

Problem Set 5: Object Tracking and Pedestrian Detection

October 11, 2025

ASSIGNMENT DESCRIPTION

Description

In Problem Set 5, you are going to implement tracking methods for image sequences and videos. The main algorithms you will be using are the Kalman and Particle Filters. Methods to be used: You will design and implement Kalman and Particle Filters from the ground up.

RULES: You may use image processing functions to find color channels, load images, find edges (such as with Canny), and resize images. Don't forget that those have a variety of parameters, and you may need to experiment with them. Refer to this problem set's FAQs.md for a list of banned function calls.

Please do not use absolute paths in your submission code. All paths should be relative to the submission directory. Any submissions with absolute paths are in danger of receiving a penalty!

Learning Objectives

- Identify which image processing methods work best to locate an object in a scene.
- Learn to how object tracking in images works.
- Explore different methods to build a tracking algorithm that relies on measurements and a prior state.
- Create methods that can track an object when occlusions are present in the scene.

Starter Code

Obtain the starter code from this assignment's repository in GitHub.

INSTRUCTIONS

Programming Instructions

Your main programming task is to complete the functions described in the file **PS5.py**. The driver program **experiment.py** helps you illustrate the results and will output the files needed for the writeup.

Write-up Instructions

Create **ps5_report.pdf** - a PDF file that shows all your output for the problem set, including images labeled appropriately (by filename, e.g. ps5-1-a-1.png) so it is clear which section they are for and the small number of written responses necessary to answer some of the questions (as indicated). You are required to do the following when creating your report:

- Use the LATEX template provided.
- PLEASE be sure to add your name and email, exactly as it is on CANVAS, as we need to know who did this work. ALSO, please change the term to the current semester.

How to Submit

Two assignments have been created on Gradescope: one for the report - **PS5_report**, and the other for the code - **PS5_code**.

- Report: the report (PDF only) must be submitted to the PS5_report assignment.
- Code: all files must be submitted to the PS5_code assignment. DO NOT upload zipped folders or any sub-folders, please upload each file individually. Drag and drop all files into Gradescope.

Note:

You only need to submit ps5.py and experiment.py to Gradescope. Any functions added or modified in ps5_utils.py will not be checked by gradescope and you may get an error if your implementation depends on them. However, you can still add any code in the ps5_utils.py for local use/saving output etc.

Notes

- You can only submit to the autograder **10** times in an hour. You'll receive a message like "You have exceeded the number of submissions in the last hour. Please wait for 36.0 mins before you submit again." when you exceed those 10 submissions. You'll also receive a message "You can submit 8 times in the next 53.0 mins" with each submission so that you may keep track of your submissions.
- If you wish to modify the autograder functions, create a copy of those functions and DO NOT mess with the original function call.

YOU MUST SUBMIT your report and code separately, i.e., two submissions for the code and the report, respectively. Only your last submission before the deadline will be counted for each of the code and the report.

Grading

The assignment will be graded out of 100 points.

ASSIGNMENT QUESTIONS

1 THE KALMAN FILTER

The Kalman Filter (KF) is a method used to track linear dynamical systems with Gaussian noise. Recall that both the predicted and the corrected states are Gaussians, allowing us to keep the mean and covariance of the state to run this filter. Use the Kalman filter on the provided images to track a moving circle.

First, we need to define the KF's components. We will use the notation presented in the lectures for the N-dimensional case.

State vector:

We represent the filter's state using the current position and velocities in the x, y plane. We will not use acceleration in our exercise.

$$X = [x, y, v_x, v_y]$$

Dynamics Model transition 4x4 matrix D_t :

This matrix will map the equations below to the state components. Determine how this 4x4 matrix can be used to represent these equations using the state vector.

$$x_t = x_{t-1} + v_x \Delta t$$

$$y_t = y_{t-1} + v_y \Delta t$$

$\Delta t = 1$ for simplicity

State and Covariance prediction:

We assume that the state vector prediction is based on a linear transformation using the transition matrix D_t . The covariance is obtained from the squaring operation plus the process noise Σ_{d_t} .

We will also assume this noise is independent from each component in the state vector.

$$\begin{aligned} X_t^- &= D_t X_{t-1}^+ \\ \Sigma_t^- &= D_t \Sigma_{t-1}^+ D_t^T + \Sigma_{d_t} \end{aligned}$$

$$\Sigma_{d_t} = \begin{bmatrix} \sigma_{d_x}^2 & 0 & 0 & 0 \\ 0 & \sigma_{d_y}^2 & 0 & 0 \\ 0 & 0 & \sigma_{d_{v_x}}^2 & 0 \\ 0 & 0 & 0 & \sigma_{d_{v_y}}^2 \end{bmatrix}$$

Before we continue to the Correction state, we will define our sensor's function as a way to obtain the object's position. In our case, we will use a template-matching code to obtain these components. In order to make this a little more interesting, we will add some noise to these measurements.

Sensor measurement 2x4 matrix M_t :

This matrix maps the state vector defined above to the available measurements. Because we can only obtain two values, x and y, it contains two rows and four columns.

Kalman Gain K_t :

Using the measurement matrix and the predicted covariance, we can now compute the Kalman Gain. Here you will also define a measurement noise represented by Σ_{m_t} .

$$K_t = \Sigma_t^- M_t^T (M_t \Sigma_t^- M_t^T + \Sigma_{m_t})^{-1}$$

State and Covariance correction:

Now that we have all the components we need, you will proceed to update the state and covariance arrays. Notice that to update the state, you will use the residual represented by the difference between the measurements obtained from the sensor (template matching function) Y_t and the predicted positions $M_t X_t^-$.

$$X_t^+ = X_t^- + K_t(Y_t - M_t X_t^-)$$
, where Y_t is the measurements at time t

$$\Sigma_t^+ = (I - K_t M_t) \Sigma_t^-$$
, where I is the identity matrix

In the next iteration, the updated state and covariance will be used in the prediction stage.

This may seem like a lot, but you will find coding this KF is not complicated. Complete the KalmanFilter class with the process described above. We will split the tasks the following way:

a.

Initialize the class defining the class arrays you will be using. Place these components in the `_init_(self, init_x, init_y)` function.

Here's a reminder of what should be included in this part:

- State vector (X) with the initial x, y, v_x and v_y values.
- Covariance 4x4 array (Σ^-) initialized with a diagonal matrix with some value.
- 4x4 state transition matrix D_t
- 2x4 measurement matrix M_t
- 4x4 process noise matrix Σ_{d_t}
- 2x2 measurement noise matrix Σ_{m_t}

Code: `__init__(self, init_x, init_y)`

Complete the prediction state in `predict(self)`. Here you will replace the class variables for the state and covariance arrays with the prediction process.

Code: `predict(self)`

Finally, we need to correct the state and the covariances using the Kalman gain and the measurements obtained from our sensor.

Code: `correct(self, meas_x, meas_y)`

b.

Position estimation is performed by the function `process(self, measurement_x, measurement_y)`, which uses the functions defined above. Now that you have finished creating the Kalman Filter, estimate the position of a bouncing circle. Use the images in the directory called 'circle' to find

the circle's position at each frame.

Input: circle/0000.jpg to circle/0099.jpg.

Code: part_1b()

c.

Now let's try with a more complicated scene using the same filter. We will find and track pedestrians in a video. This means we need to update our sensor with a function that can locate people in an image. Fortunately, there is a function in OpenCV that can help us with this task using Histogram of Gradients (HoG) descriptors. This object contains a method called detectMultiScale that returns the bounding boxes for each person in the scene along with the detection weight associated to it.

Use the sequence of images in the ‘walking’ directory to detect and follow the man crossing the street using the output from the HoG detector as the measurements and the Kalman Filter.

Note: You are not expected to tune values for R, Q, and Noise to improve the performance of your Kalman Filter, as you should pass all Kalman Filter tests in the autograder without their modification.

Input directory: walking/001.jpg to walking/159.jpg

Code: part_1c()

2 THE PARTICLE FILTER

Recall that we need a variety of elements:

1. a model - this is the “thing” that is actually being tracked. Maybe it’s a patch, a contour, or some other description of an entity;
 2. a representation of state x_t that describes the state of the model at time t;
 3. a dynamics model $p(x_t|x_{t-1})$ that describes the distribution of the state at time t given the state at t-1;
 4. a measurement z_t that somehow captures the data of the current image; and finally,
 5. a sensor model $p(z_t|x_t)$ that gives the likelihood of a measurement given the state.
- For Bayesian-based tracking, we assume that at time t-1 we have a belief about the state represented as a density $p(x_{t-1})$, and that given some measurements z_t at time t, we update our belief by computing the posterior density:
- $$Bel(x_t) \propto p(z_t|x_t)p(x_t|u_t, x_{t-1})Bel(x_{t-1})$$

In class, we discussed both the theory and the implementation specifics behind particle filtering. The algorithm sketch is provided in the slides and describes a single update of the particle filter. What is left up to the coder are several details, including what the model is, what the state is, the number of particles, the dynamics/control function $p(x_t|x_{t-1}, u_t)$, the measurement function $p(z_t|x_t)$, and the initial distribution of particles.

For this question, you will need to track an image patch template taken from the first frame of the video. For this assignment, the model is simply going to be the image patch, and the state will be only the 2D center location of the patch. Thus, each particle will be a (u, v) pixel location

(where u and v are the column and row number, respectively) representing a proposed location for the center of the template window.

We will be using a basic function to estimate the dissimilarity between the image patch and a window of equal size in the current image, the Mean Squared Error:

$$MSE(u_p, v_p) = \frac{1}{mn} \sum_{u=1}^m \sum_{v=1}^n (Template(u, v) - Image^t(u + u_p - \frac{m}{2}, v + v_p - \frac{n}{2}))^2$$

The funny indexing is just because (u, v) are indexed from 1 to M (or N), but the state (u_p, v_p) is the location of the center. Notice that the above equation can be simplified by using array operations. Remember, you are calculating the Mean of the Squared Error between two images, a template, and a patch of the image.

These images are color, but you can do the tracking in the full color image, in grayscale, or even in the green channel. If you use color, there would be an outer summation over the three color channels. If you want to use grayscale, you can use cvtColor in OpenCV or just use a weighted sum of R,G,B to get a luma value (try weights of 0.3, 0.58, and 0.12).

MSE, of course, only indicates how dissimilar the image patch is, whereas we need a similarity measure so that more similar patches are more likely. Thus, we will use a squared exponential equation (Gaussian) to convert this into a usable measurement function:

$$p(z_t|x_t) \propto \exp\left(\frac{-MSE}{2\sigma_{MSE}^2}\right)$$

To start out, you might use a value of $\sigma_{MSE} = 10$ (for grayscale in 0-255 range), but you might need to change this value.

For the dynamics, we're going to use normally distributed noise since the object movements can be unpredictable, and often current velocity isn't indicative of future motion; so our dynamics model is merely

$$x_t = x_{t-1} + \delta_t, \text{ where } \delta_t \text{ is } N(0, \Sigma_d), \Sigma_d = \begin{bmatrix} \sigma_d^2 & 0 \\ 0 & \sigma_d^2 \end{bmatrix}$$

which is just a fancy way to say that you add a little bit of Gaussian noise to both u and v independently.

The number of particles and initial distribution are up to you to figure out. You may need to tune your parameters for these to get better tracking/performance.

To visualize the tracker's behavior, you will need to overlay each successive frame with the following elements:

- Every particle's (x,y) location in the distribution should be plotted by drawing a colored dot point on the image. Remember that this should be the center of the window, not the corner.
- Draw the rectangle of the tracking window associated with the Bayesian estimate for the current location, which is simply the weighted mean of the (x,y) of the particles.
- Finally, we need to get some sense of the standard deviation or spread of the distribution. First, find the distance of every particle to the weighted mean. Next, take the weighted sum of these distances and plot a circle centered at the weighted mean with this radius.

You are encouraged to produce these visuals for selected frames. You will not have to submit the entire video.

For reading the frames OpenCV users should look at the class VideoCapture. For sampling based on a distribution, see the functions numpy.random.multinomial or numpy.random.choice.

a.

Implement the particle filter. You can find the bounding box for the initial position in template_rect - this is your template. Tweak the parameters, including window size, until you can get the tracker to follow this object. Run the tracker and save the video frames described below with the visualizations overlaid.

Classes, methods: Define a class ParticleFilter, with the following methods:

- `_init_(frame, template, **kwargs)` : (constructor) Initialize particle filter object.
- `get_error_metric(template, frame_cutout)` : Calculates the error metric used (i.e. MSE).
- `resample_particles()` : Returns a list of particles resampled based on their weights.
- `process(frame)` : Process a frame (image) of video and update filter state.
* Hint: You should call `resample_particles()` and `get_error_metrics()` here.
- `render(frame_in)` : Visualize current particle filter state.

Finally, modify the variables `num_particles`, `sigma_mse`, `sigma_dyn` in the `part_2a()` function to appropriate values.

A helper function, `run_particle_filter()`, has been provided to run your particle filter on a video. You can also display every video frame to verify that your particle filter is working properly.

Note: The helper function `run_particle_filter()` can save the template image patch and outputs generated by your `render()` method for desired frames. See template code for details. To clarify, in `process()` you will:

- Add noise to the existing particles based on `sigma_dyn`
- Update the particle weights based on the `get_error_metric()` between the template and the current particle positions (cutout of image)
- Use the updated weights to resample the particles using the `resample_particles` and set them as the new particles.

Input directory: circle/0000.jpg to circle/0099.jpg

Code: ParticleFilter() and part_2a() in ps5.py

b.

Now use the particle filter with a noisy video sequence of images in the 'pres_debate_noisy' directory.

Note: If you encounter a "float division by zero" error for parts 2a and 2b when submitting to the autograder. You'll need to tune some of the initial parameters in the `part_2a` functions at the bottom of ps5.py first for the autograder to reference.

Input directory: pres_debate_noisy/000.jpg to pres_debate_noisy/099.jpg

Code: part_2b() in ps5.py

3 CHANGES IN APPEARANCE

In the last section, you were working with a static template, assuming that the target would not change in shape. This can be addressed by updating the model's appearance over time.

Modify your existing tracker to include a step that uses the history to update the tracking window model. We can accomplish this using an Infinite Impulse Response (IIR) filter. The concept is simple: we first find the best tracking window for the current particle distribution as displayed in the visualizations. Then, we update the current window model to be a weighted sum of the last model and the current best estimate.

$$Template(t) = \alpha Best(t) + (1 - \alpha) Template(t - 1)$$

where $Best(t)$ is the patch of the best estimate or mean estimate. It's easy to see that by recursively updating this sum, the window implements an exponentially decaying weighted sum of (all) the past windows. For this assignment, you may assume t to be the frame number, in lieu of a time measure.

a.

Implement the appearance model update feature. Run the tracker on the presidential debate images and adjust parameters until you can track the person's hand (the one not holding the microphone). Your tracker's bounding box should only contain the hand at all times. Run the tracker. You can save the video frames with the visualizations overlaid for reference.

Classes, methods: Derive a class `AppearanceModelPF` from `ParticleFilter`, retaining the same interface - i.e., with methods `process()` and `render()`, as defined above. The constructor and `process()` method should transparently handle model updates. You can supply additional keyword arguments to the constructor, if needed.

Finally, modify the variables `num_particles`, `sigma_mse`, `sigma_dyn` in the `part_3()` function to appropriate values.

Input directory: `pres_debate/000.jpg` to `pres_debate/165.jpg`

Code: `AppearanceModelPF()` and `part_3()` in `ps5.py`

4 PARTICLE FILTERS AND OCCLUSIONS

For this part, we will work with a much more difficult video to perform tracking with, `pedestrians.mp4`. We'd like to be able to track the blond-haired woman (the one with the white jacket) as she crosses the road. If you try applying your adaptive tracker to this video, you will probably find that you will have difficulty dealing simultaneously with occlusion and the perspective shift as the woman walks away from the camera. Thus, we need some way of relying less on updating our appearance model from previous frames and more on a sophisticated model based on the dynamics of the figure we want to track.

Expand your appearance model to include window size as another parameter. This will change the representation of your particles. You are highly recommended to use `cv2.resize` for your

implementation to resize the template to track.

a.

Run the tracker and save the video frames described below with the visualizations overlaid. You will receive full credit if you can reliably track (illustrate this with the rectangle outline as well as your *particles*) all the way to the end of the street and deal gracefully with the occlusions (reasonable tracking at frame 300). The tracking box must track the person's position and size. The bounding box is expected to contain more than 50% of the tracked person and should be within 0.5 to 2 times the size of the person.

Classes, methods: Derive a class MDParticleFilter from AppearanceModelPF, retaining the same interface - i.e., with methods process() and render(), as defined above. The constructor and process() method should transparently handle model updates in appearance and dynamics. You can supply additional keyword arguments to the constructor and other methods, if needed.

Finally, modify the variables *num_particles*, *sigma_mse*, *sigma_dyn* in the *part_4()* function to appropriate values.

Code: MDParticleFilter() and part_4() in ps5.py

Report: Frames to record 40, 100, 240, 300.

-Input: pedestrians/000.jpg to pedestrians/319.jpg.

-Output: ps5-4-a-1.png, ps5-4-a-2.png, ps5-4-a-3.png, and ps5-4-a-4.png

-Text Answer: Describe what you did. How did you modify the Particle Filter class to continue tracking after occlusions?

5 TRACKING MULTIPLE TARGETS

Now use either a Kalman or Particle filter to track multiple targets as they move through the given video. Use the sequence of images in the TUD-Campus directory to detect the people shown below. The bounding box is expected to contain more than 50% of the tracked person and should be within 0.5 to 2 times the size of the person.



Input directory: TUD-Campus/01.jpg to TUD-Campus/71.jpg

Code: part_5() in experiment.py

Report:

- Frames to record: 29, 56, 71
- Output: ps5-5-a-1.png, ps5-5-a-2.png, ps5-5-a-3.png
- Text Answer: Describe what you did. How different it was to use a KF vs PF? Which one worked best and why? Include details about any modifications you had to apply to handle multiple targets.

6 DETECT PEDESTRIANS FROM A MOVING CAMERA

Finally, detect pedestrians from the given video of a moving camera. Use the images in the directory 'follow' to track the man with the hat holding a white plastic bag. Similar to earlier tasks, the bounding box is expected to contain more than 50% of the tracked person and should be within 0.5 to 2 times the size of the person. Remember to visualize your *particles* as well.

Input directory: follow/001.jpg to follow/186.jpg

Code: part_6() in experiment.py

Report:

- Frames to record: 60, 160, 186
- Output: ps5-6-a-1.png, ps5-6-a-2.png, ps5-6-a-3.png
- Text Answer: Describe what you did. Did this task present any additional challenges compared to the previous sections? Include details about any modifications you had to apply.