3D Visual Tracking with Particle and Kalman Filters

Burak Bayramlı

June 29, 2010

Abstract

One of the most visually demonstrable and straightforward uses of filtering is in the field of Computer Vision. In this document we will try to outline the issues encountered while designing and implementing a particle and kalman filter based tracking system.

1 Introduction

Kalman and particle filters are examples of recursive filters; using Kalman filters for example, the state equations are in the form of

$$\begin{aligned} x_{t+1} &= Ax_t + Q \\ y_t &= Hx_t + R \end{aligned}$$

is utilized where x_t and y_t are distributed as Gaussians. The beauty of recursive filtering is that the incoming data is for y_t, and based on this data, we can ”reverse the arrow”, and calculate x_t, which is the hidden state. We know the transition equation beforehand (multiplication of x_t by A with the addition of noise, Q) as well as the observation equation, y_t. Together, these two equations could represent an object moving following a certain movement (a robot moving, a plane flying) and our observation error which could arise due to sensor sensitivity, unknown calibration parameters or other external conditions.

We need to emphasize however, that knowing transition and observation equations ”exactly” does not mean that the mathematical motivation for filtering is weak. For our example we had to determine a 3D location based on successive pixel readings, that is 2D data readings. As anyone who dealt with depth calculation from 2D images can attest to, we lose data when we transition from 3D to 2D; by looking at a single image, we can never determine where an object really is. We could be looking at an image of the Empire State building 1000 meters away, or a miniature of it, right in front of our nose. This is why 3D the ”reverse” calculation requires a succession of images; we match a pixel on all images in order to calculate its 2D displacement, using that and the viewer’s known physical 3D displacement, we can try to calculate (triangulate) a 3D location.

There are other problems. Using pure graphical methods, such as Multiple View Geometry for 3D calculation is not an easy task, somewhat error prone, and still does not give us true scale of an object. Filtering not only can take into account more than 2 images, but it can also account for noise, uncertainty of our transition and observation model, all based on an initial assumption which can encode scale assumptions in it as well. Then, every new measurement makes the estimation for x_t better, plus this method is suited perfectly for online applications – all history is encoded and reflected in x_t, historical values themselves are not kept beyond a single frame.

We implemented a tracking solution using both Kalman and particle filters. A chessboard plane was simply moved on a flat surface (table) on constant speed toward our camera while we tracked a single reference point on this image. State x_t was used to represent the 3D location of our chessboard plane. The transition equation of the Kalman filters only needed to account for constant velocity, along one axis. Matrix A looked like

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

Note that matrix A is 4x4, not 3x3. We used homogenous coordinates to represent 3D location, which made x_t a 4x1 vector, therefore the transition equation captured in A had to be 4x4 so we could multiply it with x_t.

A is the identity matrix I with one difference; its (3,4)’th item is a constant, d, that is our displacement. To verify if this A can be used for calculating displacement, one could perform a sample multiplication using $x_t = [\ a1 \quad a2 \quad a3 \quad a4\ ]$ and see Ax_t, or x_{t+1}, becomes $[\ a1 \quad a2 \quad a3 + d \quad a4\ ]$. We can see displacement d is added to the z-coordinate, depth. For our testing we picked d = -0.5, meaning for each frame chessboard plane moved 0.5 cm toward the camera. Each transition for x_t adds to uncertainty, and that is reflected in Q.

Observation equation y_t handles the calculation where a 3D coordinate is projected onto 2D (pixel measurements) on screen with added noise. For standard pinhole camera model, camera matrix P is responsible for 2D projection. P is unique for each camera, and a process called camera calibration can determine camera matrix P, and various methods for calibration exist in literature. OpenCV library contained a function for calibration which we tested, but we were not happy with the results. At the end, we calibrated the camera manually; by simply deciding on a 3D real world location (middle point at the end of our flat surface / table) and tested various P values while at the same time drawing an imaginery chessboard image on screen (shown as a green square on Figure **??**) using this projection matrix P. This was repeated until the imaginary board was located at the desired location.

We also tested if manual, virtual displacements in centimeters from the test point was reflected in the projection accurately. Projection matrices are 3x3, Kalman filter also adds a 4th row [0 0 0] to this P turning it into an H matrix in 4x3 dimensions.

The reason for using chessboard image was so `cvFindChessboardCorners` and `cvDrawChessboardCorners` OpenCV functions could be used. Using these two methods, a chessboard image can be detected and marked (on screen, real-time) with great accuracy and speed. The chessboard image had 9 squares on it, giving us 9 points of which, we only used the 5th, middle point. At each frame `cvFindChessboardCorners` detected the 2D pixel locations, and we passed these values over to the Kalman filter that recursively updated its hidden state.

We tested two scenarios for tracking, one starting from 36cm to the left, the other from 30 cm to the right to the midend point of the flat surface, in each case moving toward the camera in constant speed. In each case initial condition for the filter is the middle end point, somewhat away from both starting locations, therefore the uncertainty Q of the filter at time = 0 had to be large. Hence we used $Q = I * 150cm$ for the Kalman filter.

1.1 Particle Filters

Another method for recursive estimation is called particle filters. Particle filters are able to represent a distribution using "particles" which are weights in an array that are assigned to possible values of x_t which are elements in an array. These values and their associated weights together form a discrete distribution that can represent any distribution at an accuracy allowed by the number of particles. Bayesian filtering through particles is achived by first applying a transition where uncertainty increases, then we apply the observation function to "update" the distribution by processing an error function. The error function for each particle is

$$w^{[i]} = \frac{1}{1 + (y^{[i]} - p^{[i]})^2}$$

where $y^{[i]}$ is an observation value and $p^{[i]}$ is the "guess" that was calculated by applying the transition function to previous estimate, akin to $Ax_t + Q$ transition of the Kalman filter. That transition for particle filters is calculated by sampling from a uniform distribution and adding the results to x, for each particle. We sample from $\mathtt{Unif}(-0.1, -1)$ for z-coordinate addition, and from $\mathtt{Unif}(-40, 40)$ for x-coordinate addition. In other words, we add forward motion between 0.1 and 1 centimeters, and an uncertainty bubble 80 cm wide, horizontally, in both directions.

The reason we add '1' to $(y^{[i]} - p^{[i]})^2$ in the fraction should be clear; this way the error function can give back a probability-like result. For small errors, $1 + \mathtt{error}$ divides the '1' in the nominator, giving back a number close to 1. This is what we want, smaller errors resulting in greater weight, hence greater probability, larger errors causing the opposite.

The resampling process is also straightforward; it makes sure that particles, values with greater weight are repeated more than particles with smaller probabilities. Note however resampling procedure does not create new particle values, it simply repeats (or skips) existing particles.

2 Conclusion

All code was written in Python language, using Scipy and Numpy libraries, and also OpenCV. The results received from this experiment were satisfactory, it was witnessed that in both Kalman and particle filter cases tracking worked successfully. Our example code can be downloaded from our blog[1], and selected screenshots for both approaches can be found below.

References

[1] S. Marsland, *Machine Learning: An Algorithmic Perspective*, CRC Press, 2009.

[1]http://ascratchpad.blogspot.com/2010/06/3d-tracking-using-kalman-and-particle.html

[2] S. Thrun, W. Burgard, and D. Fox, *Probabilistic Robotics*, MIT Press, Cambridge, MA, 2005