ECSE 4963 Final Project

Aidan Westphal

December 2023

Purpose

Tracking is an important task in Computer Vision with applications in robotics, surveillance, the military, and more. Several robust algorithms exist for this task, such as Kalman and Particle Filtering. This paper will implement a Particle Filtering algorithm and compare its performance with Kalman Filtering.

Theory

Before discussing Particle Filtering, I feel the need to discuss Kalman Filtering as a building block. The following is a quick explanation of Kalman Filtering.

Kalman Filtering is a probabilistic method of feature tracking that combines two probability distributions: a prediction based on previous locations of the object and a measurement with error bounds, to determine the object's location. It is an iterative process in which, as it progresses, the variance of its prediction shrinks.

Kalman Filtering makes the following assumptions: States change linearly and uncertainty follows a Gaussian distribution. We define a tracking point as $p = (c_t, r_t)$ at time t (or frame at time t) and its velocity $v_p = (v_{ct}, v_{rt})$. We then define a state and the following state as follows:

$$s_t = \begin{bmatrix} c_t \\ r_t \\ v_{ct} \\ v_{rt} \end{bmatrix} \text{ and } s_{t+1} = \begin{bmatrix} c_t + v_{ct} \\ r_t + v_{rt} \\ v_{ct} \\ v_{rt} \end{bmatrix}$$

Note how we use the assumption that states transition linearly. We can better encode this transition by using a state transition matrix, where $s_t = \Phi s_{t-1} + w_t$. Note w_t represents system pertubation (error source) and is normally distributed as $w_t \sim N(0, Q)$ where Q is a covariance matrix. This

leads to the following probability distribution: $p(s_t|s_{t-1}) = N(\Phi s_{t-1}, Q)$.

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

We now develop a measurement model, where $z_t = (z_{ct}, z_{rt})$ is a measured target position at the time (frame) t. We note that $z_t = Hs_t + \epsilon_t$, which the matrix H pulls the row and column values from s_t and defines error ϵ_t between the predicted state and the measured state. We let $\epsilon_t \sim N(0, R)$ where R is measurement uncertainty and can define our measurement model as: $p(z_t|s_t) = N(Hs_t, R)$.

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

So to wrap up terminology, s_t is a state which contains the row and column of a tracking point along with its velocity. z_t is a measurement at time instance t. We can predict the next state $s_{t+1} = \Phi s_t + w_{t-1}$ which resembles the following distribution: $p(s_t|s_{t-1}) = N(\Phi s_{t-1}, Q)$. Similarly, we can take a measurement value and note $z_t = Hs_t + \epsilon_t$ where we describe the error between our measurement value and our actual state s_t, which resembles the following distribution: $p(z_t|s_t) = N(Hs_t, R)$. Thus, our end goal is to find the distribution of our new state s_t given all previous (and current) measurements, i.e. $p(s_t|z_{1:t})$.

Since Kalman Filtering is an iterative process, also noting that we never precisely know a value s_t due to uncertainty, we are given $p(s_{t-1}|z_{1:t-1}) = N(U_{t-1}, \Sigma_{t-1})$, the distribution of the previous state given all previous measurements. We can split $p(s_t|z_{1:t})$ as follows:

$$p(s_t|z_{1:t}) \propto p(s_t|z_{1:t-1})p(z_t|s_t)$$

Where we have a term for prediction based on previous measurements and a term for measurement given the current state. We start with $p(s_t|z_{1:t-1}) = N(U_t^-, \Sigma_t^-)$, which can be found to equal the following after some derivation (involves integrating the product of probability distributions):

$$U_t^- = \Phi U_{t-1} \; \Sigma_t^- = \Phi \Sigma_{t-1} \Phi^T + Q$$

For notation, this is known as the "Temporal Prior of s_t." We then take our measurement distribution term, $p(z_t|s_t)$, known as the "Likelihood of s_t". We have already calculated this above, as $N(Hs_t, R)$. We now note the proportion defined earlier and note $p(s_t|z_{1:t}) = N(U_t^-, \Sigma_t^-)N(Hs_t, R)$. After more derivation...

$$U_t = U_t^- + K_t(z_t - HU_t^-)\ \Sigma_t = (I - K_tH)\Sigma_t^-$$

where $K_t = \arg\min trace(\Sigma_t) = H^T(H\Sigma_t^-H^T+R)^{-1}$ represents a weight/gain matrix that measures the relative contribution between the prediction and the measurement components and is obtained by minimizing the uncertainty of the final prediction.

As such, Kalman Filtering is often implemented by a state-to-state transition system using the above equations to calculate the next mean and variance. Oftentimes the algorithm is quick, especially as the variance of the prediction shrinks as iterations continue. However, Kalman Filtering is limited by the following assumptions:

1. Linear state transitions
2. Uncertainty (System / Measurement) follows Gaussian distribution
3. Returns a Gaussian (Unimodal) distribution

We now consider Particle Filtering, which eliminates these assumptions. Particle filtering makes no assumptions about state transition (necessarily, I will explain), system and measurement models can follow any distribution as long as it's defined, and returns a distribution that has no constraints on model/modality.

Particle Filtering works by casting a large number of "particles" uniformly on an image, measuring in the vicinity of each particle to assign a weight (based on how far it is from the measurement usually), and redistributing the particles based on these weights.

More specifically, Particle Filtering works off of a probability distribution function, described by discretely-weighted particles whose weights sum to one. With each new frame, this distribution changes. The distribution is sampled with some noise, providing a new set of particles. Each particle is

then weighted by its distance from a measurement taken in its neighborhood trying to find a matching feature to the primary target. Recognize that particles farther from these features, or modes, are weighted less for the next distribution, and as such less sampled. The result is a grouping of particles into "clouds" around strong modes.

Notice how this differs from the assumptions made by Kalman Filtering. First off, Particle Filtering returns a probability distribution based on discrete particles, which can have several modes. System and measurement uncertainty is still defined, but is not constricted to a Gaussian distribution (system noise is important for redistribution and measurement noise for weighting). Finally, state transitions doesn't have to be defined necessarily, however some implementations may redistribute particles in a way that accounts for state transition. We will now derive the algorithm.

Getting rid of linear state transitions implies we no longer need to track velocity, thus we simplify states to $s_t = [c_t \ r_t]^T$. Similar to Kalman Filtering, we want to define $P(s_t|z_{1:t})$, the distribution of states at time t given all previous measurements. We similarly decompose this into a temporal prior and likelyhood term:

$$P(s_t|z_{1:t}) = kP(s_t|z_{1:t-1})P(z_t|s_t)$$

Note we added a term k which serves to normalize weights such that our distribution integrates (or discretely sums) to 1. We then find that the temporal prior can be approximated discretely as follows:

$$P(s_t|z_{1:t-1}) = \int P(s_t|s_{t-1})P(s_{t-1}|z_{1:t-1})ds_{t-1} \approx \frac{1}{N}\sum_{i=1}^{N} P(s_t|s'_{t-1,i})$$

where each $s'_{t-1,i}$ refers to a sampled particle i from the distribution $P(s_{t-1}|z_{1:t-1})$. Recognize this is approximating by splitting s_{t-1} using the distribution $P(s_{t-1}|z_{1:t-1}$ (note that this product is roughly zero for points poorly weighted from this distribution, we can approximate the integral by restricting the domain to discrete samples from this distribution). Then, given these samples, we're sampling $P(s_t|s_{t-1})$ on these specific values of s_{t-1}.

We can now plug this into the initial equation for $P(s_t|z_{1:t})$:

$$
\begin{aligned}
P(s_t|z_{1:t}) &\approx kP(z_t|s_t)\frac{1}{N}\sum_{i=1}^{N} P(s_t|s'_{t-1,i}) \\
&= \frac{k}{N}\sum_{i=1}^{N} P(z_t|s_t)P(s_t|s'_{t-1,i}) \\
&= \frac{k}{N}\sum_{i=1}^{N} P(z_t|s_{t,i})P(s_{t,i}|s_{t-1}) \\
&= \{s_{t,i}, w_{t,i}\}_{i=1}^{N}
\end{aligned}
$$

Recognize how each step here is valid. $P(z_t|s_t)$ is constant concerning i and we can redefine our distribution in terms of how particles are spread out concerning the previous distribution of particles. With this change, the likelihood term (measurement term) changes to measurements given the location of each particle. We can define this distribution thus as a sum of discrete particles with normalized weights (sum to one).

As such, we can define the following algorithmic procedure:

1. Redistribute $P(s_{t-1}|z_{1:t-1})$ through sampling and applying system noise to each location, in this implementation assumed to be $P(s|s_{t-1,i}) \sim N(s_{t,i}, Q)$ for Q being the system covariance matrix.

2. Perform a detection around each particle $s_{t,i}$ to produce measurements $z_{t,i}$. This implementation uses the Sum of Squared Differences (SSD) method.

3. Compute the initial weights via a Chi-Square distribution with two degrees of freedom and an expected value of $s_{t,i}$. Thus, $\alpha_{t,i} = P(z_{t,i}|s_{t,i})$.

4. Normalize the above weights.

Results

I implemented the above algorithm in MATLAB. I wrote several iterations of the algorithm to compensate for shortcomings and will attempt to show these in order.

The first iteration of my work had issues where particles would trend/float toward the top left of the image, which I sourced to my redistribution/sampling algorithm which was insufficient. I fixed this by implementing a uniform distribution through randomly generated row and column values. The following are some results with varying particle counts:

(a) Halfway (b) End

Figure 1: N = 100 with ball video

(a) Halfway (b) End

Figure 2: N = 300 with ball video

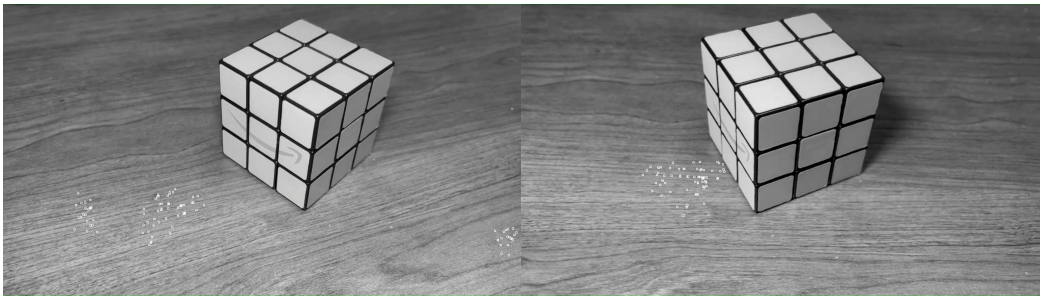
(a) Halfway (b) End

Figure 3: N = 500 with ball video

As one can expect, the algorithm was meant the track the ball, however wound up converging on several dark areas. I hypothesized that more particles would help the uniform distribution recognize the ball, i.e. that some particles would be in a close enough range to snap onto it. I was never able to get more than a few onto the ball, likely because the ball's speed "shook" any

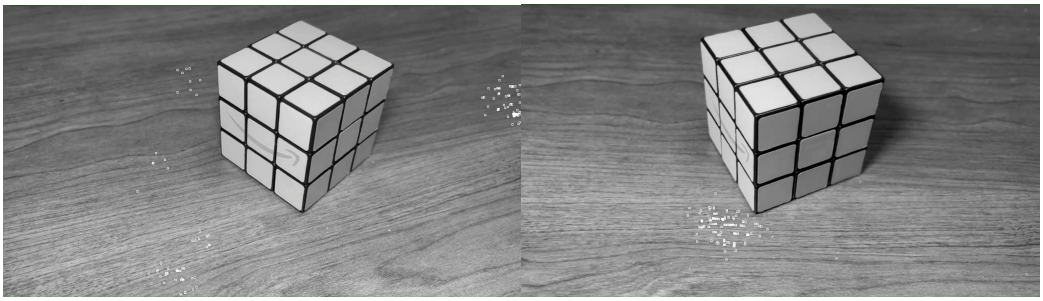
of said particles off, leading to these particles redistributing to other modes. Another thing I noticed was that smaller clouds of particles tended to die off over time. As the algorithm continued, less significant modes tended to die off. An explanation for this is difficult. Redistribution is not expected to change any modes, although it's always possible that some smaller areas get weaker (just as likely as it is for them to get stronger). More so, since these particles are weighted based on distance from a measured point, and they are already clustered around a measured point, their weights shouldn't change too much. As such my explanation for this trickles down to changes in the feature such as how fast it moves or how light hits it along with chance with the algorithm.

The following shows the same version of code attempting to track a Rubik's cube, tracking the frontmost corner (closest to the camera):



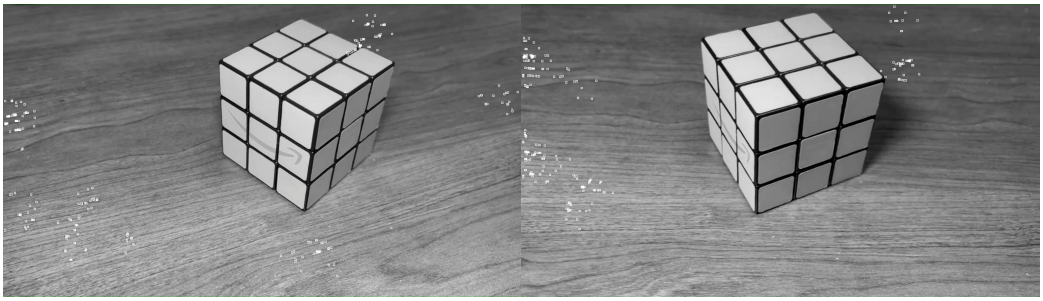
(a) Halfway (b) End

Figure 4: N = 100 with cube video



(a) Halfway (b) End

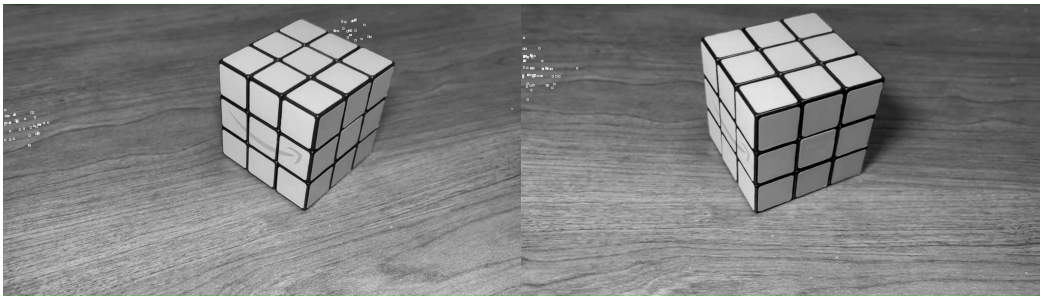
Figure 5: N = 300 with cube video



(a) Halfway (b) End

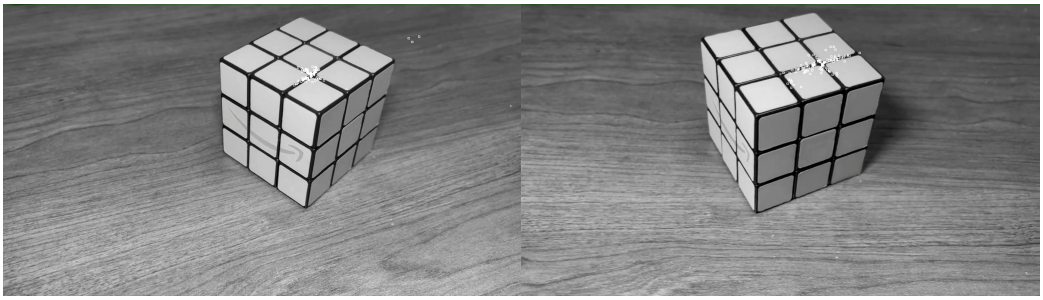
Figure 6: N = 500 with cube video

Note that this was much harder to track because of the many "false modes." Specifically, the black lines in the wood led to particles far from the cube finding modes and forming new clouds. This made me think about whether or not I could improve this algorithm by changing how particles are weighted. As such I added a factor to the weights of each particle relating to how good the measurement was (since I used SSD, this was related to the difference in pixel intensities of each window). This produced the following results on the cube:



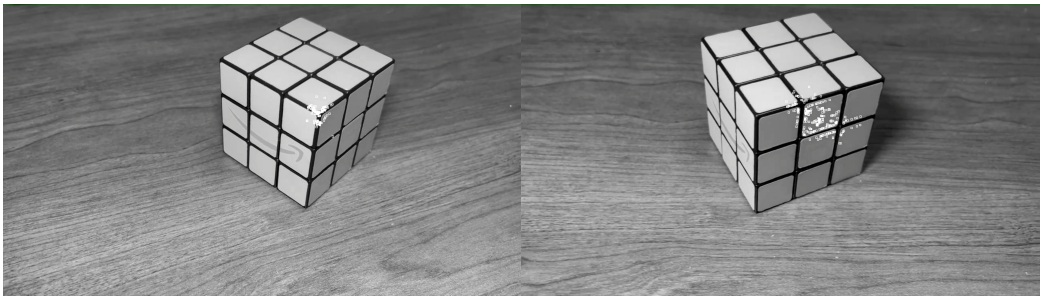
(a) Halfway (b) End

Figure 7: N = 100 with cube video, new weight



(a) Halfway (b) End

Figure 8: N = 300 with cube video, new weight



(a) Halfway (b) End

Figure 9: N = 500 with cube video, new weight

As seen here, there was a lot more success. By 500 particles, the algorithm was able to properly locate the desired target. However, now that weights are based on how close the measurement is to the expected measurement, the algorithm has significantly fewer modes. This iteration did not change much for the ball video, likely due to the lack of so many modes. My other solution involved increasing the search radius for the particles which saw limited improvement and took significantly longer to run.

Conclusion

Some of the advantages of Particle Filtering, as demonstrated and explained above, are its lack of assumptions and multi-modal distribution. This is an ideal way to track several points/possibilities on a screen, rather than just

how one moves over time. This is important in robotics, such as robot localization. Oftentimes there might be several locations where a robot could be given what it can see, while a Kalman Filter would only be able to track one. However, Particle Filtering can run into issues when there are lots of possible modes, such as with the Rubik's cube on a wooden table. A Kalman Filter can track the vertex we want over time well, while a Particle Filter may struggle to find the specific vertex simply because there are so many possible positions.

This leads to another major drawback with Particle Filtering: running time. The best way to improve the performance of a particle filter is to add more particles, with the notion that one can cover more of the screen and hit more possible locations. This rapidly increases running time. Some of the 500 particle videos I generated took over ten minutes to create while Kalman Filtering was done within a minute. Another possible solution would be to increase the search neighborhood around each particle, which similarly amplified the running time. Kalman filtering only has to search once, compared to N times for N particles. Similarly, its search radius shrinks over time as variance decreases, so the algorithm speeds up. However, once again, Kalman filters make several assumptions AND you have to predefine the first stage or perform object detection first.

In the end, it depends on what task one needs to accomplish. If you need to track multiple modes / non-Gaussian distributions, Particle Filtering will be the better choice but will likely be more noisy and also more computationally heavy. If you are fine with the drawbacks of Kalman Filtering, it is much faster.

As for the self-evaluation section of this conclusion, I am happy with my results. I went through three variations of code which helped me understand Particle Filtering (each was related to some drawbacks with the algorithm as described above). My work is in a mostly-finished state although I would like to finish improving the algorithm. Specifically, how I fixed the cube video strayed away from the intended implementation of Particle Filtering. A more proper solution would be to increase the search radius, but as mentioned before, this takes too much time. I would be interested in researching this problem more. Other than this, I am satisfied with my work. I enjoyed this project and believe I learned a lot from it.