

Condensation Tracker

Computer Vision Lab 06 Report

HS2023

CHRISTOPHER ZANOLI

czanoli@ethz.ch

Student Number: 23-942-394

Abstract

The report is divided into two main sections: **Condensation Tracking Implementation** and **Condensation Tracking Experiments**. In the first section, I provide a comprehensive analysis of the implementation decisions, while in the second section I discuss the results both at the level of theoretical foundations and empirical observations.

1 Condensation Tracking Implementation

The following are the main implementation steps:

1.1 Color Histogram

The `color_histogram()` function enables the tracking of an object in a video sequence by providing a color-based feature descriptor that can be compared across frames for similarity assessment.

The first step consists of performing the bound box constraints. This ensures that the coordinates of the bounding box are within the valid range of the frame's dimensions, preventing errors from trying to access areas outside the image. Coordinates are rounded to the nearest integer since pixel indices in an image must be integers. The second step consists of extracting the bounding box from the frame through slicing. The third step involves computing the histogram for each RGB channel, leveraging the `cv2.calcHist()` function. Finally, the three histograms are concatenated into a single array and then flattened. The resulting histogram is normalized by dividing by the total sum of all bins, which makes the histogram's sum equal to 1, thus creating a probability distribution.

```
1 def color_histogram(xmin, ymin, xmax, ymax, frame, hist_bin):
2     xmin = round(max(1, xmin))
3     xmax = round(min(xmax, frame.shape[1]))
4     ymin = round(max(1, ymin))
5     ymax = round(min(ymax, frame.shape[0]))
6
7     bounding_box_img = frame[ymin:ymax, xmin:xmax]
8
9     hist_R = cv2.calcHist([bounding_box_img], [0], None, [hist_bin], [0, 256])
10    hist_G = cv2.calcHist([bounding_box_img], [1], None, [hist_bin], [0, 256])
11    hist_B = cv2.calcHist([bounding_box_img], [2], None, [hist_bin], [0, 256])
12
13    hist = np.concatenate((hist_R, hist_G, hist_B)).flatten()
14    hist = hist / sum(hist)
15
16    return hist
```

Figure 1: `color_histogram()` Implementation

1.2 Matrix A Derivation

In our scenario, we consider two prediction models:

- No motion at all, i.e. just noise
- Constant velocity motion model.

The following are the steps performed to derive the dynamic matrix matrix A for both cases:

1.2.1 No-Motion Model

In the no-motion model, we assume that the object does not move by itself. This means that any observed motion is purely the result of the position noise $w = [\sigma_{P_x}, \sigma_{P_y}]^T$. The state vector for each particle is defined as $s = [x, y]^T$, where x and y represent the object's position.

The crucial concept is that the object's position does not change between frames due to the model's assumptions. Therefore the object has no inherent velocity or other dynamics influencing its position.

Since the object's position does not change due to the model, we need an identity matrix that, when multiplied by the state vector, leaves it unchanged. This leads to:

$$A = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

1.2.2 Constant Velocity Motion Model

In the constant velocity motion model, we assume that the object moves at a steady speed and direction. The state vector for each particle includes both position and velocity and is defined as $s = [x, y, \dot{x}, \dot{y}]^T$, where \dot{x} and \dot{y} represent the object's velocity in the x and y directions, respectively. Since now we are taking into account both position and velocity, the noise will be represented by $w = [\sigma_{P_x}, \sigma_{P_y}, \sigma_{V_x}, \sigma_{V_y}]^T$

In this case, the assumption is that the object's velocity is constant over time, and the next position is determined by the current position plus the change in position due to the velocity over a unit of time. This means that the equations of motion are:

$$x_{t+1} = x_t + \dot{x} \cdot \Delta t$$

$$y_{t+1} = y_t + \dot{y} \cdot \Delta t$$

In order to encode these equations of motion into matrix form, we need to construct a matrix that, when applied to the state vector, updates the position based on the velocity. This leads to:

$$A = \begin{bmatrix} 1 & 0 & \Delta t & 0 \\ 0 & 1 & 0 & \Delta t \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Where the upper right 2x2 sub-matrix contains the Δt terms, which update the position, while the lower right 2x2 identity sub-matrix maintains the constant velocity.

From an implementation point of view, I decided to set $\Delta t=1$ to simplify the model. This allows us to align the time step with the frame rate of the video. Therefore, the position update directly corresponds to the number of pixels moved from one frame to the next.

1.3 Propagation

This function is responsible for moving each particle forward in time according to the selected dynamic model and introducing randomness to account for the inherent uncertainty in the motion of the object being tracked. This step allows the filter to generate a new set of hypotheses, i.e. particles, about the possible future state of the object.

The first step is represented by the parameters and state initialization. After that, there are the matrix A and noise definitions, according to the selected model. The next step is noise incorporation: noise

is added to the prediction to account for the uncertainty in the motion model. This is done by multiplying a noise vector, whose components are standard deviations, with a randomly generated vector from a normal distribution. Next, the particle state update comes into play: each particle's state is updated by multiplying its previous state with matrix A and adding the noise vector. This step propagates the particle through the state space according to the motion model. Finally, a frame boundary enforcement is performed, to ensure that particles do not go outside the image frame.

```

1  def propagate(particles, frame_height, frame_width, params):
2      num_particles = params['num_particles']
3      state_length = particles.shape[1]
4      sigma_p = params['sigma_position']
5      sigma_v = params['sigma_velocity']
6      dt = 1
7
8      if params['model'] == 0:
9          A = np.array([[1, 0],
10                       [0, 1]])
11          noise = np.array([sigma_p, sigma_p])
12
13      elif params['model'] == 1:
14          A = np.array([[1, 0, dt, 0],
15                       [0, 1, 0, dt],
16                       [0, 0, 1, 0],
17                       [0, 0, 0, 1]])
18          noise = np.array([sigma_p, sigma_p, sigma_v, sigma_v])
19
20      for i in range(num_particles):
21          w = noise * np.random.randn(state_length)
22          particles[i, :] = A @ particles[i, :] + w
23
24          particles[i, 0] = np.clip(particles[i, 0], 1, frame_width)
25          particles[i, 1] = np.clip(particles[i, 1], 1, frame_height)
26
27          if state_length == 4:
28              particles[i, 2:] = A[2:, 2:] @ particles[i, 2:]
29
30      return particles

```

Figure 2: propagate() Implementation

1.4 Observation

The `observe()` function is crucial for associating the predicted state of each particle with the actual observations from data. The purpose of this function is to update the weights of each particle based on how well the particle's predicted state agrees with the actual observations.

The first step is computing the bounding box for each particle based on its center position and the dimensions of the bounding box. This determines the region of the image frame that corresponds to where the particle believes the object should be. After that, for each particle, `color_histogram()` is called to calculate the color histogram within the particle's bounding box. This histogram is a representation of the observed data for that particle. The third step involves leveraging the `chi2_cost()` function to calculate the χ^2 distance between the color histogram of the observed data and the target color histogram. Each particle's weight is updated using the provided equation, which involves a Gaussian probability density function. The weights reflect the likelihood of each particle based on the χ^2 distance, i.e. particles with a histogram closer to the target have higher weights. After that, the weights of all particles are normalized so that they sum to 1. This normalization is important because it allows the weights to represent a probability distribution over the particles.

```

1 def observe(particles, frame, bbox_height, bbox_width, hist_bin, hist_target, sigma_observe):
2     num_particles = particles.shape[0]
3     particles_w = np.zeros(num_particles)
4
5     x_min = particles[:, 0] - bbox_width / 2
6     x_max = particles[:, 0] + bbox_width / 2
7     y_min = particles[:, 1] - bbox_height / 2
8     y_max = particles[:, 1] + bbox_height / 2
9
10    for i in range(num_particles):
11        x_min[i] = max(1, min(frame.shape[1], x_min[i]))
12        x_max[i] = max(1, min(frame.shape[1], x_max[i]))
13        y_min[i] = max(1, min(frame.shape[0], y_min[i]))
14        y_max[i] = max(1, min(frame.shape[0], y_max[i]))
15
16        hist_i = color_histogram(x_min[i], y_min[i], x_max[i], y_max[i], frame, hist_bin)
17
18        chi_dist = chi2_cost(hist_target, hist_i)
19
20        particles_w[i] = 1 / (np.sqrt(2 * np.pi) * sigma_observe) * np.exp(-0.5 * chi_dist**2 / (sigma_observe**2))
21
22    particles_w /= np.sum(particles_w)
23
24    return particles_w

```

Figure 3: observe() Implementation

1.5 Estimation

The weighted mean state of the particles in the particle filter is computed by this function. To begin, the weights `particles_w` are reshaped to be in a two-dimensional column vector format, allowing for element-wise multiplication with the particles array. The function then computes the weighted sum of the particles' states by multiplying the particles by their weights element by element and summing the result. The weighted mean state is then obtained by normalizing the sum by the entire sum of the weights.

```

1 def estimate(particles, particles_w):
2     particles_w = particles_w.reshape(-1, 1)
3     mean_state = np.sum(particles * particles_w, axis=0) / np.sum(particles_w)
4     return mean_state

```


Figure 4: estimate() Implementation

1.6 Resampling

The purpose of the `resample()` function is to discard low-probability particles and concentrate on those with higher likelihoods, as indicated by their weights.

The strategy is designed to reduce the randomness associated with the resampling step. It ensures that particles with higher weights are more likely to be selected, while those with lower weights may not be selected at all. The variable `r` is initialized with a random number between 0 and $1/\text{numparticles}$. This helps to offset the starting point for the resampling wheel. The cumulative weight `w` is initialized with the weight of the first particle.

A loop iterates over all particles. For each iteration, it calculates a threshold `U` based on the initial random offset `r` and the particle's index. The while loop increments index `i` until the cumulative sum of weights exceeds `U`. This step effectively selects the particle at index `i` for duplication, based on its weight. Finally, the selected particle's state and weight are duplicated into the new set of resampled particles. After resampling, the weights are normalized so that the new set of particles has a sum of weights equal to 1. This is important as the resampling process can introduce duplicates, and normalizing ensures that the weights remain a valid probability distribution.



```

1  def resample(particles, particles_w):
2      num_particles = particles.shape[0]
3      state_length = particles.shape[1]
4
5      particles_resampled = np.zeros((num_particles, state_length))
6      particles_w_resampled = np.zeros(num_particles)
7
8      r = np.random.uniform(0, 1/num_particles)
9      w = particles_w[0]
10     i = 0
11
12     for m in range(num_particles):
13         U = r + m * (1 / num_particles)
14
15         while U > w:
16             i = (i + 1) % num_particles
17             w = w + particles_w[i]
18
19         particles_resampled[m, :] = particles[i, :]
20         particles_w_resampled[m] = particles_w[i]
21
22     particles_w_resampled /= np.sum(particles_w_resampled)
23
24     return particles_resampled, particles_w_resampled

```

Figure 5: resample() Implementation

2 Condensation Tracking Experiments

This section shows and discusses the various results obtained on the 3 available videos and answers the various questions requested by the handout.

The plots that will be shown in this section will present two trajectories:

- The blue trajectory indicates the mean state of the priori particles
- The red trajectory indicates the mean state of the posterior particles

2.1 Video 1

The first video shows a hand moving through space without occlusions. The initial frame shows only the upper part of the hand (the fingers), making tracking the entire hand in later frames quite complicated. The speed of the hand is fairly constant, except in the first few frames where there is a small acceleration and in the last frames where there is a downward acceleration of the hand.

Below are the results obtained with the **default parameters** for both $model_0$ (no-motion) and $model_1$ (constant velocity).

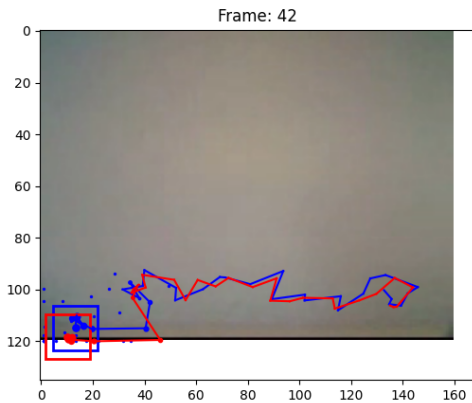


Figure 6: $Model_0$

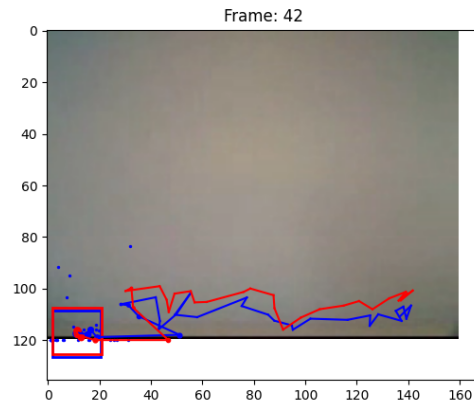


Figure 7: $Model_1$

As can be seen from the results obtained with the default parameters, both tracks are quite satisfactory. However, a common problem with both is the fact that at one point the tracker was tracking the wrist instead of the hand. Furthermore, the produced trajectories are wandered rather than smooth. The following parameter configuration turned out to be the best to solve these issues, for both $model_0$ and $model_1$:

```
1  params = {
2      "draw_plots": 1,
3      "hist_bin": 32,
4      "alpha": 0.5,
5      "sigma_observe": 0.2,
6      "model": 1,
7      "num_particles": 45,
8      "sigma_position": 8,
9      "sigma_velocity": 1,
10     "initial_velocity": (-5, -15)
11 }
```

Figure 8: $Video_1$ parameters change

Below are the **successful** results obtained with these new 2 configurations of parameters for $model_0$ and $model_1$ respectively:

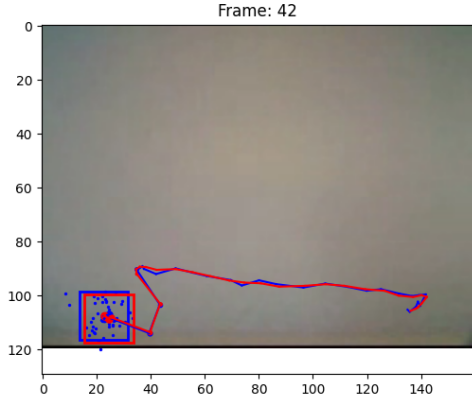


Figure 9: $Model_0$

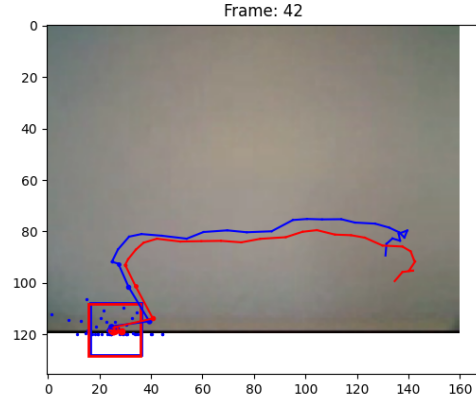


Figure 10: $Model_1$

After running several runs trying many different combinations, what I noticed is that in the case of $model_0$ the parameters that have the most influence on the results are:

- `alpha`
- `sigma_position`

In the case of $model_1$ the parameters that have the most influence on the results are:

- `alpha`
- `sigma_position`
- `initial_velocity`

In particular, an increase in the value of `alpha` makes it possible to cope with the change in hand appearance by updating the target model's color histogram over time. This is important because, as mentioned before, the shading of the hand changes significantly from the first frame to subsequent frames. As for the `sigma_position`, it has been observed that its value effectively influences the degree of randomness introduced to the position estimates of particles during the propagation step. In this particular case, I decreased its value since a tighter `sigma_position` keeps particles more tightly clustered, enhancing the trajectory smoothness.

Finally, only for $model_1$, the `initial_velocity` plays a key role in predicting correctly the subsequent positions of the particles. In fact, in $model_1$, the assumption is that the object maintains a roughly constant velocity over time, so starting with a realistic velocity estimate (based on the observed movement of the hand) can align the particles' predictions more closely with the actual motion of the object.

As specifically requested by the handout, below are also an example of **unsuccessful** results for both the $model_0$ and $model_1$:

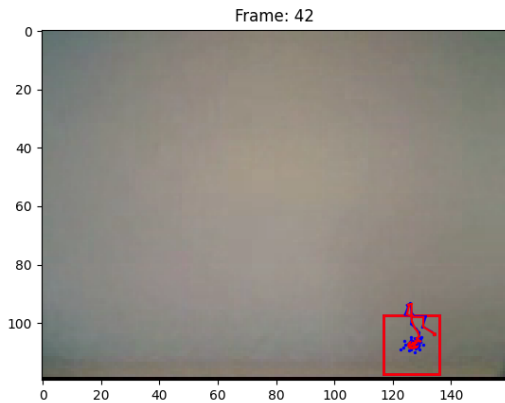


Figure 11: $Model_0$

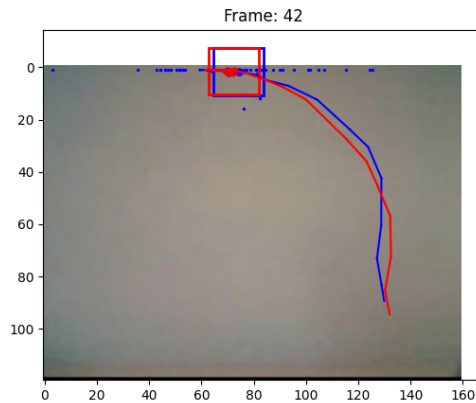


Figure 12: $Model_1$

There are a variety of parameter combinations that can cause unsuccessful results for both *model*₀ and *model*₁. I decided to report these two interesting results, where the crucial parameters are `sigma_position` and `sigma_observe` for *model*₀ and *model*₁ respectively. In the first case, a too-low `sigma_position` value (`sigma_position` = 1) causes the filter to be unable to catch up with the object, as the particles don't cover enough area to capture the new position. In the second case, a `sigma_observe` that is too high (`sigma_observe` = 0.8) undermines the particle filter's ability to effectively track the target, making it less accurate in its state estimation. This is potentially due to ineffective weighting, i.e. the weight distribution among particles becomes overly uniform, eventually failing to significantly differentiate between more accurate and less accurate particles.

2.2 Video 2

The second video again shows the movement of a hand. In this case, the main obstacle is the presence of an occlusion object and the presence of an object in the background (the poster). The movement of the hand is definitely more steady and slower than in the previous video.

Below are the results obtained with the **default parameters** for both *model*₀ (no-motion) and *model*₁ (constant velocity).

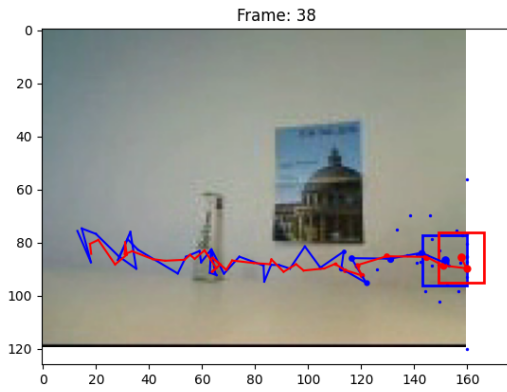


Figure 13: *Model*₀

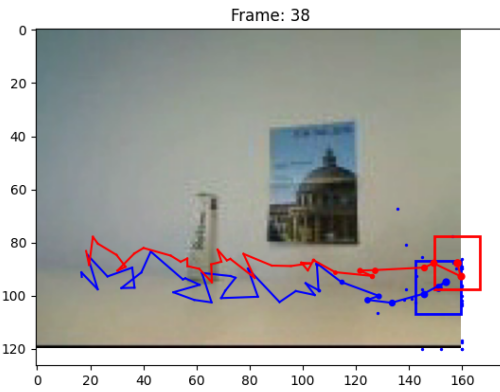


Figure 14: *Model*₁

Again, as can be seen from the results obtained with the default parameters, both tracks are quite satisfactory. However, both results have the same problems as those reported for *video*₁. The following parameter configuration turned out to be the best to solve these issues for *model*₁:

```

1  params = {
2      "draw_plots": 1,
3      "hist_bin": 32,
4      "alpha": 0.5,
5      "sigma_observe": 0.2,
6      "model": 1,
7      "num_particles": 100,
8      "sigma_position": 1,
9      "sigma_velocity": 1,
10     "initial_velocity": (1, 0)
11 }

```

Figure 15: *Video*₂ parameters change

Below is the **successful** results obtained with this new configuration of parameters for *model*₁:

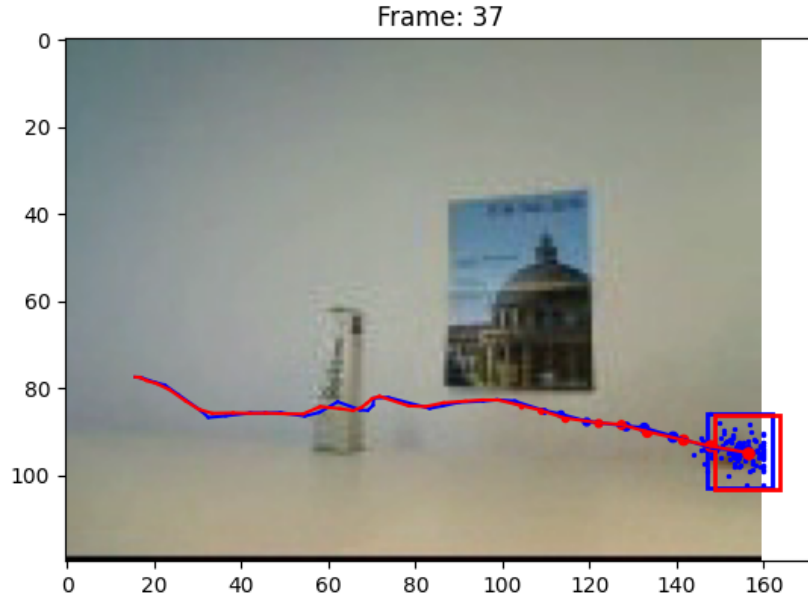


Figure 16: $Model_1$

In this case, given the setting and previously explained behavior of the object to be tracked, $model_1$ (constant velocity) is more appropriate than $model_0$. This reasoning highlights a negative aspect of the condensation tracker, namely the fact that the decision on which motion model to use is often made by considering the nature of the scene and the expected object behavior.

This allows us to answer the first of the 3 questions asked in the handout:

What is the effect of using a constant velocity motion model?

The use of the $model_1$ for this particular case is absolutely beneficial. This model is particularly good at predicting the trajectory of the hand during linear and steady motion, allowing for efficient tracking over video frames. This model's ability to maintain tracking continuity throughout times of occlusion is a crucial strength. In fact, it extrapolates the hand's position based on its last detected velocity, providing a solid forecast when the hand is briefly not visible.

Furthermore, by emphasizing motion over appearance, the model successfully distinguishes the moving hand from fixed background components (in this case the poster), lowering the likelihood of tracking mistakes due to background interference.

The primary drawback of the constant velocity model is its implicit assumption of a continuous motion pattern, which can make detecting rapid changes in hand movement, such as sudden stops or directional switches, difficult. This limitation is mitigated in the context of the second video, as the hand's motion does not exhibit such abrupt changes. Because the hand maintains a constant trajectory throughout, the constant velocity model is ideal for this tracking case.

Below are shown **unsuccessful** results for $model_1$, based on system noise (`sigma_position` and `sigma_velocity`) and the measurement noise (`sigma_observe`):

System Noise Variations

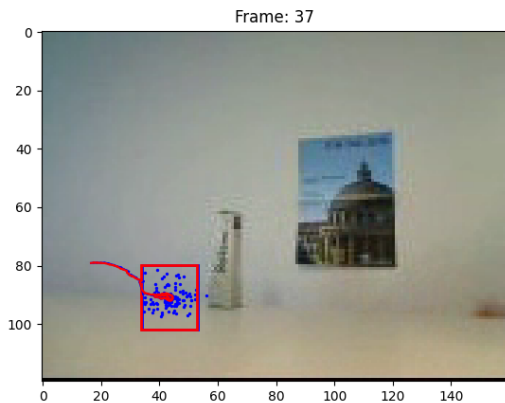


Figure 17: Too-low `sigma_velocity`

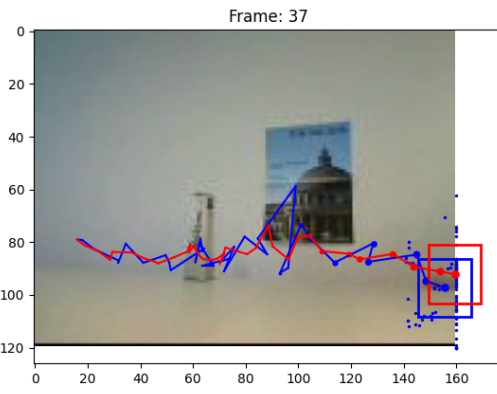


Figure 18: Too-high `sigma_velocity`

Measurement Noise Variations

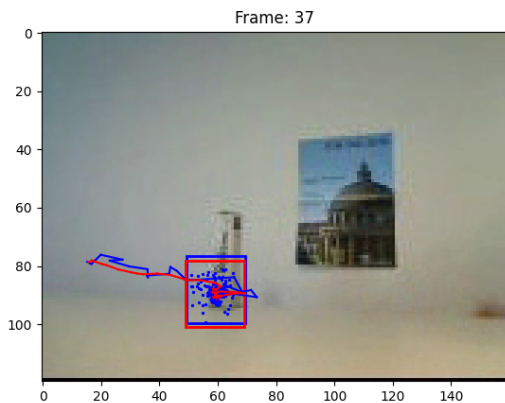


Figure 19: Too-low `sigma_observe`

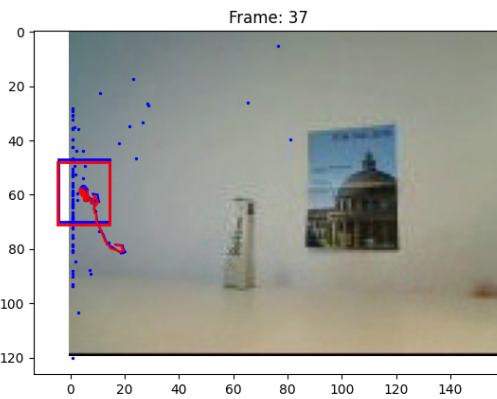


Figure 20: Too-high `sigma_observe`

These results allow us to answer the other 2 questions in the handout:

*What is the effect of assuming decreased/increased system noise?
What is the effect of assuming decreased/increased measurement noise?*

For $model_1$ the system noise is comprised of `sigma_position` and `sigma_velocity`.

Regarding `sigma_position`, increasing its value also increases the filter's adaptability to abrupt changes in the object's position, as particles are more dispersed and can cover a broader area of the state space. However, a too-large value may result in an overly diffused particle cloud that can lose precision in the estimate. On the other hand, decreasing `sigma_position` enhances the precision of the tracking by focusing the particle spread in a smaller region, which is beneficial when the object's motion is smooth and predictable, such as in this case. However, a value that is too low, can make the filter unable to catch up with the object, as the particles may not cover enough area to capture the new position.

Regarding the `sigma_velocity`, decreasing its value enhances tracking accuracy. In fact, lower `sigma_velocity` implies greater confidence in the motion model, leading to more precise predictions

of the hand's movement, which is beneficial in this particular case. Moreover, particles are less likely to deviate significantly from the predicted path, focusing the tracking on a narrower area. Similar to what was said for `sigma_position`, a too-low value of `sigma_velocity` can potentially lead to a motion complexity underestimation, i.e. the model cannot keep up with the object's motion, potentially resulting in tracking errors, such as the one shown in figure 17. Conversely, An increased `sigma_velocity` corresponds to a greater adaptability to unexpected motion since higher system noise allows particles to explore a wider area of the state space. The main drawback is a potential decreased precision in tracking. In fact, with higher noise, particle predictions are more spread out, which might reduce the accuracy of tracking, especially since the hand movement is relatively steady and slow. Figure 18 shows exactly this issue.

Regarding the measurement noise, a decrease in `sigma_observe` leads to improved confidence in observations. This is because a lower measurement noise means the filter places more trust in the observed data (color histogram), potentially improving tracking accuracy when the observations are reliable. The main drawback is that if the observed data is affected by occlusions or background objects (such as in this case), too much confidence in these observations might lead to incorrect tracking. Figure 19 shows exactly this issue. On the other hand, an increase in `sigma_observe` makes the filter more tolerant of inaccuracies in the observations, which can be beneficial in handling the complexities introduced by the background object (the poster) and occlusions. However, a too-high value could result in a more diffused weighting of particles, potentially leading to wrong tracking. Figure 20 shows exactly this problem.

2.3 Video 3

In this case we have a black ball moving horizontally from left to right, bouncing off a wall, and coming back along the same direction. The two main obstacles in this setting are the fact that in the impact with the wall the ball undergoes a change in velocity and the fact that the lower right part of the image present a dark part that can potentially generate a color histogram very similar to that of the object to be tracked.

As required by the handout, the starting point is to observe the result obtained with the best model used for *video2*:

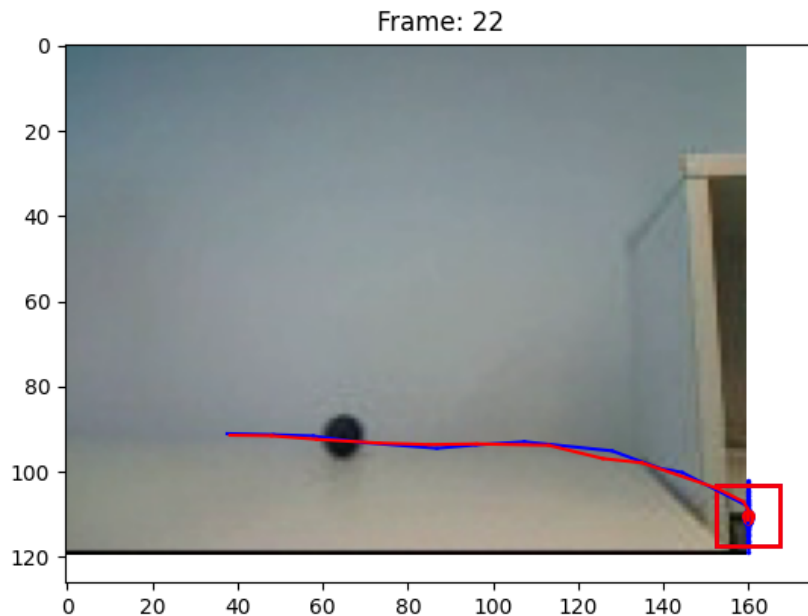


Figure 21: Best *video2* model result on *video3*

As expected, *model1* manages to effectively handle the tracking of the object in the first phase, when the ball's velocity is constant, however, as soon as the bounce on the wall occurs, the model becomes unstable and tracking focuses on the lower right area, whose color histogram is very similar to that of the object of interest.

This behaviour allows us to answer the question 2.2 also for *video₃*. In this particular case, *model₀* (no motion) emerges as a better alternative than *model₁* (constant velocity). This preference derives mostly from the ball's dynamic contact with the wall, which causes a significant shift in velocity upon impact (a condition in which the constant velocity model may struggle). *Model₀* is less prone to errors caused by such abrupt directional changes because it does not rely on velocity predictions. Furthermore, the appearance of a dark area in the lower right corner of the image, which may approximate the ball's color histogram, poses an additional problem. A model that does not assume continuing motion allows for a more flexible response to these variations in the ball's trajectory, making it better suited to precisely tracking the ball's movement in this specific context.

The following parameter configuration turned out to be the best to solve these issues for *model₀*:

```

1  params = {
2      "draw_plots": 1,
3      "hist_bin": 32,
4      "alpha": 0,
5      "sigma_observe": 0.1,
6      "model": 0,
7      "num_particles": 100,
8      "sigma_position": 10,
9      "sigma_velocity": 3,
10     "initial_velocity": (10, 0)
11 }

```

Figure 22: *Video₃* parameters change

Below is the **successful** result obtained with this new configuration of parameters for *model₀*:

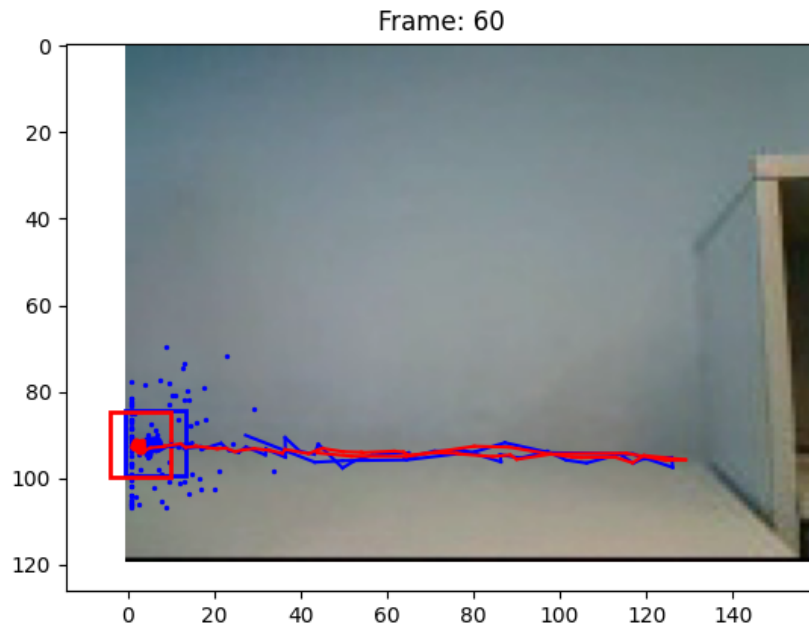


Figure 23: *Model₀*

Below are shown **unsuccessful** results for *model₀*, based on system noise (*sigma_position*) and the measurement noise (*sigma_observe*):

System Noise Variations

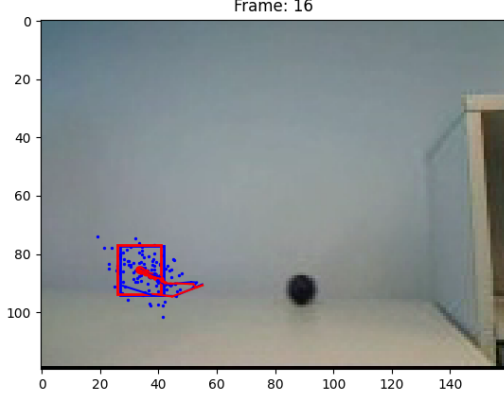


Figure 24: Too-low `sigma_position`

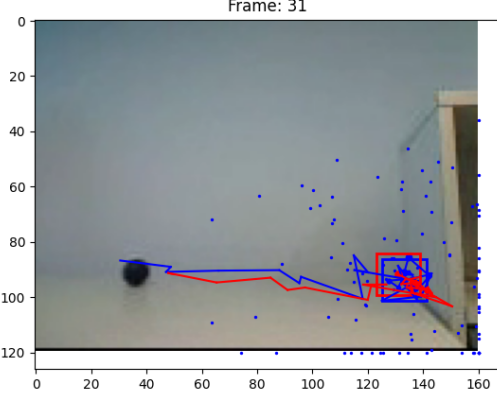


Figure 25: Too-high `sigma_position`

Measurement Noise Variations

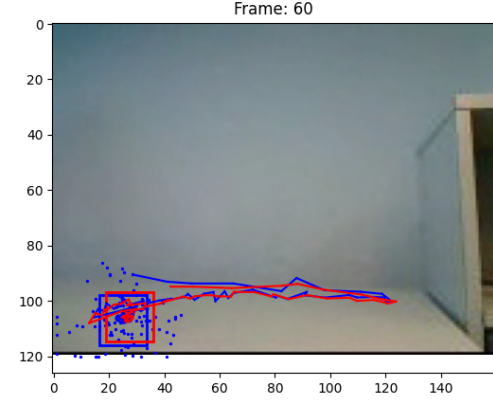


Figure 26: Too-low `sigma_observe`

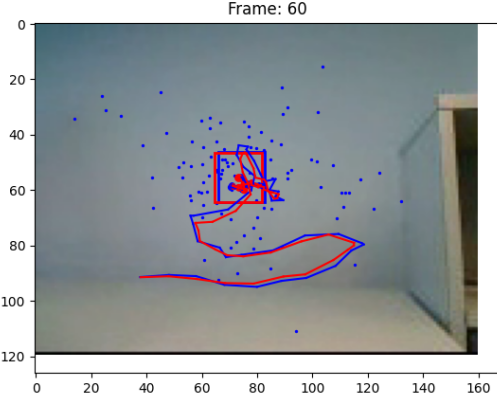


Figure 27: Too-high `sigma_observe`

The **successful** and the **unsuccessful** results allow us to answer the same previous 2 questions (2.2) also for *video₃*.

Speaking generally, the same arguments made for *video₂* apply regarding the advantages and disadvantages of varying `sigma_position`, as well as the potential problems that can occur from choosing values that are too low or too high. In this regard, the effects described in the previous section in decreasing or increasing the value of `sigma_position` are empirically confirmed in the case of *video₃* as well. In particular, a too-low value of `sigma_position` does not allow the model to keep up with the dynamism of the object, resulting in the bounding box being stuck after some frames. This observation is in line with what was stated in the previous section. A too-high value of `sigma_position` causes, in the first stretch of ball motion, the same effect observed in *video₂*, namely a loss of tracking accuracy due to an overly diffused particle cloud. However, when bouncing occurs, the model is unable to cope with the dynamicity of the scene, losing proper tracking and blocking the bounding box due to the dark area located in the lower-right part of the video. Again, theoretically speaking, the same effect observed in the previous section should have been observed even for a too-high value of `sigma_position`. However, the presence of the dark area in the lower-right part of the image prevented the model from carrying out proper tracking of the ball after rebounding.

Regarding measurement noise, the effect obtained in *video₃* for a too-low value of `sigma_observe` confirms the reasoning made in the previous section. In fact, in the absence of occlusions, it can be seen that tracking tends to work well. However, it can be seen that toward the last frames a shift of

the bounding box occurs, this is potentially due to the particle becoming overly sensitive to the color information in each specific frame, which is not reliable in the last frames due to lighting changes in that specific area. The behavior observed in the presence of a too-high value of `sigma_observe` also confirms the reasoning made in the previous section. In particular, an absolute loss of tracking accuracy can be observed, which follows a trajectory completely independent of the object of interest. In this case, as a result of the particles' broader weight distribution, the filter will have difficulties differentiating between particles that closely match the target's color histogram and those that do not. Hence, particles that do not accurately represent the ball's position may be assigned disproportionately high weights.

Finally, the handout asked to answer the following additional questions:

What is the effect of using more or fewer particles?

What is the effect of using more or fewer bins in the histogram color model?

What is the advantage/disadvantage of allowing appearance model updating?

Despite being answered implicitly in all the various previous discussions, I decided to answer the following final questions in an organized manner, also showing the effects obtained on the *video3*.

Below are shown and further discussed the results obtained by varying the following parameters:

- `num_particles`
- `hist_bin`
- `alpha`

num_particles Variations

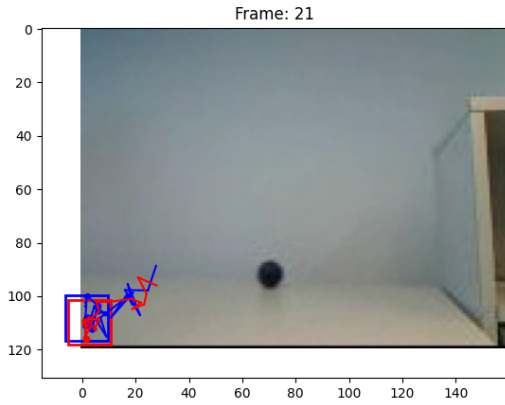


Figure 28: Too-low `num_particles`

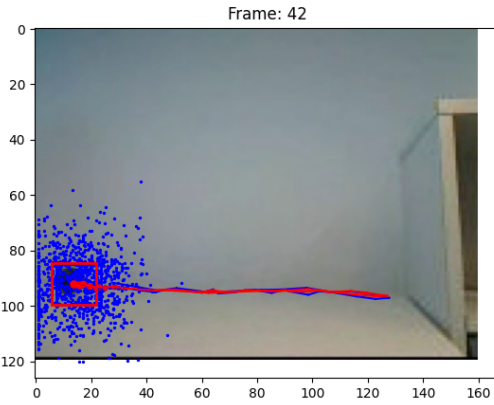


Figure 29: Too-high `num_particles`

hist_bin Variations

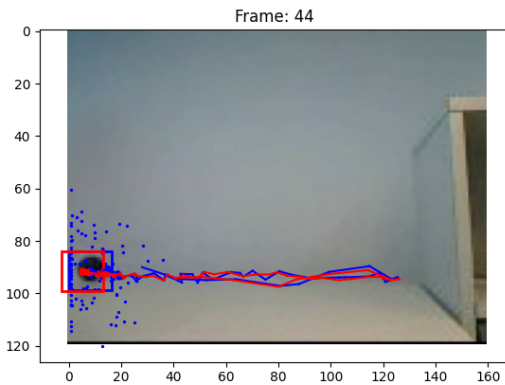


Figure 30: Too-low `hist_bin`

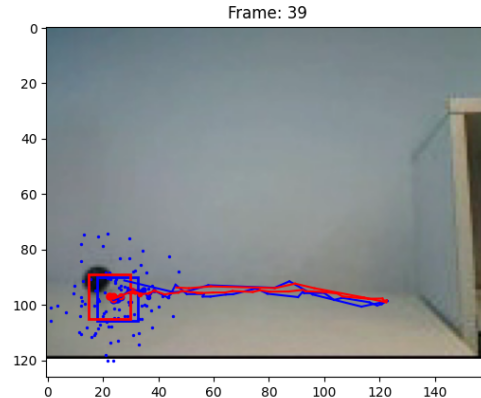


Figure 31: Too-high `hist_bin`

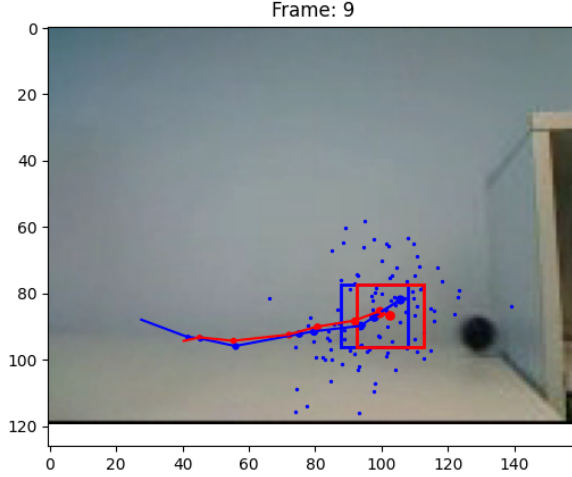


Figure 32: Too-high α

As empirically confirmed by the results shown in figure 29, using more particles results in improved accuracy. This is because a larger number of particles allows for a finer sampling of the state space, increasing the likelihood of accurately capturing the true state of the object. However, this comes at the cost of increased computational load, since more particles require more processing, which can slow down the algorithm. Hence, beyond a certain point, adding more particles may not significantly improve accuracy, but still adds to computational overhead. Conversely, fewer particles can be processed more quickly, but there’s a higher chance of missing the true state, especially in complex scenes, where we might encounter the risk of filter degeneracy, i.e. if too few particles are used, the filter might converge prematurely to an incorrect hypothesis. Figure 28 empirically confirms this reasoning.

Using more bins results in finer color resolution, since it is possible to capture subtle color variations, improving the ability to distinguish the object from the background. Figure 31 shows behavior in line with this argumentation. However, this comes at the cost of increased sensitivity to noise. This is because a higher-resolution histogram might overfit specific color variations that are not representative of the object of interest. Conversely, fewer bins can generalize better over color variations, making the model more robust to changes in lighting or minor appearance changes. This comes at the risk of a coarse color resolution that might impede the ability to differentiate the object from backgrounds or other objects with similar but not identical colors. In fact, as can be seen in figure 30, in the last few frames the bounding box gradually moves toward the shaded area in the lower left of the video, empirically confirming what has just been asserted.

As shown in figure 23, a low α helps to preserve tracking consistency by avoiding reacting to minor changes in the appearance or background of the object. This is advantageous in this scenario since the basic properties of the object remain consistent throughout the video, allowing the tracker to effectively exploit the initial appearance model for accurate tracking. A higher value allows the model to adapt to changes in the object’s appearance. However, the model might react too quickly to non-representative changes, such as occlusions or, in this particular case, lighting fluctuations. Figure 32 shows exactly this issue.