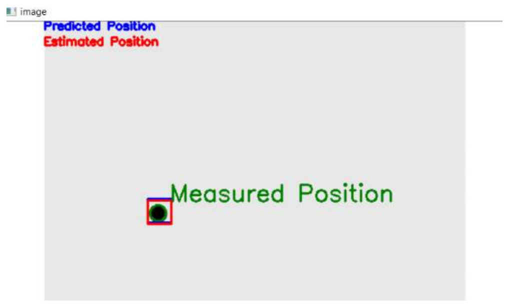
TP 1: Single Object Tracking with Kalman (Centroid-Tracker): 10 points

Objective: Implement object tracking in a 2D (two-dimensional) using pre-existing object detection algorithm and integrate the Kalman Filter for smooth and accurate tracking.

- Object representation: point (centroid)
- SOT (Single Object Tracker)
- Single Object Tracking



Provided Materials:

- Data: A video containing sequence of frames with one object to be tracked
- Object Detection Code: Detector.py contains one function for detecting multi-objects in each frame using Canny edge detection. It returns the centers of detected objects.

Python implementation:

The project will have two additional files: KalmanFilter.py and objTracking.py

1. Kalman Filter Implementation
   1.1 Create KalmanFilter.py. This file must contain one class called KalmanFilter consisting of three functions: __init__(), predict(), update()
   1.2 **Function Initialization**: __init()__ . The class will be initialized with six parameters:
      - *dt* : time for one cycle used to estimate state (sampling time)
      - *u_x, u_y* : accelerations in the x-, and y-directions respectively
      - *std_acc*: process noise magnitude
      - *x_sdt_meas, y_sdt_meas* :standard deviations of the measurement in the x- and y-directions respectively
   - Define:
      - Control input variables *u=[u_x, u_y]*
      - Initial state matrix: $(\widehat{x_k})=[x_0=0, y_0=0, v_x=0, v_y=0]$
      - Matrices describing the system model *A*,*B* with respect to the sampling time dt (Δt) :

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & \Delta t & 0 \\ 0 & 1 & 0 & \Delta t \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (6)$$

$$\mathbf{B} = \begin{bmatrix} \frac{1}{2}(\Delta t)^2 & 0 \\ 0 & \frac{1}{2}(\Delta t)^2 \\ \Delta t & 0 \\ 0 & \Delta t \end{bmatrix} \quad (7)$$

      - Measurement mapping matrix *H*

$$\mathbf{H} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \quad (10)$$

- Initial process noise covariance matrix *Q* with respect to the standard deviation of acceleration (*std_acc*) σ_a :

$$\mathbf{Q} = \begin{bmatrix} \frac{\Delta t^4}{4} & 0 & \frac{\Delta t^3}{2} & 0 \\ 0 & \frac{\Delta t^4}{4} & 0 & \frac{\Delta t^3}{2} \\ \frac{\Delta t^3}{2} & 0 & \Delta t^2 & 0 \\ 0 & \frac{\Delta t^3}{2} & 0 & \Delta t^2 \end{bmatrix} \sigma_a^2 \qquad (12)$$

- Initial measurement noise covariance *R.* Suppose that the measurements z (*x, y)* are both independent (so that covariance x and y is 0), and look only the

$$\mathbf{R} = \begin{array}{c c} & \begin{array}{cc} x & y \end{array} \\ \begin{array}{c} x \\ y \end{array} & \begin{bmatrix} \sigma_x^2 & 0 \\ 0 & \sigma_y^2 \end{bmatrix} \end{array} \qquad (13)$$

  variance in the x and y: *x_sdt_meas* =σ_x^2 , *y_sdt_meas*=σ_y^2

➢ Initialize covariance matrix *P* for prediction error as an identity matrix whose shape is the same as the shape of the matrix *A*.

1.3 **Function predict()**

This function does the prediction of the state estimate $\hat{x}_k^-$ and the error prediction P_k^-. This task also call the time update process (*u*) because it projects forward the current state to the next time step.

➢ Update time state

$$\hat{\mathbf{x}}_k^- = A\hat{\mathbf{x}}_{k-1} + B\mathbf{u}_{k-1}$$

➢ Calculate error covariance

$$\mathbf{P}_k^- = A\mathbf{P}_{k-1}A^T + \mathbf{Q}$$

1.4 **Function: update ()**

This, function takes measurements z_k as input (centroid coordinates x,y of detected circles)

➢ Compute Kalman gain

$$S_k = HP_k^- H^T + R$$
$$K_k = P_k^- H^T {S_k}^{-1}$$

➢ Update the predicted state estimate $\widehat{x_k}$ and predicted error covariance P_k

$$\hat{\mathbf{x}}_k = \hat{\mathbf{x}}_k^- + \mathbf{K}_k(\mathbf{z}_k - H\hat{\mathbf{x}}_k^-)$$

$$\mathbf{P}_k = (I - \mathbf{K}_k H)\mathbf{P}_k^-$$

2. Create the main file of this project that will be execute to track an object (objTracking.py)

2.1 Import function detect() and KalmanFilter

2.2 Create the object of the class Kalman filter and set parameters values as:

- *dt=0.1, u_x=1, u_y=1, std_acc=1, x_dt_meas=0.1, y_dt_meas=0.1*

You can also try to set other values and observe the performance.

2.3 Create video capture object

2.4 Object Detection Integration. Use provided object detection code to detect black cercle in each frame.

2.5 If centroid is detected then track it. Call the Kalman prediction function and the Kalman filter updating function.

2.6 Visualize for tracking results:

➢ Draw detected circle (green color)

- Draw a blue rectangle as the predicted object position
- Draw a red rectangle as the estimated object position
- Draw the trajectory (tracking path) in an image

TP 2: IOU-based Tracking (Bounding-Box Tracker): 15 points

Objective: Develop a Simple IoU-Based Tracker and extend it for Multiple Object Tracking

- Object representation: bounding box
- MOT (Multiple Object Tracker)
- Data: pre-generated detections loaded from text file

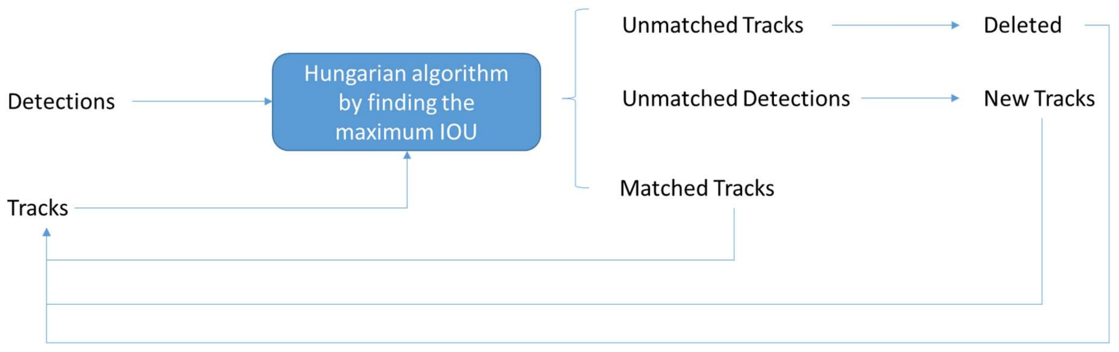
Data:

- ADL-Rundle-6 sequence (img1 folder). All images are in JPEG and named sequentially to a 6-digit file name (e.g. 000001.jpg)
- Detections (det folder) and ground truth annotations (gt folder) are simple comma-separated value (CSV) files. For the detections, three different folders are available: public dataset (file download from MOT Challenge site), Yolov5s (using small model), Yolov5l (using large model). Choose the data that suits you best but be aware that the detection quality varies. The files are named the same way as their corresponding images.

1. Load detections (det) stored in a MOT-challenge like formatted text file. Each line represents one object instance and contains 10 values (fieldNames = [<frame>, <id>, <bb_left>, <bb_top>, <bb_width>, <bb_height>, <conf>, <x>, <y>, <z>]
   - frame = frame number
   - id = number identifies that object as belonging to a trajectory by assigning a unique ID (set to −1 in a detection file, as no ID is assigned yet).
   - bb_left, bb_top, bb_width, bb_height: bounding box position in 2D image coordinates *i.e.* the top-left corner as well as width and height
   - conf: detection confidence score
   - x,y,z: the world coordinates are ignored for the 2D challenge and can be filled with -1.
2. Create similarity matrix:

   2.1 Initialization
      - Define the lists or arrays to store the current tracked bounding boxes and the new detections for the current frame.

   2.2 Calculate IoU for all pairs
      - Create a similarity matrix (a 2D array) where each entry (i,j) corresponds to the IoU value between the i^{th} tracked object and the j^{th} new detection
      - The dimensions of this matrix will be (N×M), where N is the number of tracked objects and M is the number of new detections. Compute similarity score using the Jaccard index (intersection-over-union) for each pair of bounding boxes
3. Associate the detections to tracks
   - Apply the Hungarian algorithm using existing libraries (e.g. function linear_sum_assignement from scipy library for Python) to find the optimal assignment of detections to tracked objects
4. Implement track management
   - Each object can be assigned to only one trajectory (ID)
   - Create and update lists for matches, unmatched detections and unmatched tracks
      - ✓Matched -> update existing tracks based on associations
      - ✓Unmatched tracks -> remove tracks that exceed the maximum missed frames
      - ✓Unmatched detections -> create new tracks



5. Develop an interface for tracking results check to see if the tracker properly keeps track of objects:
   - Display Video Frames
   - Draw Bounding Boxes: Overlay bounding boxes for each tracked object on the frames
   - Show Track IDs: Label the bounding boxes with track IDs for identification
   - Save tracking video

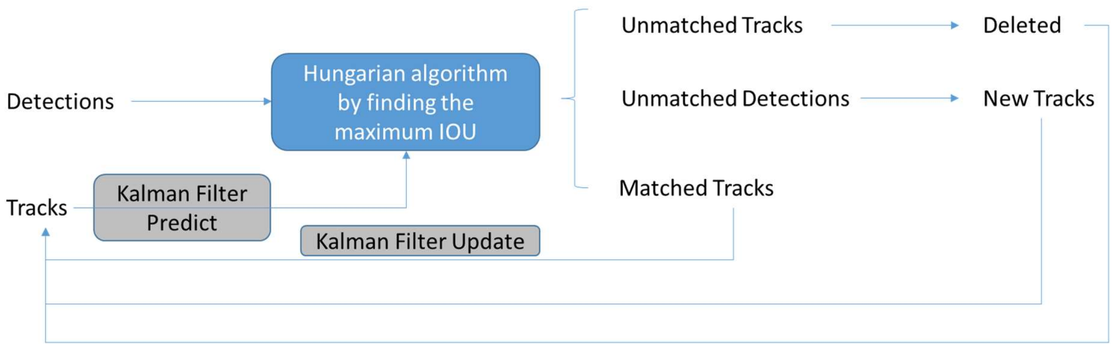
6. Save tracking results in a txt file.

The file name must be exactly like the sequence name. The file format should be the same as the ground truth file (gt.txt), which is a CSV text-file containing one-object instance per line. Each line must contain 10 values. Update the id column (2th value) with the unique ID assigned to the track. The 7th value (*conf*) act as a flag 1.

TP 3: Kalman-Guided IoU Tracking (Bounding-Box Tracker): 20 points

<u>**Objective**</u>**:** Extend IoU-based MOT with Hungarian Algorithm by adding Kalman Filter

1. Integrate Kalman Filter (from Exercise 1) into the tracking System (from Exercise 2)
   - Modify the tracking algorithm according to the diagram below. Practical tip: represent the bounding boxes by their centroids to be able to apply the Kalman filter
   - Integrate the IoU calculation with the Kalman Filter's prediction step. The association process will now first predict the state of each track



TP 4: Appearance-Aware IoU-Kalman Object Tracker :25 points

Objective: Extend IoU-Kalman tracker to include object re-identification (ReID)

Instead of relying solely on geometric information (IoU) consider also the appearance similarity between the predicted object state (form frame t-1) and the detected objects (from frame t).

1. Implement Object Re-Identification
   - Feature Extraction**:** Use a pre-trained **lightweight** deep learning model (EfficientNet, OSNet or MobileNet) to extract features from detected objects (image patches). You can download checkpoints of REiD model from obtained link or search in the Git repository.
   - Patch preprocessing:
     - Generate a patch for each detected object. The size of each patch is defined by its bounding box => im_crops
     - Patch Resizing: Resize each patch (im_crops) to the size of the image patches used to train the ReID model you are using. If the provided checkpoints are utilized, resize to (64, 128), following the approach of the Market1501 dataset used to train this OSNet model.
     - Convert BGR to RGB
     - Normalize

     Here is an example function to preprocess image patches:

     ```
     def preprocess_patch(self, im_crops):
         roi_input = cv2.resize(im_crops, (self.roi_width, self.roi_height))
         roi_input = cv2.cvtColor(roi_input, cv2.COLOR_BGR2RGB)
         roi_input = (np.asarray(roi_input).astype(np.float32) - self.roi_means) /self.roi_stds
         roi_input = np.moveaxis(roi_input, -1, 0)
         object_patch = roi_input.astype('float32')
         return object_patch
     ```

   - Compute the ReID vector for each patch
   - Distance Metrics for ReID**:** Use metrics like cosine similarity or Euclidean distance to compare feature vectors of detected objects with those of tracked objects
2. Combine IoU and Feature Similarity to make the association more robust. One common approach is to use a weighted sum. You could define a combined score S as follows:

$$S = \alpha \cdot IoU + \beta \cdot Normalized\ Similarity$$

   where: α and β are weights that you can tune based on your application

   Normalized Similarity is obtained by normalizing the feature similarity score (e.g., for cosine similarity, this could be directly used; for Euclidean distance, you would need to invert it to get a similarity score).

$$Normalized\ Similarity = \frac{1}{1 + Euclidean\ Distance}$$

   This ensures that a lower distance results in a higher similarity score.

BONUS: Appearance-Aware IoU-Kalman Object Tracker: Detector Extension

1. Integrate more efficient **lightweight** deep Learning-based object detector for pedestrian detection. To accelerate tracker inference, you can regenerate a file named `det.txt` with the obtained detections and then read the file as you have been doing so far (Pay attention to the file format).

2. Extra: Evaluate the performance of your multi-object tracking system (the version you have been able to develop). "Utilize the ground truth data file (gt) to compute HOTA, IDF1, and ID_Switch metrics, as well as the tracking speed on your PC.
   - Download official evaluation kit TrackEval : https://github.com/JonathonLuiten/TrackEval
   - If you have any issues with the tool, feel free to ask me for a step-by-step tutorial guide

Evaluation

You are required to submit an **Individual Report** that outlines your experience with the project, accompanied by a **video** that visualizes the **tracking results**. In your report, clearly describe the tasks you undertook throughout the project. Identify any challenges you encountered and explain how you addressed them. **Upload the whole project to GitHub.**

***

Note finale sur 20=((Points TPs + Points Rapport)/ 100)×20 + Bonus (0–2 max)

1. **Travaux Pratiques progressifs (TPs)** – **70% de la note finale**

Ces TPs sont enchaînés : chaque TP reprend et enrichit le travail du précédent :

- TP1 : 10pts
- TP2 : 15pts
- TP3 : 20pts
- TP4 : 25pts
- TP5 : comporte des **extras optionnels** donnant lieu à un **bonus** (+1 point chacun sur 20, maximum +2, sans pénalisation si non réalisés).

2. **Rapport final** – **30% de la note finale**

- Clarté & structure : 15pts
- Analyse critique : 10pts
- Qualité des figures (vidéos) & légendes : 5pts

Deadline for project submission: 15.12.2025