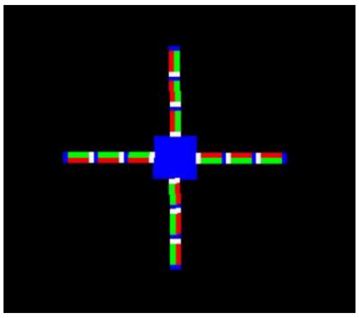
TP3 LANCEREAU FAURE

Object Tracking

on all the displays particles are displayed on the TOP LEFT CORNER OF THE GREEN SQUARE THAT TRACK OUR OBJECTS

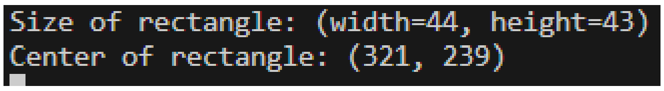
1 Tracking simple objects in videos



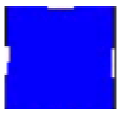
We want to implement a particle filter for tracking object.
We work on a video of a blue square that is moving:

The video is decomposed into different frames on our program. Our algorithm tries to track the square on each frame and return the last frame with the track area and the position of the center of the track area.

First, we define the tracked object on the first frame:



```
Size of rectangle: (width=44, height=43)
Center of rectangle: (321, 239)
```



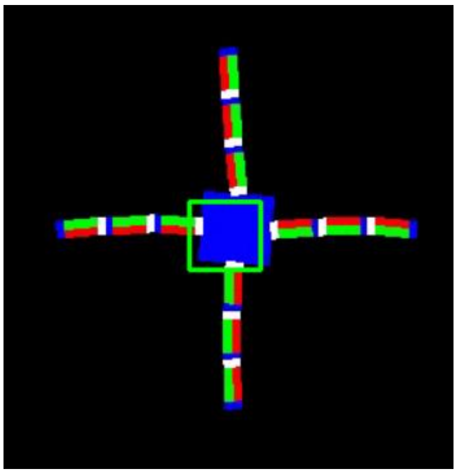
To be more precise the square is 40x40 and start at the center of frames[0]

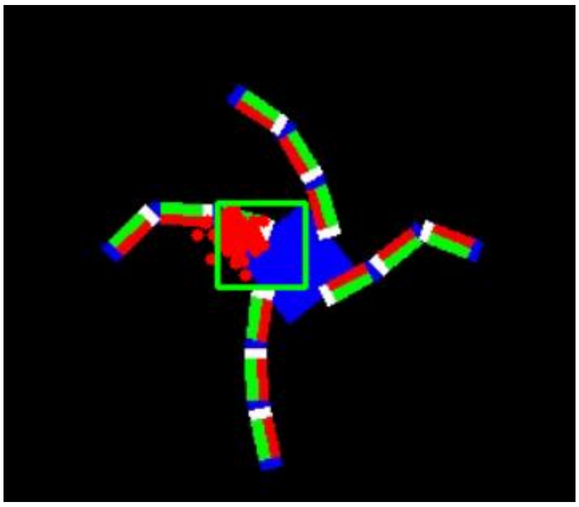
The algorithm uses different step:

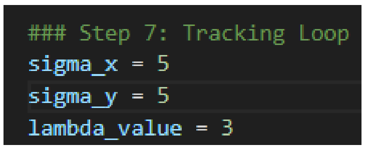
- Prediction of the position of the tracked object on next frame
- Correction: with our N particles we are searching for particles with same or closest Histograms than o the referenced Histograms of the blue squared. Here we compute the Histogram on grayscale (so we have to convert the frame).
- Finally, we resampled the particles to accord more weight to the closest particles and decrease the weight of far particles.

To determine the position of the tracked object we multiply each position of a particle by it weight ad we sum that.

The result after a tracking loop:







```python
### Step 7: Tracking Loop
sigma_x = 5
sigma_y = 5
lambda_value = 3
```

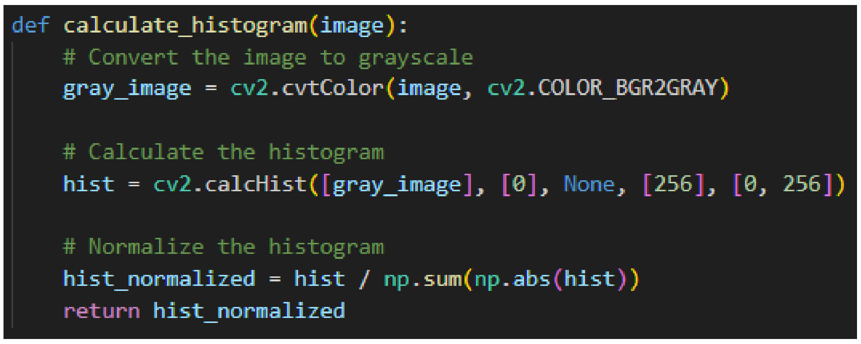
We display each frame during the loop with in red the left top corner of all the particles and in green the tracked square.

We choose default parameters for sigma_x and sigma_y : the covariance matrix and for lambda wich is used in the likelihood function. Moreover, we choose 100 particles

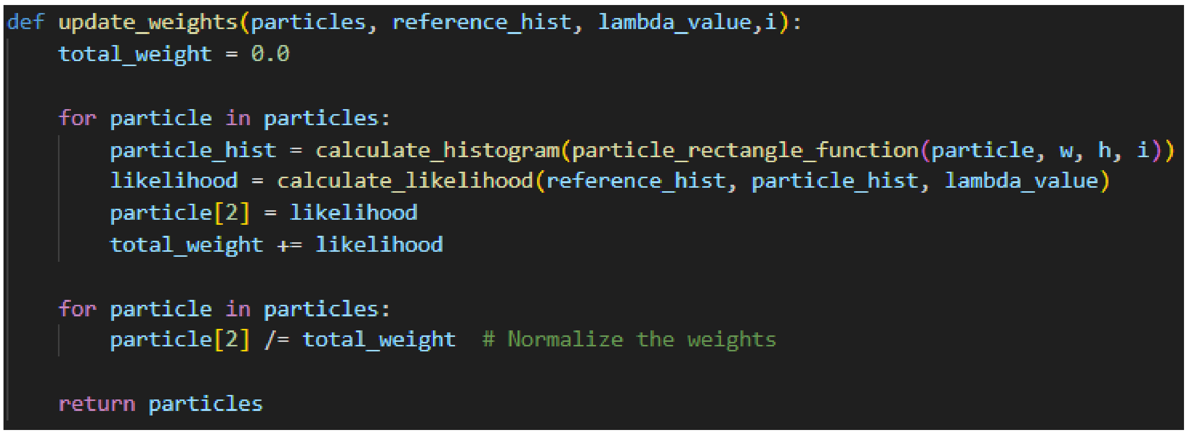
The parameters are set like this because the square move slowly.

We can see here for the last frame that the program is quite accurate.

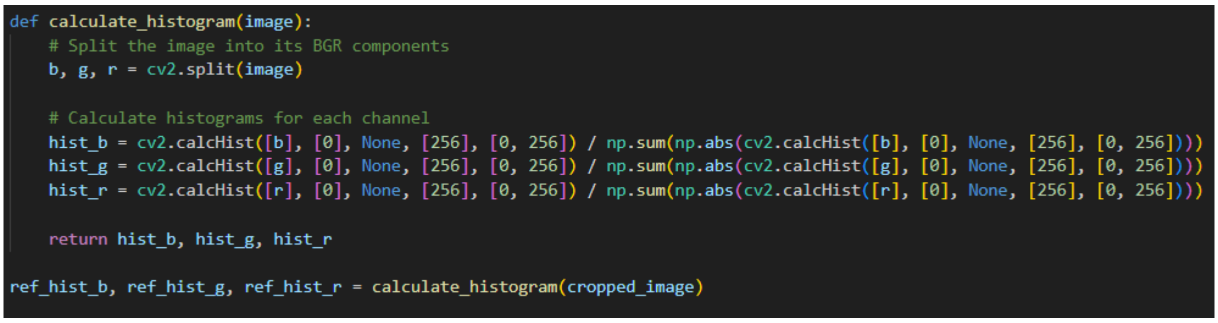
Greyscale histogram



```python
def calculate_histogram(image):
    # Convert the image to grayscale
    gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

    # Calculate the histogram
    hist = cv2.calcHist([gray_image], [0], None, [256], [0, 256])

    # Normalize the histogram
    hist_normalized = hist / np.sum(np.abs(hist))
    return hist_normalized
```

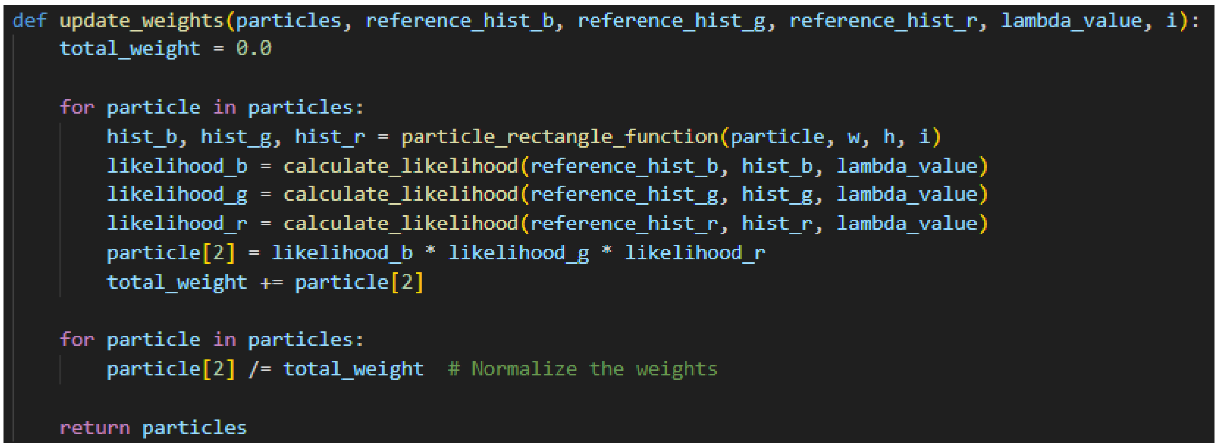


```python
def update_weights(particles, reference_hist, lambda_value,i):
    total_weight = 0.0

    for particle in particles:
        particle_hist = calculate_histogram(particle_rectangle_function(particle, w, h, i))
        likelihood = calculate_likelihood(reference_hist, particle_hist, lambda_value)
        particle[2] = likelihood
        total_weight += likelihood

    for particle in particles:
        particle[2] /= total_weight  # Normalize the weights

    return particles
```



```python
def calculate_histogram(image):
    # Split the image into its BGR components
    b, g, r = cv2.split(image)

    # Calculate histograms for each channel
    hist_b = cv2.calcHist([b], [0], None, [256], [0, 256]) / np.sum(np.abs(cv2.calcHist([b], [0], None, [256], [0, 256])))
    hist_g = cv2.calcHist([g], [0], None, [256], [0, 256]) / np.sum(np.abs(cv2.calcHist([g], [0], None, [256], [0, 256])))
    hist_r = cv2.calcHist([r], [0], None, [256], [0, 256]) / np.sum(np.abs(cv2.calcHist([r], [0], None, [256], [0, 256])))

    return hist_b, hist_g, hist_r

ref_hist_b, ref_hist_g, ref_hist_r = calculate_histogram(cropped_image)
```

Coloured histogram:

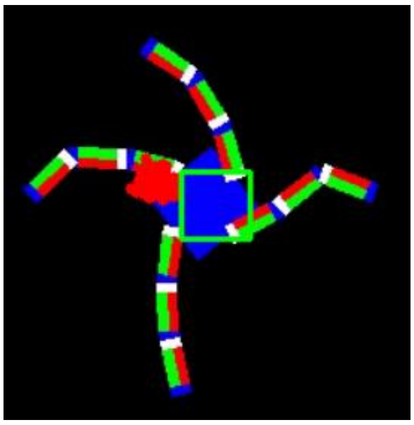


```python
def update_weights(particles, reference_hist_b, reference_hist_g, reference_hist_r, lambda_value, i):
    total_weight = 0.0

    for particle in particles:
        hist_b, hist_g, hist_r = particle_rectangle_function(particle, w, h, i)
        likelihood_b = calculate_likelihood(reference_hist_b, hist_b, lambda_value)
        likelihood_g = calculate_likelihood(reference_hist_g, hist_g, lambda_value)
        likelihood_r = calculate_likelihood(reference_hist_r, hist_r, lambda_value)
        particle[2] = likelihood_b * likelihood_g * likelihood_r
        total_weight += particle[2]

    for particle in particles:
        particle[2] /= total_weight  # Normalize the weights

    return particles
```

For coloured histograms we have to calculate three histograms: one for each color and make the product of the three likelihoods in the update weight function ().

Here the squared is uniformly colorised so colours histograms don't bring more accuracy. However, if we define the tracked area as the square plus a little border with different color then the accuracy is better.

We choose still 100 particles and decrease x_sigma and y_sigma to 4 as with this trick the algorithm tracks better the blue square so we have less to spread particles during the prediction.



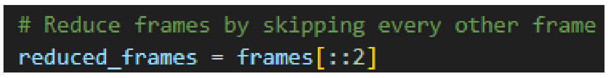
We can now study the different parameters:

- The transition model Σ (transition covariance): A higher value of Σ results in a more significant influence of the transition model, making the predicted positions of particles more spread out. This can lead to smoother but potentially slower tracking. However Higher values of Σ make the tracker more robust to noisy measurements but might lead to over-smoothing, causing the tracker to lag behind fast-moving targets. A lower Σ can result in more abrupt and responsive tracking.

- Likelihood Model λ: Increasing λ makes the likelihood model more sensitive to differences between the reference and observed histograms. This can allow to discriminate between the target and background but may lead to increased sensitivity to noise. A lower λ makes the likelihood model less sensitive, which can improve robustness but might result in inaccurate tracking, especially in challenging scenarios.

- Color Histogram Bins (Nb): Increasing the number of histogram bins (Nb) provides a more detailed representation of color information. This can improve tracking accuracy, especially for objects with subtle color variations. However, a higher Nb also increases computational complexity.

- Number of Particles (N): Increasing the number of particles (N) generally improves tracking accuracy. More particles allow the tracker to explore a larger state space and provide a more accurate representation of the target's position. A higher number of particles can make the tracker more robust to occlusions and changes in appearance but might be computationally expensive. However, a higher number of particles also increase computational load.



```python
# Reduce frames by skipping every other frame
reduced_frames = frames[::2]
```

We want now to reduce the number of frames:

If we want still a performant tracker:

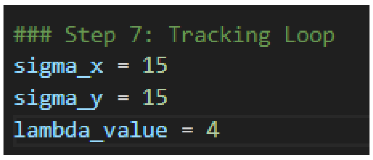
Transition Model (Σ):
If frame skips result in irregular motion updates, consider adjusting the transition model covariance (Σ) to accommodate potential larger jumps between frames.

Likelihood Model (λ):
With fewer frames, there are fewer observations to update the likelihood model. You may need to

adjust the likelihood model sensitivity (λ) to appropriately balance sensitivity to changes in appearance and robustness to noise.

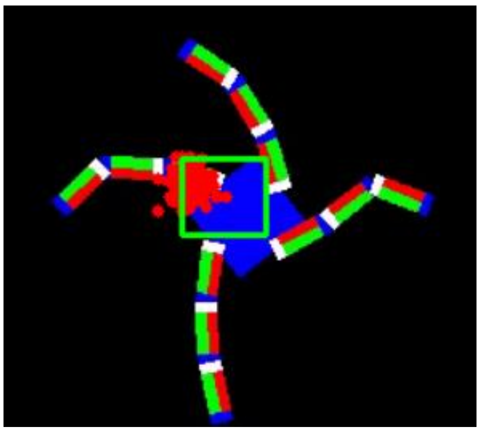
Number of Particles (N):
With a reduced number of frames, you might consider reducing the number of particles (N) to maintain real-time performance.

Histogram Update Frequency:
If the color histogram is used and the object appearance changes rapidly, consider adapting the histogram more frequently to capture changes in color distribution even with fewer observations.



```
### Step 7: Tracking Loop
sigma_x = 15
sigma_y = 15
lambda_value = 4
```

New parameters for a good accuracy:



We can see on the last frame that the tracked area is still accurate with our modified parameters. We had to increase sigma_x and sigma_y to more widespread the particles between two frames and increase particles number to 200.

What about dynamic tracking object size:

- Instead of assuming a fixed size during initialization, initialize the particles with varying sizes that cover a reasonable range of possible object sizes.
- Modify the likelihood model to compare histograms in a way that is robust to changes in size. One approach is to resize the target region to a standard size before computing the histogram.
- Adjust the resampling step to account for changes in object size. You might consider using a resampling algorithm that takes particle weights and sizes into account.

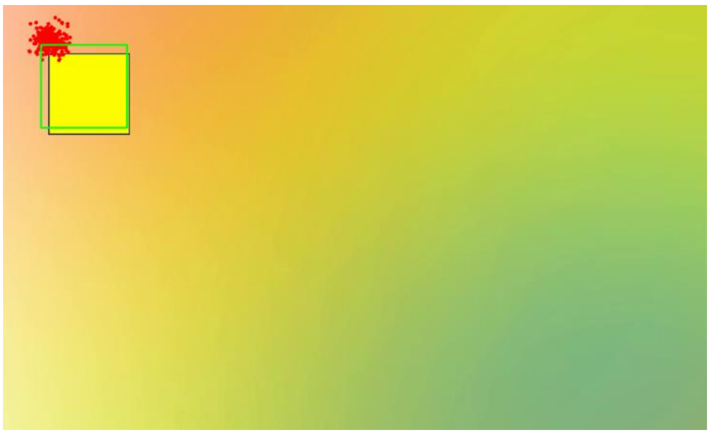
Observations in terms of tracking accuracy:

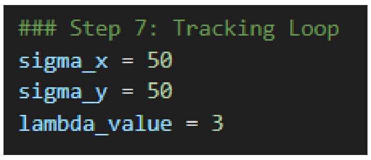
- Robustness to Size Changes: With the adaptation to varying object sizes, the tracker becomes more robust to changes in the size of the tracked object. It should be able to handle scenarios where the object undergoes scale variations.

- Sensitivity to Initialization: The accuracy of the tracker may still depend on the quality of the initial size estimation and the choice of the size range during initialization. A wider size range can increase robustness but may also introduce challenges in maintaining accurate tracking.

- Computational Complexity: The resizing of target regions during the likelihood computation adds computational overhead. Depending on the size of the video frames and the number of particles, this may impact real-time performance.

- It's important to experiment with different size ranges, resizing strategies, and resampling algorithms to find the right balance.





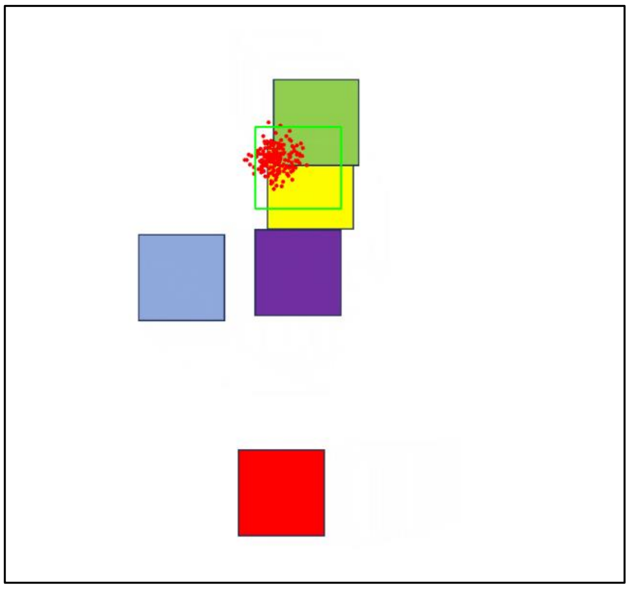
```
### Step 7: Tracking Loop
sigma_x = 50
sigma_y = 50
lambda_value = 3
```

More complex videos:

1) Non uniformed background

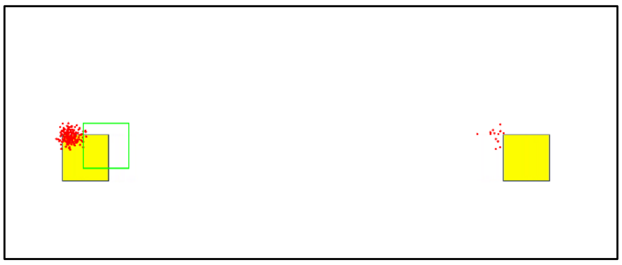
With 300 particles our tracker for this example is still very accurate!

Nice



2) Multi squared (yellow squared tracked)

The accuracy is a bit lower but the green rectangle is moving fast and sometimes overlap other squares. (still 300 particles)



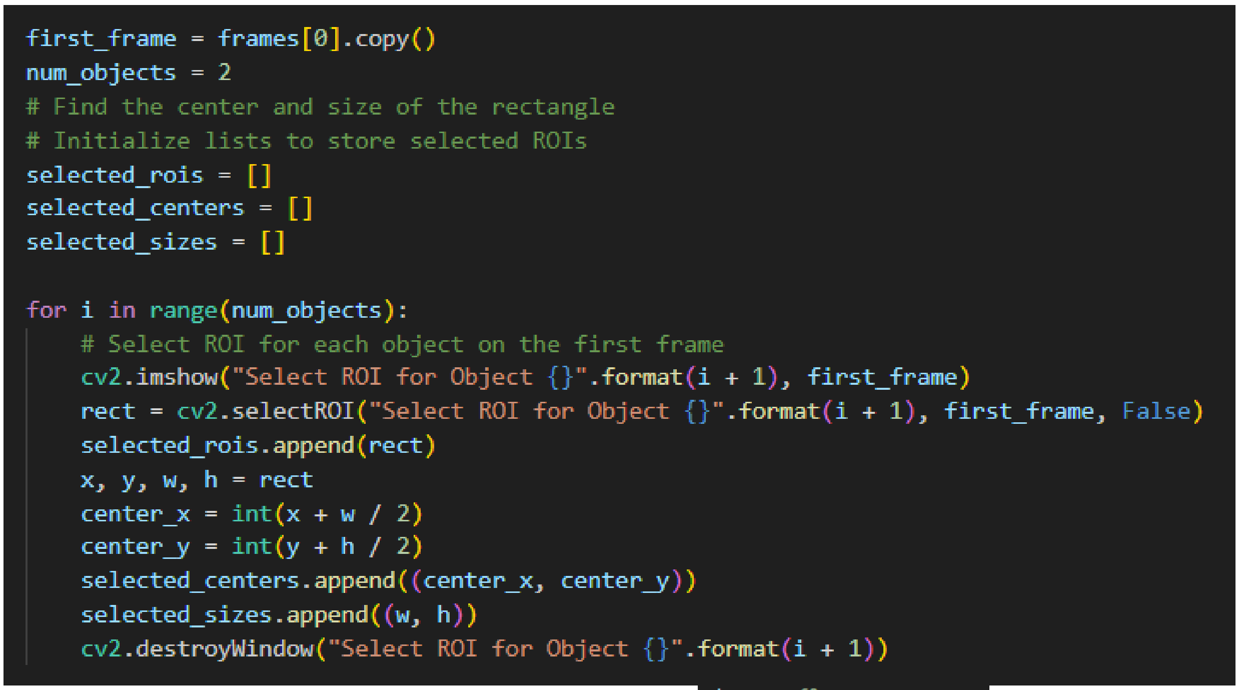
3) An other one

Two same square that are crossing theirselves. The model is fighting to maintain the tracking object on the good square but succeed it even if some particles are lost on the wrong squared

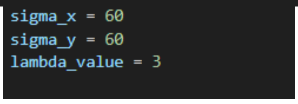
parameters: 300 particles, x_sigma and y_sigma = 60, lambda value = 3

2 Tracking complex objects in videos

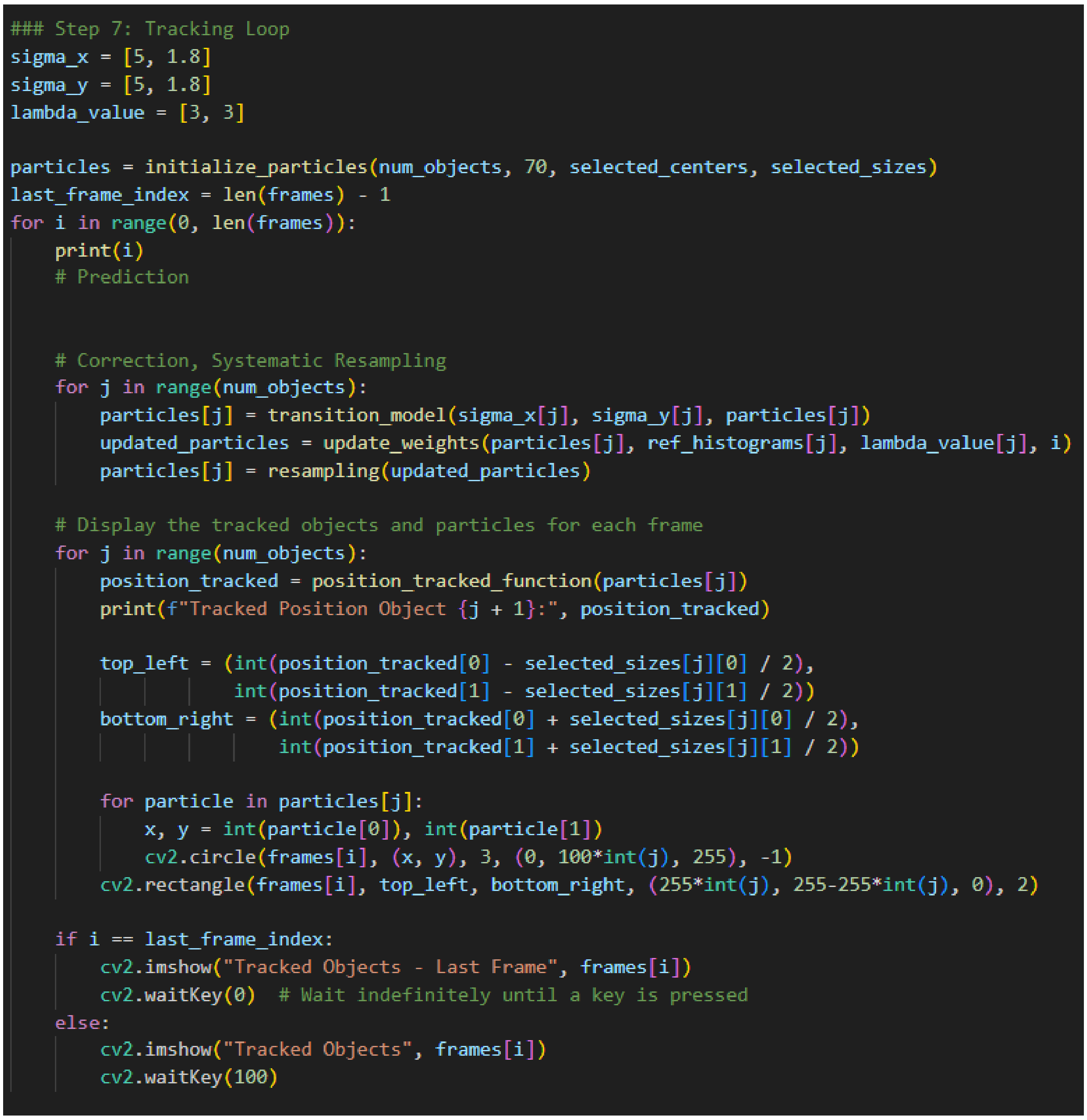
Now we track several objects at the same time. It means that we define x_sigma, y_sigma and lambda value for each object. We have also to define on the first frame several objects tracked.



```
first_frame = frames[0].copy()
num_objects = 2
# Find the center and size of the rectangle
# Initialize lists to store selected ROIs
selected_rois = []
selected_centers = []
selected_sizes = []

for i in range(num_objects):
    # Select ROI for each object on the first frame
    cv2.imshow("Select ROI for Object {}".format(i + 1), first_frame)
    rect = cv2.selectROI("Select ROI for Object {}".format(i + 1), first_frame, False)
    selected_rois.append(rect)
    x, y, w, h = rect
    center_x = int(x + w / 2)
    center_y = int(y + h / 2)
    selected_centers.append((center_x, center_y))
    selected_sizes.append((w, h))
    cv2.destroyWindow("Select ROI for Object {}".format(i + 1))
```



```
sigma_x = 60
sigma_y = 60
lambda_value = 3
```



```python
### Step 7: Tracking Loop
sigma_x = [5, 1.8]
sigma_y = [5, 1.8]
lambda_value = [3, 3]

particles = initialize_particles(num_objects, 70, selected_centers, selected_sizes)
last_frame_index = len(frames) - 1
for i in range(0, len(frames)):
    print(i)
    # Prediction


    # Correction, Systematic Resampling
    for j in range(num_objects):
        particles[j] = transition_model(sigma_x[j], sigma_y[j], particles[j])
        updated_particles = update_weights(particles[j], ref_histograms[j], lambda_value[j], i)
        particles[j] = resampling(updated_particles)

    # Display the tracked objects and particles for each frame
    for j in range(num_objects):
        position_tracked = position_tracked_function(particles[j])
        print(f"Tracked Position Object {j + 1}:", position_tracked)

        top_left = (int(position_tracked[0] - selected_sizes[j][0] / 2),
                    int(position_tracked[1] - selected_sizes[j][1] / 2))
        bottom_right = (int(position_tracked[0] + selected_sizes[j][0] / 2),
                        int(position_tracked[1] + selected_sizes[j][1] / 2))

        for particle in particles[j]:
            x, y = int(particle[0]), int(particle[1])
            cv2.circle(frames[i], (x, y), 3, (0, 100*int(j), 255), -1)
        cv2.rectangle(frames[i], top_left, bottom_right, (255*int(j), 255-255*int(j), 0), 2)

    if i == last_frame_index:
        cv2.imshow("Tracked Objects - Last Frame", frames[i])
        cv2.waitKey(0)  # Wait indefinitely until a key is pressed
    else:
        cv2.imshow("Tracked Objects", frames[i])
        cv2.waitKey(100)
```