IGVC 2021 EKF Derivation

Kevin Robb,
based on the work of Justin Kleiber

July 17, 2021

1 Introduction

The EKF process begins with a one-time initialization: essentially a psuedo-measurement stage in which we give the filter an educated guess of our starting position. This is almost always going to be a full state of zeroes, since the robot is not moving when it turns on, and the starting point is the origin of our coordinate system.

After the initialization, the EKF cycles back and forth between the *Predict* and *Update* stages forever. The prediction phase involves state estimation and covariance extrapolation. The update phase involves calculating the innovation from new measurements, deriving the Kalman Gain, and updating the state and the covariance. After every update, the state is published to a ROS topic where it can be picked up by any other node that can use the localization information.

The following sections detail each of these aspects as they were used for the IGVC 2021 EKF.

2 The State

A Kalman Filter's main goal is to track and estimate a *state*, which is a column vector containing variables we care about.

$$\hat{\boldsymbol{x}} = \begin{pmatrix} x & \dot{x} & y & \dot{y} & \phi & \dot{\phi} & v_l & v_r \end{pmatrix}^T \tag{1}$$

We use the EKF primarily for tracking our position, x and y, and our global heading ϕ. To do this well, we also need to track the yaw rate $\dot{\phi}$, component velocities $\dot{x}$ and $\dot{y}$, and wheel velocities v_l and v_r. Since we don't care about the actual performance of the EKF when tracking these subsidiary variables, we can assume they're constant in our calculations without hindering the accuracy of the variables we do care about. This assumption of constant velocities makes our motion model very simple, but still quite powerful for the variables with accurate motion models.

The "hat" over the state indicates that it is an estimate, since we can't ever say definitively that our state equals the ground truth. As a further notation clarification, when a vector or matrix is subscripted by two values, the first is the timestep for which the variable estimates, and the second is the timestep in which the estimate was made. So a prediction for the next state made at timestep k would be denoted $\hat{\boldsymbol{x}}_{k+1,k}$, while a prediction for the current state made in the previous timestep would be $\hat{\boldsymbol{x}}_{k,k-1}$. When the state is updated, it also applies to the current timestep, so this would be $\hat{\boldsymbol{x}}_{k,k}$.

3 Motion Model

The motion model, f, is responsible for predicting the next state given the current state (and the control commands, if applicable).

I use a simpler motion model than Justin's from 2020. Primarily, we do not use GPS coordinates or control parameters for making predictions, and these variables are not included in the state.

$$f(\hat{\boldsymbol{x}}_{k-1}, \boldsymbol{u}, \Delta t) = \begin{pmatrix} x_{k-1} + \dot{x}_{k-1} \cdot \Delta t \\ \dot{x}_{k-1} \\ y_{k-1} + \dot{y}_{k-1} \cdot \Delta t \\ \dot{y}_{k-1} \\ \phi_{k-1} + \dot{\phi}_{k-1} \cdot \Delta t \\ \dot{\phi}_{k-1} \\ v_{l,k-1} \\ v_{r,k-1} \end{pmatrix} \tag{2}$$

We need to encode this system of equations into a matrix that can be used directly in the EKF. We cannot use simple matrix multiplication to compute $\cos\phi$ and $\sin\phi$, so these are re-computed and set at the start of the *Predict* phase on every clock cycle, and treated as if they are constants (cos_phi and sin_phi). Important robot characteristics are the constants $R =$ WHEEL_RADIUS and $L =$ WHEELBASE_LEN.

$$\boldsymbol{F}_k = \begin{pmatrix} 1 & \Delta t & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \frac{R}{2}\text{cos\_phi} & \frac{R}{2}\text{cos\_phi} \\ 0 & 0 & 1 & \Delta t & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \frac{R}{2}\text{sin\_phi} & \frac{R}{2}\text{sin\_phi} \\ 0 & 0 & 0 & 0 & 1 & \Delta t & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \frac{R}{L} & -\frac{R}{L} \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \tag{3}$$

This matrix gives us the correct behavior of predicting the next state given the current state, as shown in the following equation.

$$\hat{\boldsymbol{x}}_{k+1} = \boldsymbol{F}_k \cdot \hat{\boldsymbol{x}}_k \tag{4}$$

We use SymPy to calculate the Jacobean of the motion model. SymPy is a great python package for performing symbolic operations such as this. We obtain the following Jacobean, which is useful for linearizing the EKF.

$$\begin{pmatrix} 1 & \Delta t & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -\frac{R}{2}(v_l + v_r)\sin\phi & 0 & \frac{R}{2}\cos\phi & \frac{R}{2}\cos\phi \\ 0 & 0 & 1 & \Delta t & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{R}{2}(v_l + v_r)\cos\phi & 0 & \frac{R}{2}\sin\phi & \frac{R}{2}\sin\phi \\ 0 & 0 & 0 & 0 & 1 & \Delta t & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \frac{R}{L} & -\frac{R}{L} \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \tag{5}$$

4 The Measurements

After predicting what the state will look like at the next timestep, we need a way to check this prediction against the real world. Since we don't know the true values of our state variables unless we're working exclusively in simulation, we can't simply compare them to some ground truth to evaluate our filter's accuracy.

An important step in the EKF cycle is the *Update* phase, in which the current values of all sensors are recorded. Sensors tend to have different update frequencies, so in my code I have a "buffer" vector that is updated anytime new sensor values are received from a ROS topic, and on the *Update* step, all values in the buffer are loaded into the actual measurement vector, $\boldsymbol{z}$.

$$\boldsymbol{z} = \begin{pmatrix} x & y & \phi & \dot{\phi} & v_l & v_r \end{pmatrix}^T \tag{6}$$

These first two entries correspond to the x and y position of the robot. We can't directly measure these, but we can include our GPS measurements in order to give the filter more information to work with. We use the fact that in a small, localized area, GPS coordinates can be linearly converted to meters without requiring complex calculations involving the radius of the earth. We use the website http://www.csgnetwork.com/degreelenllavcalc.html and enter the coordinates of the venue we will be using. For our simulator, we use the OU football stadium; for the IGVC competition, we looked up the GPS coordinates of the host school, Oakland University. It is very simple to update these values once on-site, as we can use our GPS sensor to grab the longitude and latitude around the center of the course, and use the linked website to acquire new conversion factors. These values appear in my EKF code as *LAT_TO_M* and *LON_TO_M*.

The next two measured values are the yaw and yaw rate, taken from the IMU. An IMU typically outputs a quaternion, so we use the transformations package to convert this into an euler tuple of yaw, pitch, and roll, which are much more easily human-comprehensible. These are simple to compare to the state, as we're tracking something we can directly measure.

The final two measured values are left and right wheel velocities obtained from encoders on the drive wheels. In our case, the encoders publish linear velocities in m/s, but we convert them to angular velocities using the known wheel radius before saving them as a measurement.

With our measurements obtained, we need a way to compare them to the predictions and update the state. This is where our measurement model comes in.

5 Measurement Model

The measurement model serves the goal of allowing us to retrieve the equivalent measurement values for a given state. This is necessary for calculating the Innovation, and thus the Kalman Gain, which allows us to update the state in the best way possible by taking both the predictions and the measurements into account. This use implies the necessity of being able to retrieve the measurements from the state, which is why we include the wheel velocities v_l and v_r in the state despite not actually using these values for navigation; without including them directly, it is hard to recover them from the remaining state variables.

The desired behavior is

$$\boldsymbol{z}_{\text{equiv}} = \boldsymbol{H} \cdot \hat{\boldsymbol{x}}_{n,n-1} \tag{7}$$

In words, we multiply $\boldsymbol{H}$ by the prediction for the current state made in the previous timestep to recover an equivalent measurement. Since our measurement vector is simply a subset of the state, our measurement model is

$$\boldsymbol{H} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \tag{8}$$

We can then calculate our innovation,

$$\boldsymbol{y}_n = \boldsymbol{z}_n - \boldsymbol{z}_{\text{equiv}} \tag{9}$$

6 Covariance and Uncertainty

Covariance is the relationship that every variable has with every other variable, and it is very important in multidimensional systems. We have our main covariance $\boldsymbol{P}$, also called the estimate uncertainty, which is modified both in the *Predict* and in the *Update* phases.

In the *Predict* step, we extrapolate the estimate uncertainty to create a prediction for $\boldsymbol{P}$ at the next timestep. This is done by using our state transition matrix previously obtained from the motion model.

$$\boldsymbol{P}_{k+1} = \boldsymbol{F} \cdot \boldsymbol{P} \cdot \boldsymbol{F}^T + \boldsymbol{Q} \tag{10}$$

where $\boldsymbol{Q}$ is a matrix representing the process noise. This is hard to nail down and understand physically, but think of it as the uncertainty in the process of the EKF itself. There are formulas and ways to find more complicated versions of all entries, but it suffices in our case to simply use the 8x8 identity matrix multiplied by a constant that we can tune. This constant tends to remain in the range (1.1, 1.3) in our application.

In addition to these uncertainties in the estimations and in the process, we of course have uncertainties in our measurements. If sensors give direct values for their variance, those should be used. We define the measurement uncertainty as a 6x6 matrix, $\boldsymbol{R}$, with the variance of each measurement as the entry on the diagonal corresponding to that measurement's position in $\boldsymbol{z}$.

We know that our approximations for the positions using GPS coordinates are fairly scuffed and have a standard deviation around 5 meters, whereas our IMU and encoders can easily be more accurate than a tenth of a unit. As such, we used the following matrix as a starting point. We frequently modify these values on the fly based on which sensors are performing well and which ones are having more interference than expected. Check the competition build of the code to see what we finalized at the venue.

$$\boldsymbol{R} = \begin{pmatrix} 35 & 0 & 0 & 0 & 0 & 0 \\ 0 & 25 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.01 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.01 \end{pmatrix} \tag{11}$$

Note that if a sensor is being absolutely horrible or breaks it can be essentially disabled by setting its uncertainty value in this matrix to something huge like 100,000. Even if the filter never receives data from a sensor (i.e., it is unplugged or not functioning), it will use the starting value, 0, weighted with a massive uncertainty; this means the filter will still work if a sensor is disconnected, albeit not as well as with all the expected data.

7 The Kalman Gain

The Kalman Gain, $\boldsymbol{K}$, is the heart of the kalman filter's *Update* step. It is a matrix that allows uncertainties and covariances to influence the evolution of the state, and it depends on the innovation covariance $\boldsymbol{S}$.

$$\boldsymbol{S} = \boldsymbol{H} \cdot \boldsymbol{P} \cdot \boldsymbol{H}^T + \boldsymbol{R} \tag{12}$$

The inverse of this innovation covariance is directly used to compute the Kalman Gain.

$$\boldsymbol{K} = \boldsymbol{P} \cdot \boldsymbol{H}^T \cdot \boldsymbol{S}^{-1} \tag{13}$$

Note: This equation should work, but we noticed the EKF was blowing up to infinity because of weird divergences in the off-diagonal elements of the Kalman Gain. As such, we made a last-minute hack by multiplying this $\boldsymbol{K}$ component-wise by $\boldsymbol{H}^T$. This has the effect of forcing all off-diagonals to zero, which is definitely bad but fixed the problems in our time of need. Please do something better than this in the future.

Now that we have $\boldsymbol{K}$, we can update the state and the covariance, using our innovation and the predictions for $\boldsymbol{x}$ and $\boldsymbol{P}$ made in the previous timestep.

$$\hat{\boldsymbol{x}}_{k,k} = \hat{\boldsymbol{x}}_{k,k-1} + \boldsymbol{K} \cdot \boldsymbol{y}_k \tag{14}$$

$$\boldsymbol{P}_{k,k} = (\boldsymbol{I}_8 - \boldsymbol{K} \cdot \boldsymbol{H}) \cdot \boldsymbol{P}_{k,k-1} \tag{15}$$

where $\boldsymbol{I}_8$ is the 8x8 identity matrix.

Note: Equation (15) is a simplified version of the equation which mostly works, but is unstable, and will multiply significant errors which could ruin the whole filter. It works mostly fine, but for reference the more correct equation is included below.

$$\boldsymbol{P}_{k,k} = (\boldsymbol{I}_8 - \boldsymbol{K} \cdot \boldsymbol{H}) \cdot \boldsymbol{P}_{k,k-1} \cdot (\boldsymbol{I}_8 - \boldsymbol{K} \cdot \boldsymbol{H})^T + \boldsymbol{K} \cdot \boldsymbol{R} \cdot \boldsymbol{K}^T \tag{16}$$

8 Overall Process and Final Thoughts

When all the previous logic has been implemented in a class, the actual ROS node is extremely simple. It involves simply initializing the filter, starting a timer, and defining the timer callback to loop through the Predict, Measure, and Update stages. This should make it easy to separately develop the filter from the node that uses the filter.

In the actual competition, it was not done as neatly as this; I attempted to remedy this afterwards by deleting all dead code and consolidating the things I had written into one python class file and one ROS node file, but could not get it to work with the simulator. The gross but working build is in master, but my cleaner (but not exactly working) build is in the cleanup/ekf branch.

Feel free to ask me (Kevin) or Justin your EKF-related questions in the Slack or Discord. Even if you're making your own thing rather than using what we've done, we'd still be able to help with your ideas and understanding. Best of luck.