feats["j_mean"], wj*feats["j_std"],
ws*feats["s_mean"], ws*feats["s_std"],
feats["straightness"], feats["duration"]]
rows.append(vec); ids.append(tid); meta.append(feats)
X = np.array(rows, dtype=np.float32) if rows else np.zeros((0,10), np.float32)
return ids, X, pd.DataFrame(meta, index=ids)
```

6 Clustering Methods

We consider K-Means and DBSCAN as complementary unsupervised algorithms.

6.1 K-Means

Given feature matrix $X \in \mathbb{R}^{n\times d}$ and cluster count k, K-Means minimizes the within-cluster sum of squares:

$$\min_{\mathcal{C}*r*r=1^k} \sum_{r=1}^{k} \sum_{\mathbf{z}_i\in\mathcal{C}_r} \|\mathbf{z}_i - \boldsymbol{\mu}_r\|_2^2, \quad \boldsymbol{\mu}_r = \frac{1}{|\mathcal{C}*r|} \sum *\mathbf{z}_i \in \mathcal{C}_r\mathbf{z}_i. \tag{8}$$

Here the derivative weights are already embedded in $\mathbf{z}_i$. We select k by testing small integers (e.g., $k \in 2, 3, 4, 5$) and choosing an elbow in inertia or the minimal inertia under a small range.

6.2 DBSCAN

DBSCAN groups points by density using parameters $(\varepsilon, , \text{minPts})$. A point is a core if at least minPts neighbors lie within distance ε; clusters form by connecting core-reachable points, and outliers get label -1. DBSCAN does not require pre-specifying the number of clusters and can reveal irregular groups (e.g., unusual driving behavior).

7 Pixel-to-Meter Conversion (Optional)

If a spatial calibration $\kappa = \text{m/px}$ is known (e.g., via lane width 3.5 m spanning L pixels), we scale positions and all derivatives by κ to express kinematics in SI units. Given frame rate f (fps), the sampling period is $\Delta t = 1/f$.

8 Algorithmic Summary

Algorithm 1 Autobahn Motion Extraction and Clustering

Input: video I_t, fps f, weights (w_v, w_a, w_j, w_s)
Initialize background model and tracker
for each frame t **do**
    $(\mathcal{D}_t, , M_t) \leftarrow$ DetectVehicles(I_t) ▷ Sec. 2
    Update tracker with $\mathcal{D}_t$ ▷ Sec. 3
end for
Build trajectories $(x_i(t), y_i(t))$ from tracks
for each track i **do**
    Smooth / differentiate (x_i, y_i) via SG to get $\mathbf{v}, \mathbf{a}, \mathbf{j}, \mathbf{s}$ ▷ Sec. 4
    Form per-track features and apply derivative weights ▷ Sec. 5
end for
Cluster feature vectors with K-Means and DBSCAN ▷ Sec. 6
Analyze results, choose success and failure videos, discuss limitations

9 Limitations and Failure Cases

Foreground segmentation can be disrupted by camera motion, strong shadows, rain, and headlight flicker. Centroid association may swap IDs in dense traffic or at crossings when detections overlap. SG derivatives near track endpoints are less reliable, and higher-order derivatives are inherently noise-sensitive. Clustering depends on feature scaling; extreme class imbalance may cause small clusters to be absorbed by larger ones.

10 Observed Failure Case: One Real Car, Many Spurious Tracks

In the supplied annotated video, the pipeline sometimes produces *multiple* tracked vehicles for what is clearly a *single* physical car. Instead of following one stable bounding box and track ID, the algorithm repeatedly creates and destroys short tracks around the same object. This behaviour is undesirable from an application point of view, but it is a very instructive failure mode of the chosen "classical" approach.

10.1 Where the Error Comes From

The problem arises from the interaction of three components:

(a) **Noisy foreground mask and over–segmentation.** The MOG2 background model plus thresholding and morphology produces a binary mask $M_t(x, y)$. Small changes in illumination, compression artefacts, or shadows can cause the foreground blob corresponding to a single car to be fragmented into two or more connected components in some frames. Each connected component becomes a separate detection $(x, y, w, h) \in \mathcal{D}_t$.

(b) **Distance-only Hungarian association.** In the tracking step we build a cost matrix

$$C_{ij} = \|p_i - d_j\|_2,$$

where p_i is the last centroid of track i and d_j is the centroid of detection j. The Hungarian algorithm finds a minimal assignment, but only under a distance gate $C_{ij} \leq d_{\text{th}}$. If a car is temporarily over–segmented or its centroid jumps because the blob shape changes, the distance may exceed d_{th}; the old track is then marked as "missed" and a *new* track is spawned for the same physical object.

(c) **Aggressive track aging (`max_age`).** The tracker increments a `missed` counter whenever a track cannot be matched in the current frame. Once `missed` > `max_age`, the track is deleted. If segmentation is unstable, the same car can repeatedly cross the threshold and come back, leading to several short, disconnected track IDs around one vehicle.

Altogether, small variations in M_t cause jitter and fragmentation in $\mathcal{D}_t$, which in turn leads to inconsistent matching in the cost matrix C and finally to multiple short tracks for a single car.

10.2 Possible Improvements and Remedies

Within the "no pretrained model" constraint, several classical modifications can reduce this failure:

(1) More robust foreground processing.

- **Tune morphology and area thresholds.** Increasing `min_area` and using slightly larger structuring elements or an extra dilation/closing step can merge fragmented blobs belonging to the same car, so that only one bounding box is produced per vehicle.
- **Region-of-interest (ROI) masking.** Restricting detection to the actual road area (and masking out static background like sky, barriers, trees) reduces spurious detections that appear in irrelevant parts of the frame.

(2) Tracking-side consistency checks.

- **Adaptive distance thresholds.** Instead of a fixed d_{th}, one can impose a motion model: a car cannot move more than $v_{\max}\,\Delta t$ pixels per frame. This leads to a data-driven gate on C_{ij} based on the last estimated velocity.
- **Track merging / post-processing.** After the video is processed, two short tracks T_a and T_b can be merged if the end of T_a and the start of T_b are close in both time and space and their velocities are compatible. Formally, if

$$\left\|p_a^{\text{end}} - p_b^{\text{start}}\right\|_2 \le \epsilon_{\text{space}}, \quad |t_b^{\text{start}} - t_a^{\text{end}}| \le \epsilon_{\text{time}},$$

  and the velocity vectors are similar, then T_a and T_b likely belong to the same physical car and can be fused into a single trajectory.

(3) Adding simple appearance or size cues. At the moment, the cost matrix C uses only centroid distance. It is possible to extend C_{ij} to penalize abrupt changes in bounding box size or aspect ratio, e.g.

$$C_{ij}^{\text{aug}} = \alpha\,\|p_i - d_j\|_2 \;+\; \beta\,|w_i - w_j| \;+\; \gamma\,|h_i - h_j|,$$

with weights $\alpha, \beta, \gamma > 0$. This discourages associations where the box suddenly shrinks or grows, which often happens for ghost detections.

(4) Kalman-filter-based tracking. A classical extension (still fully allowed in a numerical programming context) is to replace pure centroid association with a Kalman filter per track. The filter predicts the next centroid based on a constant-velocity (or constant-acceleration) model, and the Hungarian step matches detections to *predictions* rather than to raw last positions. This makes the tracker more robust when detections are temporarily missing or noisy.

In the accompanying "failure" video, we intentionally *do not* over-tune these hyperparameters, to highlight how sensitive classical background subtraction + centroid tracking can be to segmentation artefacts. The discussion above points to straightforward numerical and algorithmic modifications that could significantly reduce the number of spurious cars, at the cost of slightly higher complexity and more hyperparameter tuning.

11 Conclusion

We presented a fully classical, reproducible pipeline for extracting vehicle motion profiles and grouping behaviors on highway video. The mathematical core—local polynomial smoothing and differentiation with Savitzky–Golay, finite-difference fallback, and derivative-weighted feature norms—enables stable estimation of speed, acceleration, jerk, and jounce without machine learning models. Clustering on these features reveals interpretable motion patterns (e.g., steady vs. erratic drivers). Future work may include perspective correction, Kalman filtering, and learned appearance cues while preserving the no-pretrained constraint.