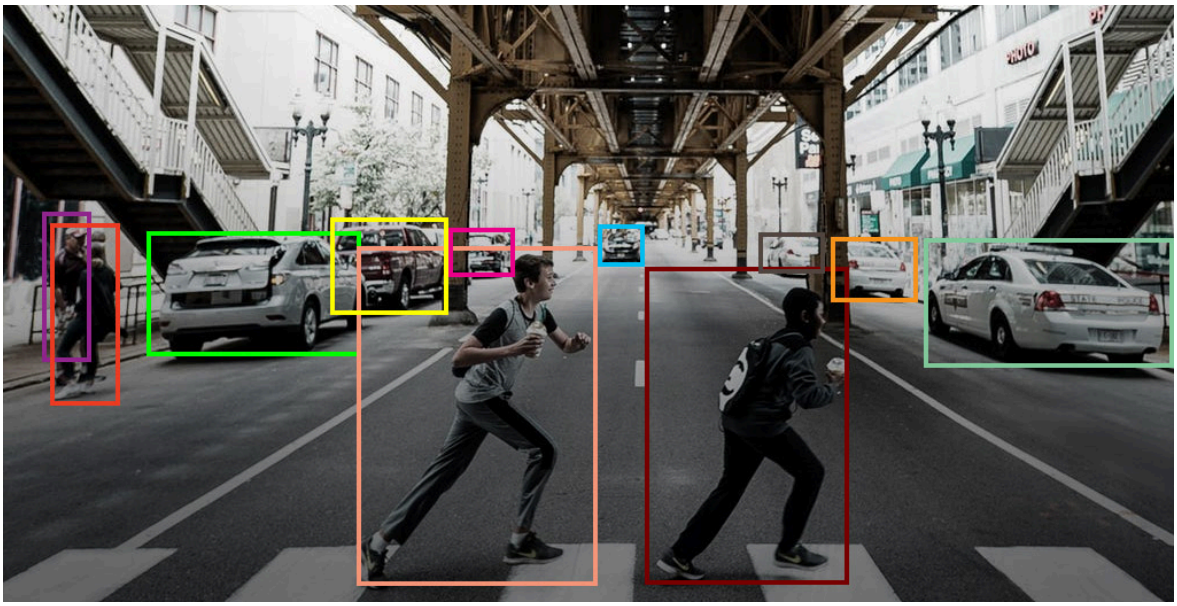
ENSEA
Beyond Engineering

TP2 : Object Tracking
Particle Filters

13.10.2025



Ziqian Peng, Kadidiatou Diallo, 3A SIA

Tracking simple objects in videos

Introduction

In this lab we build a very basic object tracker with a **Particle Filter.** The idea is to follow one simple object in a video by comparing its color histogram with many particle candidates, then updating and resampling the particles at each frame. The method follows the class handout (particle motion with Gaussian noise, likelihood from color-histogram distance, and systematic resampling).

For this first part, we use the video **escrime.avi**, where a **blue square** moves on a **black background**. The hidden state has three variables:

- **x**: horizontal position of the square's center (in pixels),
- **y**: vertical position of the square's center,
- **θ (theta)**: orientation angle of the square (in degrees), used to rotate the ROI when we compare histograms.

Workflow

- **Load the video (escrime.avi).**
  Purpose: open the data stream so we can read frames one by one and process them.
- **Display the first frame.**
  Purpose: visually check that loading works and see the object and background.
- **Select the ROI (initial rectangle).**
  Purpose: define the target area for the blue square at frame 1 (our starting ground truth). The tracker will try to keep this area "matched" over time.
- **Visualize the ROI.**
  Purpose: confirm the crop (right location, size, and rotation if used) before we build any reference features.
- **Compute the center and size of the TOI (target of interest).**
  Purpose: record the initial **(x, y, θ)** and width/height. These values set the mean and spread for particle initialization and define the patch size used for histograms.
- **Build RGB histograms; choose the Blue channel.**
  Purpose: test histograms for R, G, B in the ROI and pick the channel that separates object vs. background best. In our video the square is strongly **blue** and the background is **black**, so the **Blue** histogram gives the highest contrast and a cleaner likelihood signal (less noise from R/G). We keep only Blue to stay simple and fast.
- **Compute the reference histogram `href` (Blue, frame 1).**
  Purpose: store a **normalized** Blue histogram of the initial ROI. During tracking, each particle's patch histogram is compared to `href` to measure similarity (Bhattacharyya distance and likelihood).
- **Initialize particles.**
  Purpose: sample **N** particles around the initial center (and optional small spread in θ). Set all weights equal at start. This creates multiple hypotheses for where the square could be.
- **Tracking loop (PF + Bhattacharyya + orientation).**
  Purpose: repeat for each new frame:

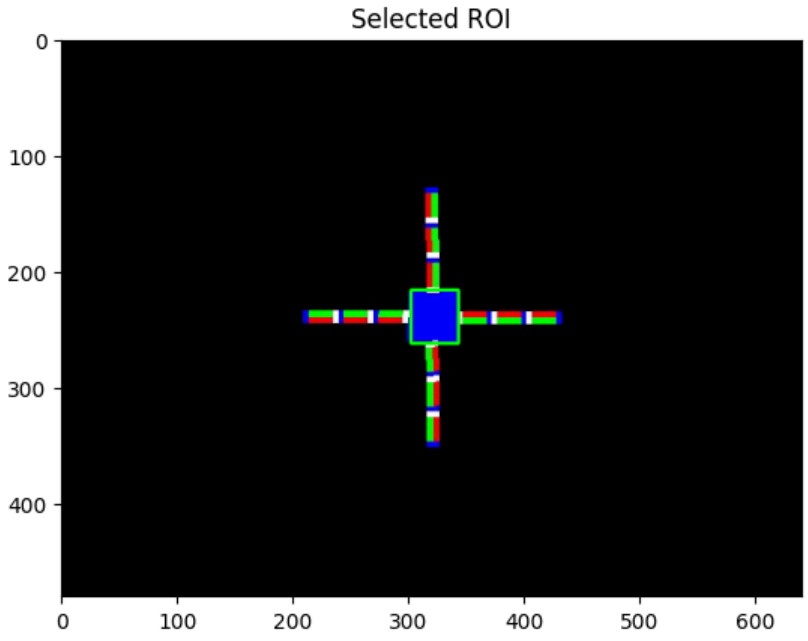
  Prediction: move particles with Gaussian noise (motion model)

  Correction: for each particle, extract the rotated ROI, compute Blue histogram, compare to `href` using **Bhattacharyya** distance; convert to likelihood and update weights.

  Resampling: fight weight degeneracy by replicating high-weight particles and removing low-weight ones (systematic resampling).

Video and ROI Initialization

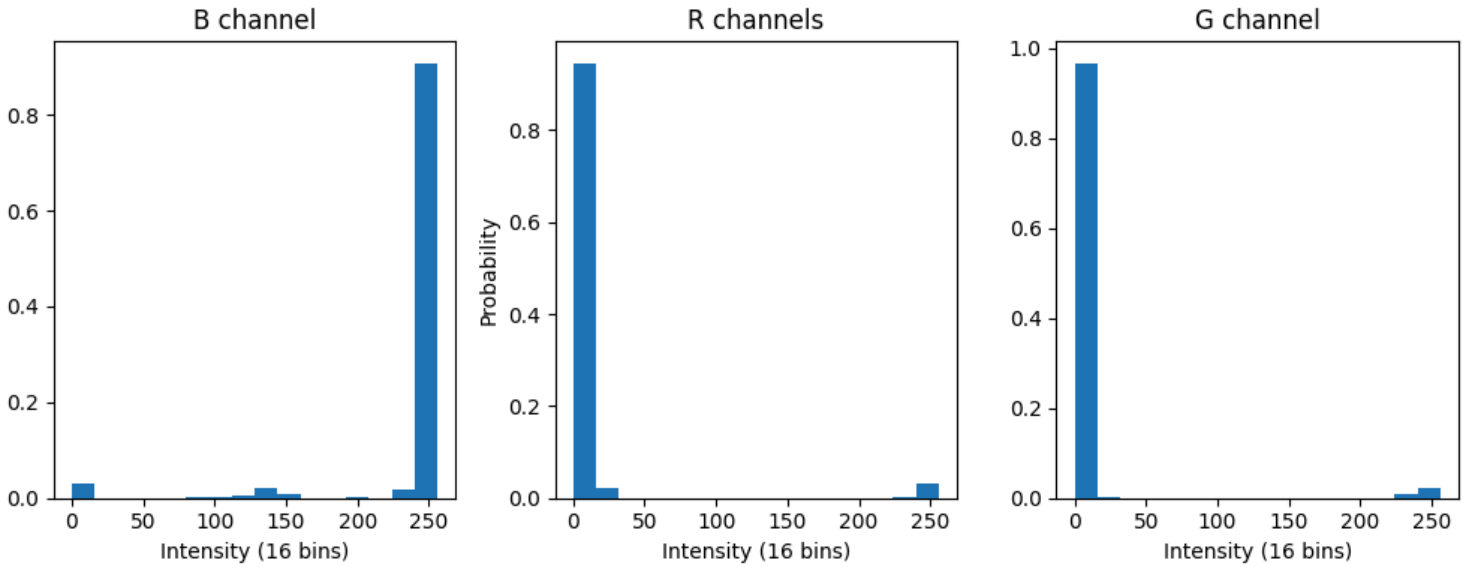
In steps 1–5, we load the video, show the first frame, select the target region, and record its geometry for the tracker. We use **OpenCV (cv2)** because it gives us simple tools for video I/O and interactive selection. After reading the first frame, `cv2.selectROI` opens a small window with a crosshair: we draw a rectangle around the blue square with the mouse and press Enter to confirm (Esc cancels). The function returns x, y, w, h where x, y is the top-left corner (in pixels) and w, h are the width and height. We immediately check that w and h are not zero to avoid an empty selection. For quick visual feedback, we copy the frame into vis, draw a green rectangle with `cv2.rectangle(vis, (x, y), (x + w, y + h), (0, 255, 0), 2)`, and display it. Because Matplotlib expects RGB while OpenCV stores BGR, we convert the colors with cv2.cvtColor before showing the image. Then we extract the actual ROI patch from the frame using `NumPy slicing: roi = frame[y:y+h, x:x+w]` (note that slicing is [rows, cols] → [y, x]). We compute the initial center as `center_roi = [x + w/2, y + h/2]`, which gives the `(cx, cy)` of the square in pixel coordinates, and we keep the size as `size_roi = (w, h)`. These center and size values define the initial target of interest (TOI) and will be used to initialize particles and to keep a consistent crop for histograms in the next steps.



Color histograms and href

We describe the target with a **color histogram**, which is just a count of pixel intensities. For one channel, we split the range 0–255 into **`Nb = 16 bins`** and count how many pixels fall in each bin. This gives a short vector (length 16) that captures the overall color of the ROI without caring about exact pixel positions. We then **normalize** the counts (divide by the total) so the histogram **sums to 1**. Normalization is useful because it removes the effect of ROI size or small changes in the crop: two ROIs with the same color distribution but different areas should look the same to the tracker. It also turns the histogram into a probability-like vector, which works well with distances such as **Bhattacharyya** in the weighting step.

We decided to compute histograms **per channel** (R, G, B) instead of mixing all channels at once. Separating channels keeps things **simple and robust**: each 1D histogram focuses on one color axis and is easy to interpret. In our video the background is black and the object is a bright **blue** square, so the **Red** and **Green** channel histograms stay 0 across all bins, while the **Blue** channel shows a **strong peak in the last bin** (high intensity). This clear separation tells us that the **Blue channel is the most discriminative** for our case.



We didn't use a joint **3D RGB histogram** (R×G×B bins) here. A 3D histogram would explode the number of bins (e.g., 16×16×16 = 4096), which needs is **slower** to compute for every particle and every frame. For a simple blue square on black, a 1D Blue histogram already gives a **clean signal** with far fewer parameters and less noise. Therefore, we set the **reference histogram** to the Blue one from the first frame, **`h_ref = h_B`** (L1-normalized). This compact signature is enough to compare each particle's rotated ROI to the target later using the Bhattacharyya-based likelihood.

Particle Initialization and Systematic Resampling

Particle Initialization & Systematic Resampling (Step 8) — with code pointers

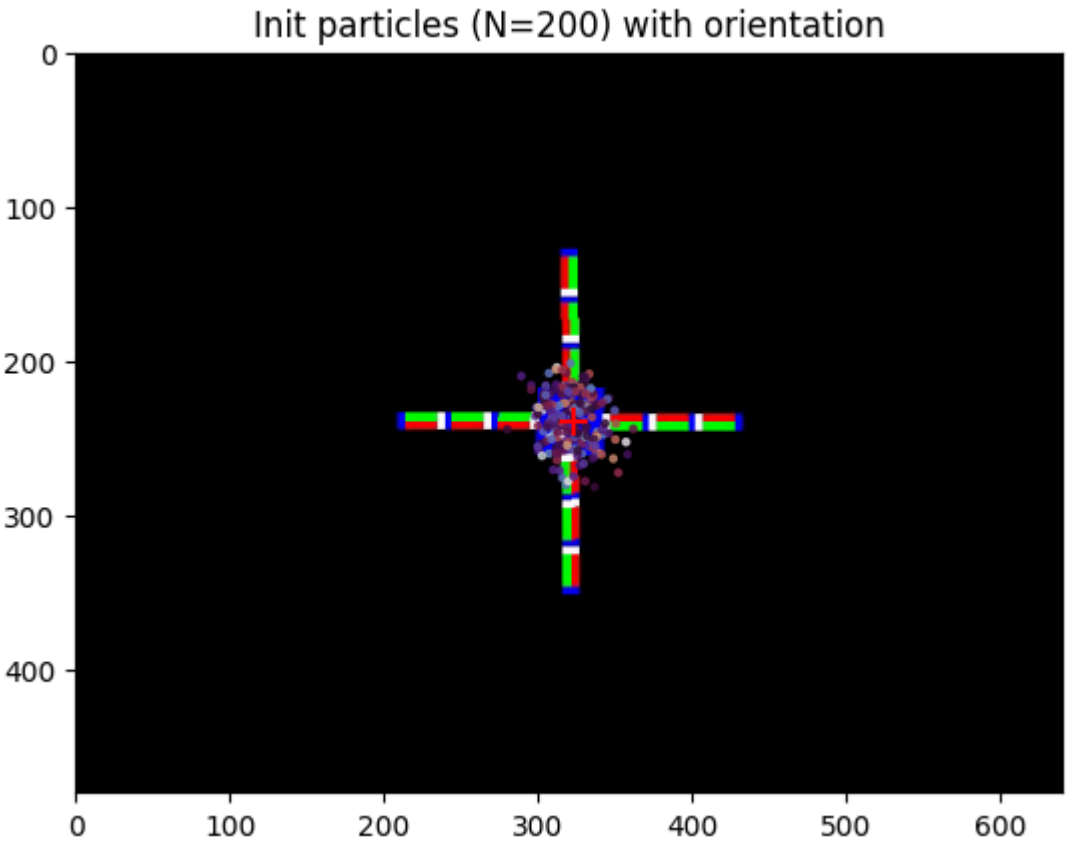
We initialize a cloud of **N = 200 particles** around the ROI center to represent hypotheses of the target state **(x, y, θ)**. Positions are sampled from a **Gaussian prior** centered at `(cx,cy)` with spread sigma=15, px, and orientations from N(θ0 =0∘, σθ=2∘). We **wrap** θ to [−180∘,180∘] to avoid drift across the discontinuity at ±180. All **weights start uniform**, wi=1/N, because before seeing new evidence, every hypothesis is equally likely. The spreads σpos and aσθ encode our **motion prior**: larger values increase robustness to initialization error but also raise ambiguity (more particles far from the true target).

Why we need resampling (and when we trigger it). After we update weights using the histogram likelihood in the tracking loop, a few particles usually dominate while most carry near-zero weight (**weight degeneracy**). This wastes computation and reduces accuracy. We monitor this with the **Effective Sample Size** $ESS = \frac{1}{\sum_i w_i^2}$ and resample when ESS falls below a threshold (e.g., N/2). Resampling replaces the current set by copies of high-weight particles (and drops low-weight ones), then resets weights to 1/N. This keeps diversity where the probability mass actually is.

Systematic resampling (what we use). Among resampling schemes, **systematic resampling** is low-variance and fast:

1. Build the cumulative weight array C (prefix sum of w).
2. Draw a single random offset u~U(0,1/N).
3. For j=0,…,N−1, set t=u+j/N and pick the smallest index i with C[i]≥t.
4. Copy particle i into the new set; after the loop, set all new weights to 1/N.

This procedure spreads the N picks **evenly across the weight mass**, yielding less variance than multinomial sampling and better stability frame to frame.



The scatter plot shows the **initialized particles** over the first frame. Points cluster around the red "+" (the ROI center). Color encodes **orientation θ** using a cyclic colormap : the slight color variations reflect the small σθ we allowed. The tight cluster confirms our prior: we believe the object is near its initial center with only small pose uncertainty. This is a good starting point for the prediction–update–resample cycle that follows.

Tracking loop (PF, Bhattacharyya, Orientation)

We run one frame-at-a-time cycle with five stages: **Prediction, Correction (likelihood), Normalization, Estimation, Resampling**, then **Visualization**.

First, we define the helpers. **`blue_hist_rotated(...)`** rotates the whole frame around (xcenter , ycenter) by θ using **`cv2.getRotationMatrix2D`** + **`cv2.warpAffine`** (border replicate), crops an axis-aligned patch [x − w/2:x + w/2, y − h/2:y + h/2], converts to RGB, takes the **Blue** channel, and returns an **L1-normalized** 1D histogram with **`nbins`** bins (code: **`blue_hist_rotated(...)`**). **`bhattacharyya(h1,h2)`** computes the Bhattacharyya distance ($d_B = \sqrt{1 - \sum_i \sqrt{h_{1,i} h_{2,i}}}$) (code: **`bhattacharyya`**). **`systematic_resample(weights`**) implements low-variance systematic resampling (prefix sum CDF + one offset) and returns the selected indices (code: **`systematic_resample`**). **`draw_oriented_box(...`**) only handles display.

Inside the loop, we **predict** particle motion. For the first few frames we "warm up" to stick to the initial ROI: we sample positions near **`center_roi`** with small noise and tiny orientation noise (**`WARMUP_FRAMES`**, code block "A) PREDICTION"). After that, we add Gaussian noise with standard deviations **`sigma_move`** (pixels) and **`sigma_theta`** (degrees). We enforce image bounds for (x,y) and wrap angles to ([-180°,180°]) (code: angle wrap + **`np.clip`**).

For **correction**, each particle [xi,yi,θi] gets a **candidate Blue histogram** **`h_cand = blue_hist_rotated(frame, xi, yi, w_roi, h_roi, ti, nbins=Nb)`**. We compute the **distance** to the reference **`h_ref`** with **`d = bhattacharyya(h_ref, h_cand)`** and convert it to a **likelihood weight** with a Gaussian kernel: **`weights[i] = exp(-lambda_b * d**2)`**, where **`lambda_b`** tunes how strongly color similarity influences weights (code: "B) CORRECTION"). We then apply **numerical safety** (replace NaN/Inf, guard near-zero sums) and **normalize** the weights to sum to 1 (code: checks on **`weights`** and **`weights /= s`**).

Next we estimate the state: x^,y^ are the weighted means (cx_est, cy_est), while θ^ uses a circular mean via weighted sin/cos and atan2 to avoid wrap-around bias (code: "C) ESTIMATION").
We then do systematic resampling to fight weight degeneracy: we build the CDF of `weights`, draw a single random offset, select indices at evenly spaced thresholds, copy the chosen particles, and reset all weights to 1/N (code: "D) RESAMPLING" with `idx = systematic_resample(weights); particles = particles[idx]; weights[:] = 1.0/N)`. This keeps many hypotheses near high-likelihood regions with low variance between frames.

Finally, for visualization, we draw the estimated oriented rectangle with `draw_oriented_box(vis, cx_est, cy_est, w_roi, h_roi, theta_est, ...)` and show it live (`cv2.imshow(...)`) (code: "E) VISU"). Key parameters we can tune are `lambda_b` (likelihood sharpness), `sigma_move / sigma_theta` (motion noise), `Nb` (hist bins), and the warm-up length.

Together, these steps implement a **full Particle Filter with Bhattacharyya-based color likelihood** and systematic resampling for robust tracking of the blue square.

Test with différent setting parameters

- We tested different bin numbers (Nb = 8, 16, 64) to see the effect on color discrimination. With **Nb = 8**, results were identical to 16, confirming that the color distribution is simple enough for a coarse histogram. With **Nb = 64**, it was slightly different and more precise, showing that too many bins make the histogram noisy and unstable since the ROI is small and mostly uniform blue. Thus, **Nb = 16** offers the best balance between precision and robustness.
- We also compared different particle counts (N = 100, 200, 400) to balance accuracy and runtime. With **N = 100**, tracking stayed stable but slightly less smooth, while **N = 200** gave the best compromise. Increasing to **N = 400** brought almost no gain in accuracy but doubled the computation time. Therefore, **N = 200** is a good trade-off between precision and efficiency.
- Nonetheless, it should be noted that these comparisons are made **visually**, since we do not have the **ground-truth motion coordinates** to quantify the error with precise metrics. Therefore, we can only estimate the **variance or consistency between different tracks**, rather than their absolute accuracy.

Test with residual resampling

We now test the **residual resampling** version of the Particle Filter to compare it with the previous **systematic resampling**. The main difference is that residual resampling combines a **deterministic phase** and a **random phase**: each particle is first copied a fixed number of times according to the integer part of (N*w_i), and the remaining slots are filled randomly based on the fractional residuals. This ensures that **high-weight particles are always kept** while still maintaining some diversity among the others. I

In practice, this method tends to reduce random fluctuations compared to systematic resampling, but here we saw no difference since the tracking problem is extremely simple: the blue square has a uniform color and a very stable motion, so both **systematic** and **residual resampling** maintain almost identical weights over time. But now it took 2min. It's a little bit longer.

Test with reduced frames

In the 9th step, we add

```
REDUCE_BY_TWO = True
```

and

```
if REDUCE_BY_TWO:

cap.grab()
```

With every other frame skipped and no parameter changes, the tracker got noticeably worse: the box fell behind during fast motion, angle updates reacted more slowly, and the box looked a bit shakier after sudden moves. **Halving** the frame rate **doubles** the time between updates (Δt), so the target moves and rotates more each step while our motion noise (σ_move, σ_theta) is still tuned for the original cadence and the likelihood (λ, Nb) stays relatively strict. Fewer measurements and a motion prior that's too narrow so poorer coverage of the true state, harsher penalties from the model, and a less confident filter.

The tracker must be retuned : we could widen the motion prior by increasing **σ_move** (about 6 to12 px) and **σ_theta** (about 2 to 5°) to cover the larger inter-frame displacement and rotation. We can also relax the measurement by slightly lowering **λ** and improve robustness by raising **N** to better explore the state space.

Object with varying size

When the object can change size, we can extend the state from [x, y, θ] to **[x, y, θ, s]**, where **s** is a scale factor. In prediction, add small noise on scale (e.g., on log-scale), and in measurement compute the histogram on a **rotated and scaled** window : width = $s \cdot w_0$, height = $s \cdot h_0$.

In terms of accuracy, allowing scale could **reduce center error** during real size changes, because the window matches the object better and leaks less background.

For better accuracy, we could slightly **lower λ** to tolerate appearance shifts as the window area grows and **increase N** (e.g., 300–400) to explore position, angle, scale at once.