SCIA Promotion 2026

# MLVOT TPs and Project Report

Friday, December 5th

## I. Single Object Tracking with Kalman (Centroid-Tracker)

### I.1. Objective

We implemented a single object tracking solution using a pre-existing object detection algorithm from OpenCV. To smooth the tracking process and improve accuracy, we integrated a Kalman Filter as specified in the assignment.

### I.2. Object Detection

Object detection is performed via a combination of the `Canny` function (to detect edges) and the `findContours` function from the OpenCV library (`cv2`). The process follows these steps:

1. Preprocessing: The frame is converted from BGR to Grayscale.
2. Edge Detection: We apply Canny edge detection, followed by a threshold to convert the image into a binary (black and white) format.
3. Contour Extraction: We use `findContours` to identify object shapes.

For each contour, we compute the center and radius of the minimum enclosing circle using `minEnclosingCircle`. To filter out noise, we only accept objects with a radius between 3 and 30 pixels. The detected circles are drawn in green on the output frames. As this is a Single Object Tracking (SOT) task, we currently only process the first center detected in the list.

### I.3. Kalman Filter

We implemented the Kalman Filter using a Python class (`KalmanFilter.py`). The class includes a constructor (`__init__`) where we initialize the system matrices ($A, B, H, Q, R, P$) and parameters based on the sampling time ($\Delta t$) and standard deviations provided in the subject.

The class defines two key methods:

- `predict()`: This function computes the prediction of the state estimate $\hat{x}_k^-$ and the error covariance $P_k^-$.
- `update()`: This function accepts the measurement $z_k$ (the coordinates of the centroid found during detection) to correct the predicted state.

Since we are performing single object tracking, a single instance of the `KalmanFilter` class is instantiated and shared across frames in the main program.

### I.4. Visualization & Results

To visualize the tracking results, we display the following on the output video:

- Predicted Position (Blue): A rectangle representing the Kalman filter's prediction step (a priori).
- Estimated Position (Red): A rectangle representing the corrected position after the Kalman update step (posteriori).

- Trajectory (Cyan): To track the object's path, we store the corrected centroids in a list and draw lines connecting each consecutive point.

*Note: While the detector calculates a specific radius for filtering, the visualization uses a fixed radius (set to 15 pixels) for the blue and red tracking indicators.*

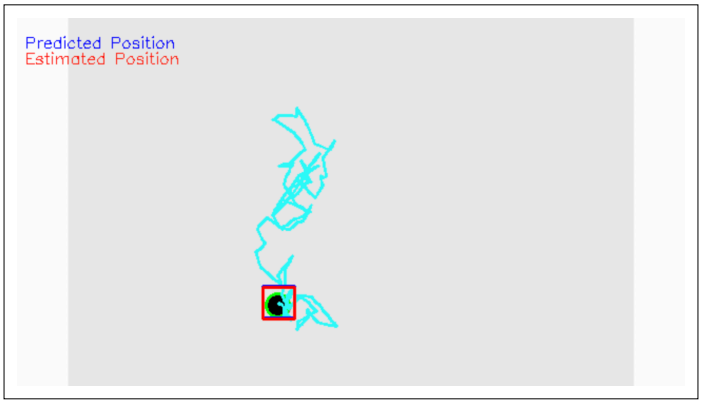


Figure 1: Tracker output frame

# II. IOU-based Tracking (Bounding-Box Tracker)

## II.1. Objective

In this second part, we develop a Multiple Object Tracker (MOT) based on the Intersection over Union (IoU) metric. Unlike the previous single object tracker, we now handle multiple targets simultaneously using bounding boxes and a data association algorithm to link detections across frames.

## II.2. Data Management

We utilize the `ADL-Rundle-6` sequence. Instead of performing detection in real-time, we load pre-generated detections from a text file (`det.txt`) generated by `Yolov5s`. We implemented a parser in the `load_detections` function that reads the MOTChallenge format (frame, id, left, top, width, height, conf, x, y, z) and stores `Detection` objects in a dictionary indexed by frame number.

Each `Detection` contains a `BoundingBox` object which pre-calculates coordinates (top-left, bottom-right) and area to facilitate IoU computation.

## II.3. Tracking Algorithm

The core tracking logic is implemented in the `assign_tracks` function. It iterates through consecutive frames to associate existing tracks with new detections.

### II.3.a. Similarity Matrix (IoU)

For every pair of tracked object $i$ in frame $t$ and new detection $j$ in frame $t+1$, we compute the Intersection over Union (IoU). The IoU is defined as:

$$\text{IoU} = \frac{\text{Area of Intersection}}{\text{Area of Union}}$$

We construct a cost matrix where the cost is defined as $1 - \text{IoU}$. This converts the problem into a minimization task suitable for standard assignment algorithms.

### II.3.b. Data Association (Hungarian Algorithm)

To solve the assignment problem, we apply the Hungarian Algorithm (via `scipy.optimize.linear_sum_assignment`) on the cost matrix. This function finds the optimal pairing of rows (tracks) and columns (detections) that minimizes the total cost (effectively maximizing the total IoU).

### II.3.c. Track Management

Based on the indices returned by the Hungarian algorithm, we categorize the results using `find_unmatched`:

- Matched Tracks: The detection in frame $t + 1$ inherits the `id` of the matched track from frame $t$.
- Unmatched Detections: These represent new objects entering the scene. We assign them a new unique `id` and increment the global counter.
- Unmatched Tracks: These represent objects that were tracked but not detected in the current frame (e.g., occlusion or exit). *(Note: While identified in the code, specific deletion logic for lost tracks is currently marked as a future implementation step).*

## II.4. Visualization & Results

We implemented `process_and_display_tracks` to visualize the results. For each frame, we draw the bounding box in green and display the assigned Track ID above it. The sequence is saved as an MP4 video (`output.mp4`) using the `avc1` codec for efficiency.

Finally, the full tracking results (with updated IDs) are exported to a text file following the standard ground truth format for evaluation.

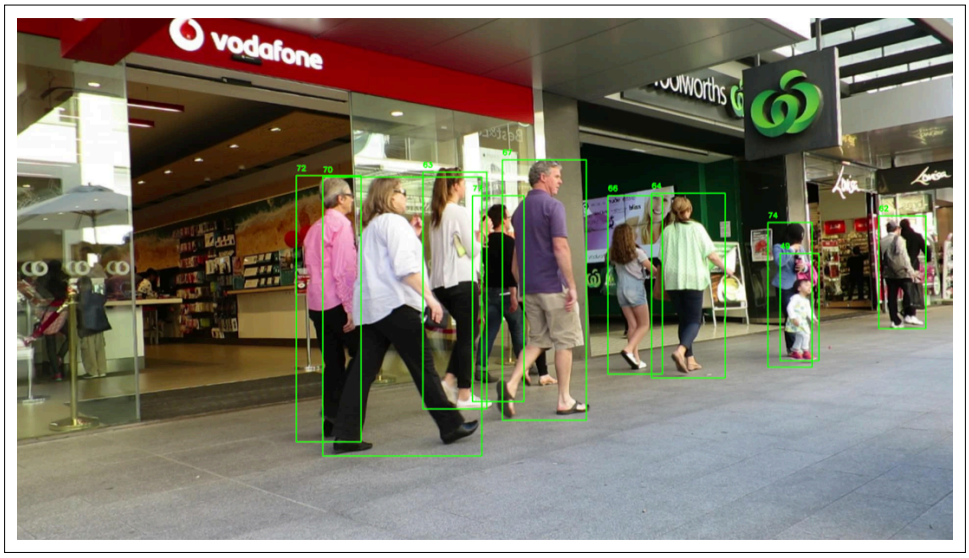


Figure 2: Tracker output frame

# III. TP 3: Kalman-Guided IoU Tracking (Bounding-Box Tracker)

## III.1. Objective

The objective of this third practical work is to extend the previous IoU-based Multiple Object Tracker (MOT) by integrating a Kalman Filter. This addition allows us to predict the future position of objects, enabling more robust data association and smoother tracking trajectories compared to purely geometric IoU matching.

## III.2. Kalman Filter Integration

We integrate the `KalmanFilter` class developed in TP1 into our tracking architecture. To manage this, we introduced a `Track` class which acts as a wrapper for each tracked object.

- State Representation: The Kalman Filter estimates the state of the object's centroid $(x, y, v_x, v_y)$.
- Dimensions: Since the standard Kalman Filter implementation here tracks only the centroid, the width and height of the bounding box are stored separately within the `Track` object and carried forward during the prediction steps.

## III.3. Algorithm Overview

The tracking loop in `assign_tracks` was modified to include the prediction/update cycle of the Kalman Filter. The workflow is as follows:

### III.3.a. Prediction Step

Before performing any data association, we iterate through all `active_tracks`. For each track, we call the `predict()` method.

- The Kalman Filter projects the state vector forward to time $t$.
- We reconstruct a Predicted Bounding Box using the new estimated centroid coordinates $(\hat{x}, \hat{y})$ and the object's existing dimensions.

### III.3.b. Data Association (Predicted vs. Detected)

The cost matrix is now computed using the IoU between the Predicted Bounding Box (from step 1) and the New Detections (from the detector).

- This is a significant improvement over TP2: we are now matching where we *expect* the object to be, rather than where it was in the previous frame.
- The Hungarian Algorithm is applied to this new cost matrix to find optimal assignments.

### III.3.c. Update Step

For every matched pair (Track $i$, Detection $j$):

- We call the `update()` method on the track.
- The Kalman Filter corrects its state using the centroid of the matched detection as the measurement $z_k$.
- The track's dimensions (width/height) are updated to match the new detection.

## III.4. Track Management

The track lifecycle is managed based on the assignment results:

- New Tracks: Unmatched detections create new `Track` instances, initializing the Kalman Filter state with the detected centroid.

- Matched Tracks: Persist to the next frame with updated states.
- Lost Tracks: In this implementation, any track that is not matched to a detection in the current frame is immediately removed from `active_tracks`.

### III.5. Visualization

The visualization remains similar to TP2, displaying bounding boxes and IDs. However, the underlying positions are now smoothed by the Kalman filter, providing more stable tracking, especially when detections are noisy.

## IV. Appearance-Aware IoU-Kalman Object Tracker

### IV.1. Objective

The final practical work aims to enhance the Multi-Object Tracker (MOT) by incorporating Object Re-Identification (ReID). By combining geometric information (IoU) with visual appearance features, the tracker becomes robust to occlusions and large displacements where IoU alone might fail.

### IV.2. Feature Extraction (ReID)

To capture the visual appearance of objects, we implemented a `FeatureExtractor` class using a Convolutional Neural Network (CNN).

- Model: We utilized `MobileNetV2` pretrained on ImageNet as a lightweight feature extractor. We removed the classifier head to obtain a raw 1280-dimensional feature vector.
- Preprocessing: Each detected bounding box is cropped from the frame and processed as follows:
  1. Resized to $(64, 128)$ pixels.
  2. Converted from BGR to RGB.
  3. Normalized using standard ImageNet mean and standard deviation.
  4. Transposed to channel-first format $(C, H, W)$ for PyTorch compatibility.

### IV.3. Appearance-Aware Data Association

The data association step now fuses two metrics to compute the cost matrix:

1. Geometric Cost: The standard Intersection over Union (IoU) between predicted tracks and new detections.
2. Appearance Cost: The Cosine Similarity between the track's stored feature vector and the new detection's feature vector.

#### IV.3.a. Combined Score Calculation

The final similarity score $S$ is a weighted sum of both metrics:

$$S = \alpha \cdot \text{IoU} + \beta \cdot \text{Normalized Similarity}$$

Where:

- $\alpha = 0.5$ and $\beta = 0.5$ are weights balancing geometry and appearance.
- The similarity matrix is normalized to the $[0, 1]$ range to match the scale of the IoU.

The Hungarian Algorithm minimizes the cost $1 - S$ to find the optimal assignment.

### IV.4. Feature Update Strategy

To handle changes in appearance (e.g., rotation, lighting), we update the feature vector of each matched track using an Exponential Moving Average (EMA):

$$F_t = \gamma \cdot F_{t-1} + (1 - \gamma) \cdot F_{\text{new}}$$

With $\gamma = 0.9$, this ensures the track retains a stable history of its appearance while gradually adapting to new visual information.

### IV.5. Results

The integration of ReID significantly improves tracking consistency. Identity switches are reduced, particularly in crowded scenes where bounding boxes overlap or when objects temporarily leave and re-enter the frame.