Arshia Firouzi

Kalman Filter Software

Using only an accelerometer for positional tracking is inaccurate in practice as the double integral of the acceleration magnifies (quadratically) and accumulates error in the actual system's position (drift). This can be solved by combining the system with a GPS and a Kalman filter to correct for drift by estimating the true position with all three components. Here we have created a Kalman Filter for future use in this integrated system. This version is for two-dimensions, position and velocity, however will be reduced to just position as velocity measurements cannot be made, only calculated from the accelerometer. The reason it was decided to create a 2D filter is if velocity can somehow be tracked (difference and direction of position measured positions with time between measurements?), or if tracking acceleration instead of just measuring it with the accelerometer becomes useful.

The Kalman Filter works (at a high-level) as shown by the cycle in Figure 1 below.

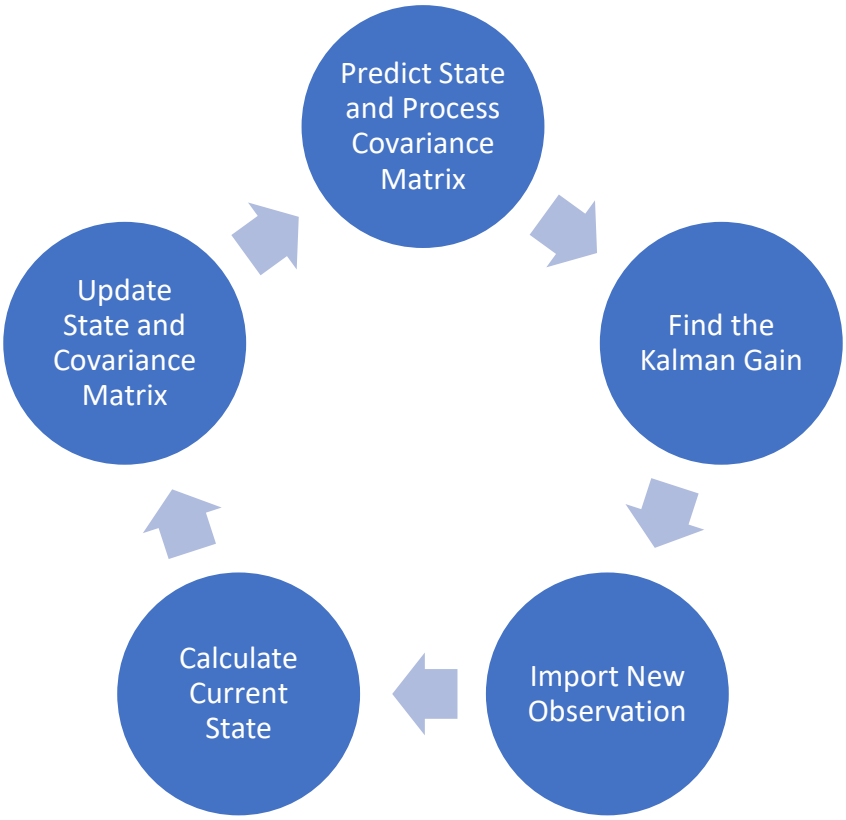

Figure 1

The following pages dive into what a Kalman Filter is and the software design, decision making, and future strategy for its use.

Arshia Firouzi

Kalman Filter Overview

Kalman Filter Class – Initialization & Mathematics

Before the Kalman Filter cycle can begin, a few matrices need to be initialized. Much of the information stated here can be found in the comments of the full code, mainly in the "__init__" method. The variable "dt" is the amount of time that has passed since the last prediction. Initially it is simply set to 1 (meaning 1 second), but once the first cycle is executed it becomes the actual amount of time/cycle. This variable is important as position and velocity are calculated/tracked via Equation 1 below.

$$x = x_o + vt + \frac{1}{2}at^2$$

Equation 1

The state matrix (named simply "State") is initialized (essentially "x" in the above equation) as a 2x3 matrix of zeros as there should be no velocity in any direction (assumption) and the object being tracked should be at the origin. There are two rows and three columns as the position and velocity is being tracked in all three dimensions. The state matrix has the following form:

$$\text{X} = \begin{bmatrix} X-Position, & Y-Position, & Z-Position, \\ X-Velocity, & Y-Velocity, & Z-Velocity, \end{bmatrix}$$

Matrix X

Equation 2 (below) is Equation 1 in matrix form. Matrix A in Equation 2 is a 2x2 matrix that takes the previous state's position/velocity (X_{n-1}) and turns those components into a 2x3 matrix (like Matrix X) with values from only the effect previous velocity to the position over the elapsed cycle time. Matrix U is the observed acceleration from the previous cycle. Matrix B takes the information from U (much like Matrix A and X_{n-1}) and transforms it into the position and velocity changes resulting from the previous cycle. Matrix W is noise and ignored in this program for simplicity. By adding the previous two matrices up, we have our predicted next state.

$$X_n = AX_{n-1} + BU_{n-1} + W$$

Equation 2

$$A = \begin{bmatrix} Previous\ Position & Previous\ Velocity * dt \\ 0 & Previous\ Velocity \end{bmatrix}$$

Matrix A

$$B = \begin{bmatrix} (\frac{1}{2}dt^2) \\ dt \end{bmatrix}$$

Matrix B

Next, the process covariance matrix (a matrix containing the information of error and cross-talk between our measured values) is initialized via the calculations for Matrix P below. This occurs only once in "__init__". From here on out P is simply updated via Equation 3. The Q matrix is the process noise (ie. latency) and ignored for simplicity.

A covariance matrix is a matrix consisting of calculations using the variance (standard deviation squared) of each tracked variable (in this case the position and velocity of our object). These variances are input by the user and are usually the expected errors due to the overall system (ie. latency). The zero'd terms in the matrix are actually covariance terms with defined values that are ignored for simplicity. They each should be, in this case, the product of the positional and velocity variance. A is Matrix A defined above. A^T is the transpose of Matrix A.

$$P_n = AP_{n-1}A^T + Q$$

Equation 3

$$P = \begin{bmatrix} (Process\ Positional\ Variance)^2 & 0 \\ 0 & (Process\ Velocity\ Variance)^2 \end{bmatrix}$$

Matrix P

The Kalman Gain (K) in this case is a 2x3 matrix calculated using Equation 4 below. The H matrix is transformation matrix meant to transform P into the form of the Kalman Gain. It is defined by the system. In our case, it is a 2x2 identity matrix. H^T is the transpose of the H matrix, which is again a 2x2 identity matrix in this case. Matrix R is a covariance matrix, like P, except that it contains the observational variance of each tracked variable (ie. sensor error). R, unlike P, is not updated each cycle and does not change. Again, like in P, covariance terms are ignored for simplicity.

$$K = \frac{PH}{HPH^T + R}$$

Equation 4

$$R = \begin{bmatrix} (Observation\ Positional\ Variance)^2 & 0 \\ 0 & (Observation\ Velocity\ Variance)^2 \end{bmatrix}$$

Matrix R

With the Kalman Gain K, the process covariance matrix P can be updated using Equation 5 below. Furthermore, the current state can be calculated by taking in an observation (Matrix Y below) and the calculation in Equation 6. Note that the output state of this calculation will become the previous state used for the next prediction.

During initialization, the only calculation that occurs is those in setting up the covariance matrices P and R. Otherwise, the other matrices (A, B, H, X, U, K) are simply initialized as blank/identity matrices. If a matrix I is ever seen, this is an identity matrix.

$$P_n = (I - KH)P_n$$

Equation 5

$$Y = \begin{bmatrix} Observed\ X - Position, & Observed\ Y - Position, & Observed\ Z - Position, \\ Observed\ X - Velocity, & Observed\ Y - Velocity, & Observed\ Z - Velocity, \end{bmatrix}$$

Matrix Y

$$X_n = X_n + K(Y - HX_n)$$

Equation 6

During initialization, the only calculation that occurs is those in setting up the covariance matrices P and R. Otherwise, the other matrices (A, B, H, X, U, K) are simply initialized as blank/identity matrices. If a matrix I is ever seen, this is an identity matrix.

Kalman Filter Class – "predictState"

This method uses Matrices A and B, as well as the previous state of the system separated into Matrices X_{n-1} and U_{n-1}. It computes Equation 2 (where Matrix W is set to zero) and returns the predicted stated (X_n) in the form of Matrix X.

Kalman Filter Class – "predictPCOV"

This method takes the previous PCOV (P_{n-1} – whether it's the initialized version, or updated after the 1st cycle) and predicts the next PCOV via Equation 3. In addition to the previous PCOV, only Matrix A is necessary to do so. After calculation, the predicted PCOV P_n is returned in the form of Matrix P.

Kalman Filter Class – "calcKalmanGain"

With P_n computed and the Matrices H and R initialized, this method can now calculate the Kalman Gain. Equation 4 is utilized in doing so, and the Matrix K (a 2x3 matrix containing the Kalman Gain terms) is returned.

Kalman Filter Class – "Observe"

This method is an API of sorts as it imports sensor data in a useful format. It then transforms the sensor observations into a form that can be used to calculate the imported current state. This returned observation is Matrix Y, a 3x3 matrix with positional, velocity, and acceleration data in all three dimensions (X, Y, Z) according to the sensors.

Kalman Filter Class – "calcCurrentState"

We now have all of the necessary information to calculate the new state and P matrix. This method does the former, utilizing Equation 6, the predicted Matrix X_n, observation Matrix Y, the Kalman Gain Matrix K, and Matrix H. This new output "current" state becomes X_{n-1} in the next cycle.

Kalman Filter Class – "updatePCOV"

Using Equation 5, this method updates the PCOV Matrix P with the predicted Matrix P_n, Kalman Gain Matrix K, identity Matrix I, and Matrix H. This new becomes P_{n-1} in the next cycle.

Function – "main"

The main function…