Kalman Filte

Abstract

The Kalman filter (KF) has received a huge interest from the industrial electronics community and has played a key role in many engineering fields since the 1980s, ranging, without being exhaustive, trajectory estimation, state and parameter estimation for control or diagnosis, data merging, signal processing, and so on. This paper provides a brief overview of the industrial applications and implementation issues of the KF in six topics of the industrial electronics community, highlighting some relevant reference papers and giving future research trends.

Introduction

Kalman filter is an algorithm that provides estimates of some unknown variables given the measurements observed over time. Kalman filters have been demonstrating its usefulness in various applications. Kalman filters have relatively simple form and require small computational power.

A common application is for guidance, navigation, and control of vehicles, particularly aircraft, spacecraft and dynamically positioned ships. Furthermore, the Kalman filter is a widely applied concept in time series analysis used in fields such as signal processing and econometrics. Kalman filters also are one of the main topics in the field of robotic motion planning and control and can be used in trajectory optimization. The Kalman filter also works for modeling the central nervous system's control of movement. Due to the time delay between issuing motor commands and receiving sensory feedback, use of the Kalman filter supports a realistic model for making estimates of the current state of the motor system and issuing updated commands.

The algorithm works in a two-step process. In the prediction step, the Kalman filter produces estimates of the current state variables, along with their uncertainties. Once the outcome of the next measurement (necessarily corrupted with some amount of error, including random noise) is observed, these estimates are updated using a weighted average, with more weight being given to estimates with higher certainty. The algorithm is recursive. It can run in real time, using only the present input measurements and the previously calculated state and its uncertainty matrix; no additional past information is required.

Optimality of the Kalman filter assumes that the errors are Gaussian.

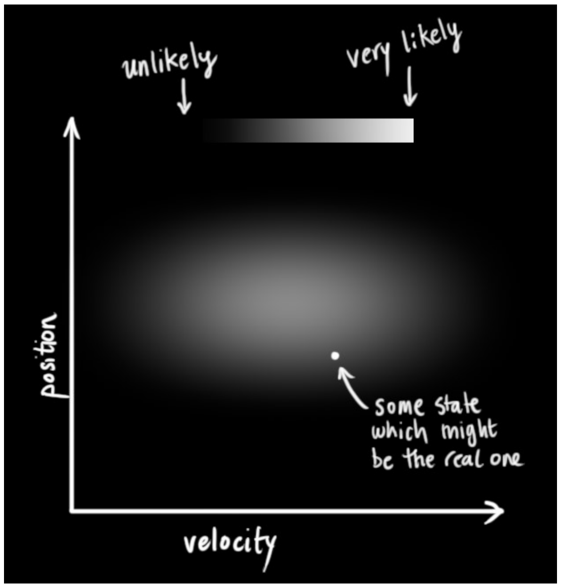
Extensions and generalizations to the method have also been developed, such as the extended Kalman filter and the unscented Kalman filter which work on nonlinear systems. The underlying model is a hidden Markov model where the state space of the latent variables is continuous and all latent and observed variables have Gaussian distributions. Also, Kalman filter has been successfully used in multi-sensor fusion, and distributed sensor networks to develop distributed or consensus Kalman filter.

How a Kalman filter sees your problem

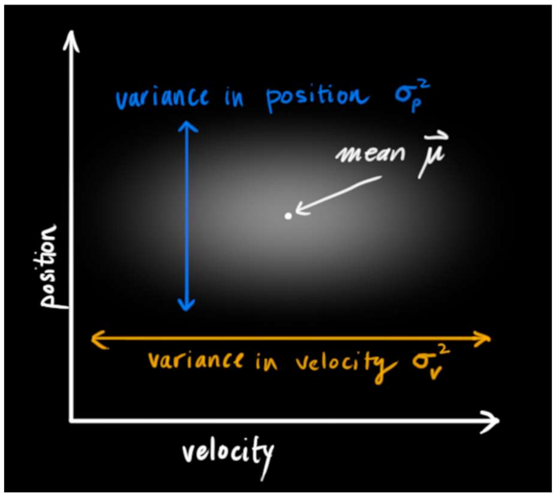
Let’s look at the landscape we’re trying to interpret. We’ll continue with a simple state having only position and velocity.

$\vec{x} =[pv]$

We don’t know what the *actual* position and velocity are; there are a whole range of possible combinations of position and velocity that might be true, but some of them are more likely than others:

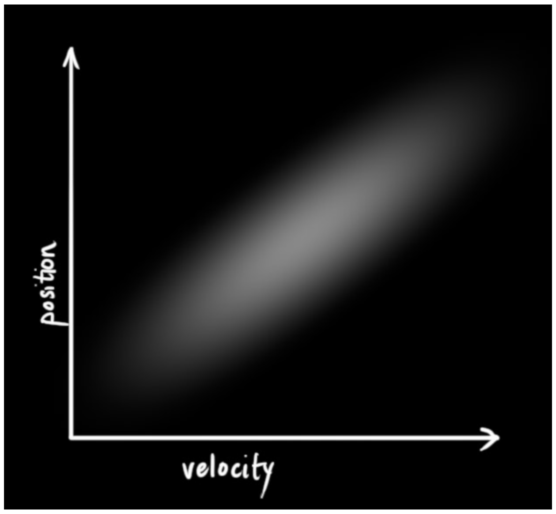


The Kalman filter assumes that both variables (postion and velocity, in our case) are random and *Gaussian distributed.* Each variable has a mean value μ, which is the center of the random distribution (and its most likely state), and a variance $\sigma 2$, which is the uncertainty:


variance in position σ_p^2
mean $\vec{\mu}$
position
variance in velocity σ_v^2
velocity

In the above picture, position and velocity are uncorrelated, which means that the state of one variable tells you nothing about what the other might be.

The example below shows something more interesting: Position and velocity are correlated. The likelihood of observing a particular position depends on what velocity you have:

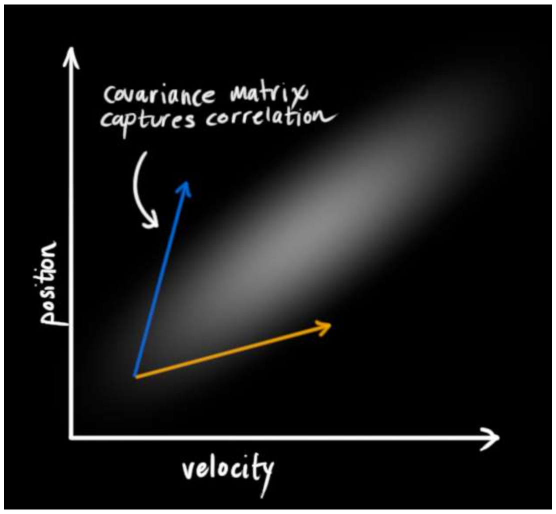


This kind of situation might arise if, for example, we are estimating a new position based on an old one. If our velocity was high, we probably moved farther, so our position will be more distant. If we're moving slowly, we didn't get as far.

This kind of relationship is really important to keep track of, because it gives us more information: One measurement tells us something about what the others could be. And that's the goal of the Kalman filter, we want to squeeze as much information from our uncertain measurements as we possibly can!

This correlation is captured by something called a covariance matrix. In short, each element of the matrix Σij is the degree of correlation between the *ith* state variable and the *jth* state variable.

You might be able to guess that the covariance matrix is symmetric, which means that it doesn't matter if you swap i and j. Covariance matrices are often labelled "**Σ**", so we call their elements "Σij".



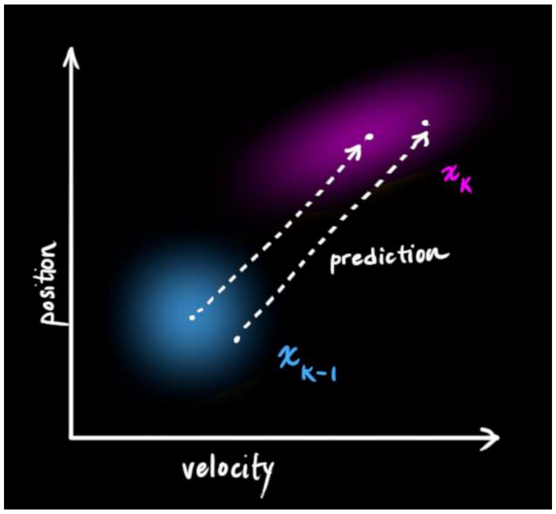
Describing the problem with matrices

We're modeling our knowledge about the state as a Gaussian blob, so we need two pieces of information at time *k*: We'll call our best estimate **x̂ k** (the mean, elsewhere named μ), and its covariance matrix **Pk**.

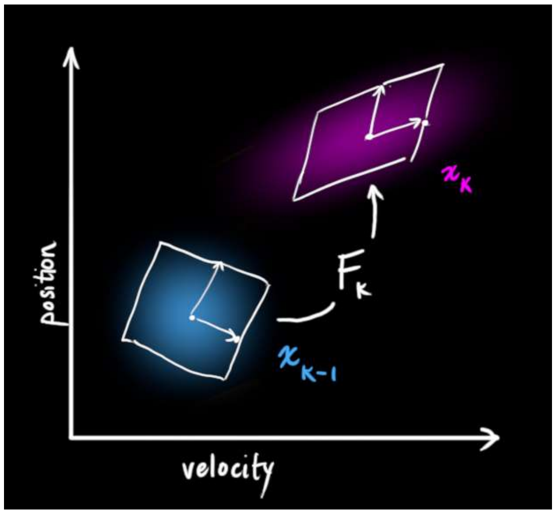
$$\hat{\mathbf{x}}_k = \begin{bmatrix} \text{position} \\ \text{velocity} \end{bmatrix}$$
$$\mathbf{P}_k = \begin{bmatrix} \Sigma_{pp} & \Sigma_{pv} \\ \Sigma_{vp} & \Sigma_{vv} \end{bmatrix}$$

(Of course we are using only position and velocity here, but it's useful to remember that the state can contain any number of variables, and represent anything you want).

Next, we need some way to look at the current state (at time k-1) and predict the next state at time k. Remember, we don't know which state is the "real" one, but our prediction function doesn't care. It just works on *all of them*, and gives us a new distribution:



We can represent this prediction step with a matrix, **Fk**:



It takes *every point* in our original estimate and moves it to a new predicted location, which is where the system would move if that original estimate was the right one.

Let’s apply this. How would we use a matrix to predict the position and velocity at the next moment in the future? We’ll use a really basic kinematic formula:

$$\begin{aligned} p_k &= p_{k-1} + \Delta t v_{k-1} \\ v_k &= \phantom{p_{k-1} + \Delta t} v_{k-1} \end{aligned}$$

In other words:

$$\begin{aligned}\hat{\mathbf{x}}_k &= \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix} \hat{\mathbf{x}}_{k-1} \\ &= \mathbf{F}_k \hat{\mathbf{x}}_{k-1}\end{aligned}$$

We now have a prediction matrix which gives us our next state, but we still don't know how to update the covariance matrix.

This is where we need another formula. If we multiply every point in a distribution by a matrix **A**, then what happens to its covariance matrix Σ?

Well, it's easy. I'll just give you the identity:

$$\begin{aligned}Cov(x) &= \Sigma \\ Cov(\mathbf{A}x) &= \mathbf{A}\Sigma\mathbf{A}^T\end{aligned}$$

So combining (4) with equation (3):

$$\begin{aligned}\hat{\mathbf{x}}_k &= \mathbf{F}_k \hat{\mathbf{x}}_{k-1} \\ \mathbf{P}_k &= \mathbf{F}_\mathbf{k} \mathbf{P}_{k-1} \mathbf{F}_k^T\end{aligned}$$

External influence

We haven't captured everything, though. There might be some changes that aren't related to the state itself— the outside world could be affecting the system.

For example, if the state models the motion of a train, the train operator might push on the throttle, causing the train to accelerate. Similarly, in our robot example, the navigation software might issue a command to turn the wheels or stop. If we know this additional information about what's going on in the world, we could stuff it into a vector called **u***k*→, do something with it, and add it to our prediction as a correction.

Let’s say we know the expected acceleration a due to the throttle setting or control commands. From basic kinematics we get:

$$\begin{aligned} p_k &= p_{k-1} + \Delta t v_{k-1} + \frac{1}{2} a \Delta t^2 \\ v_k &= \qquad\qquad\quad v_{k-1} + a \Delta t \end{aligned}$$

In matrix form:

$$\begin{aligned} \hat{\mathbf{x}}_k &= \mathbf{F}_k \hat{\mathbf{x}}_{k-1} + \begin{bmatrix} \frac{\Delta t^2}{2} \\ \Delta t \end{bmatrix} a \\ &= \mathbf{F}_k \hat{\mathbf{x}}_{k-1} + \mathbf{B}_k \vec{\mathbf{u}_k} \end{aligned}$$

$\mathbf{B}k$ is called the control matrix and $\mathbf{u}k\longrightarrow$ the control vector. (For very simple systems with no external influence, you could omit these).

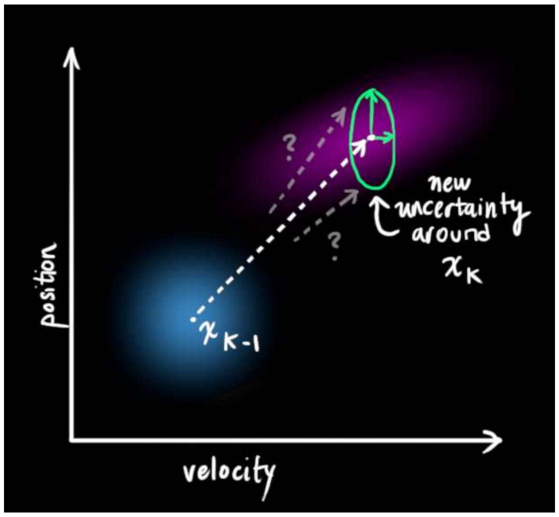
Let’s add one more detail. What happens if our prediction is not a 100% accurate model of what’s actually going on?

External uncertainty

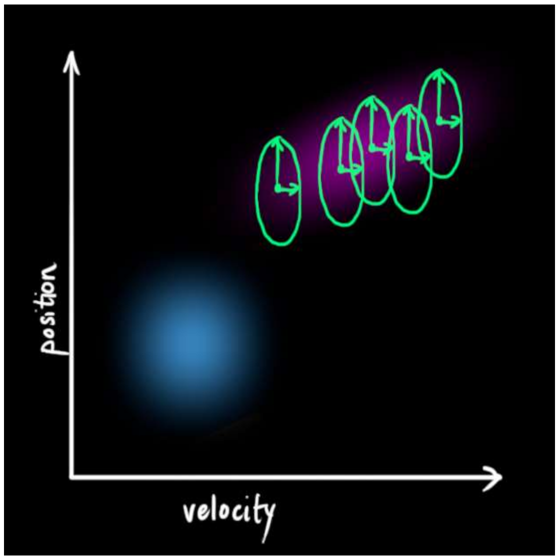
Everything is fine if the state evolves based on its own properties. Everything is *still* fine if the state evolves based on external forces, so long as we know what those external forces are.

But what about forces that we *don’t* know about? If we’re tracking a quadcopter, for example, it could be buffeted around by wind. If we’re tracking a wheeled robot, the wheels could slip, or bumps on the ground could slow it down. We can’t keep track of these things, and if any of this happens, our prediction could be off because we didn’t account for those extra forces.

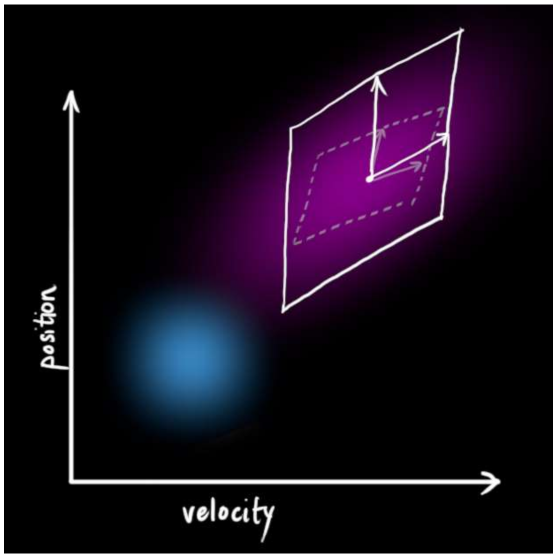
We can model the uncertainty associated with the “world” (i.e. things we aren’t keeping track of) by adding some new uncertainty after every prediction step:



Every state in our original estimate could have moved to a *range* of states. Because we like Gaussian blobs so much, we'll say that each point in $\hat{\mathbf{x}}_{k-1}$ is moved to somewhere inside a Gaussian blob with covariance $\mathbf{Q}_k$. Another way to say this is that we are treating the untracked influences as noise with covariance $\mathbf{Q}_k$.



This produces a new Gaussian blob, with a different covariance (but the same mean):



We get the expanded covariance by simply adding $\mathbf{Q}k$, giving our complete expression for the prediction step:

$$\begin{aligned}\hat{\mathbf{x}}_k &= \mathbf{F}_k\hat{\mathbf{x}}_{k-1} + \mathbf{B}_k\overrightarrow{\mathbf{u}_k}\\ \mathbf{P}_k &= \mathbf{F}_\mathbf{k}\mathbf{P}_{k-1}\mathbf{F}_k^T + \mathbf{Q}_k\end{aligned}$$

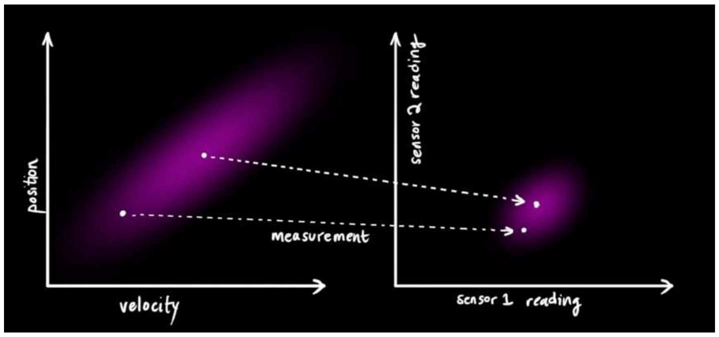
In other words, the new best estimate is a prediction made from previous best estimate, plus a correction for known external influences.

And the new uncertainty is predicted from the old uncertainty, with some additional uncertainty from the environment.

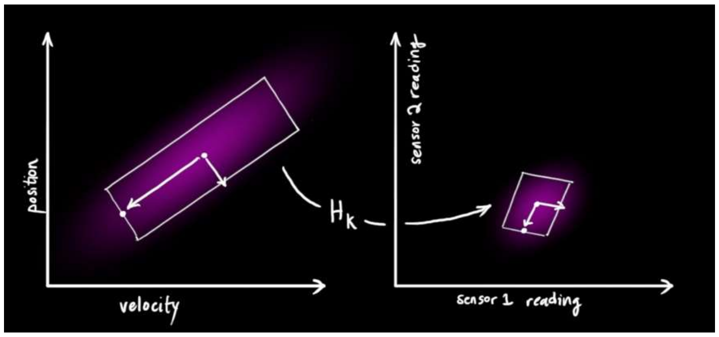
All right, so that's easy enough. We have a fuzzy estimate of where our system might be, given by $\hat{\mathbf{x}}\,k$ and $\mathbf{P}k$. What happens when we get some data from our sensors?

Refining the estimate with measurements

We might have several sensors which give us information about the state of our system. For the time being it doesn't matter what they measure; perhaps one reads position and the other reads velocity. Each sensor tells us something indirect about the state— in other words, the sensors operate on a state and produce a set of readings.



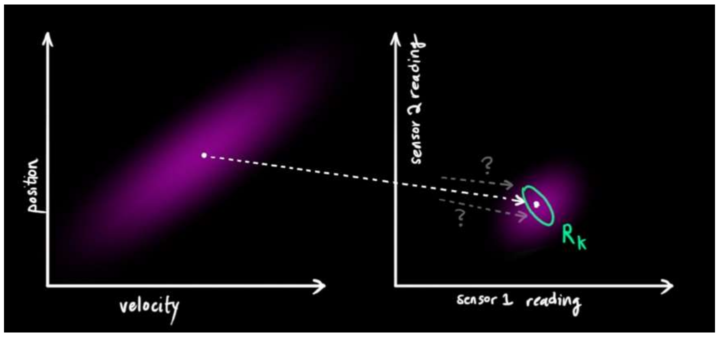
Notice that the units and scale of the reading might not be the same as the units and scale of the state we're keeping track of. You might be able to guess where this is going: We'll model the sensors with a matrix, **H***k*.



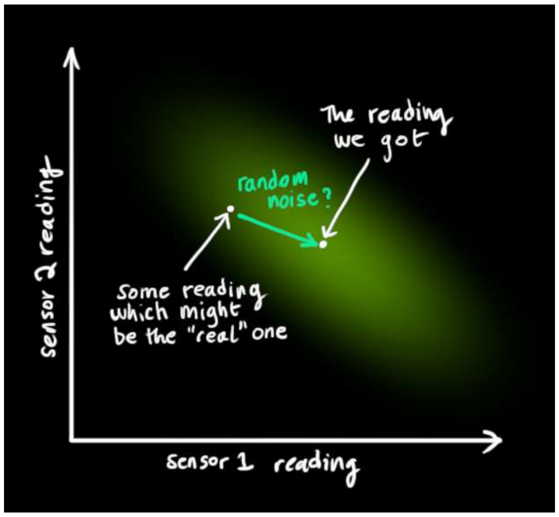
We can figure out the distribution of sensor readings we'd expect to see in the usual way:

$$\begin{aligned}\vec{\mu}_{\text{expected}} &= \mathbf{H}_k \hat{\mathbf{x}}_k \\ \mathbf{\Sigma}_{\text{expected}} &= \mathbf{H}_k \mathbf{P}_k \mathbf{H}_k^T\end{aligned}$$

One thing that Kalman filters are great for is dealing with *sensor noise*. In other words, our sensors are at least somewhat unreliable, and every state in our original estimate might result in a *range* of sensor readings.

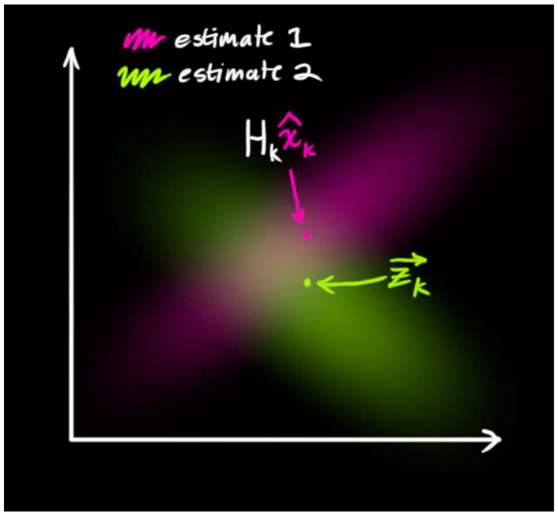


From each reading we observe, we might guess that our system was in a particular state. But because there is uncertainty, some states are more likely than others to have have produced the reading we saw:



We'll call the covariance of this uncertainty (i.e. of the sensor noise) **R***k*. The distribution has a mean equal to the reading we observed, which we'll call **z***k*→.

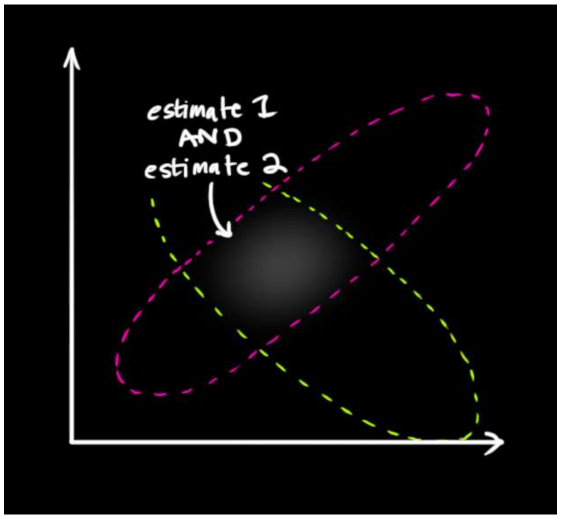
So now we have two Gaussian blobs: One surrounding the mean of our transformed prediction, and one surrounding the actual sensor reading we got.



We must try to reconcile our guess about the readings we'd see based on the predicted state (pink) with a *different* guess based on our sensor readings (green) that we actually observed.

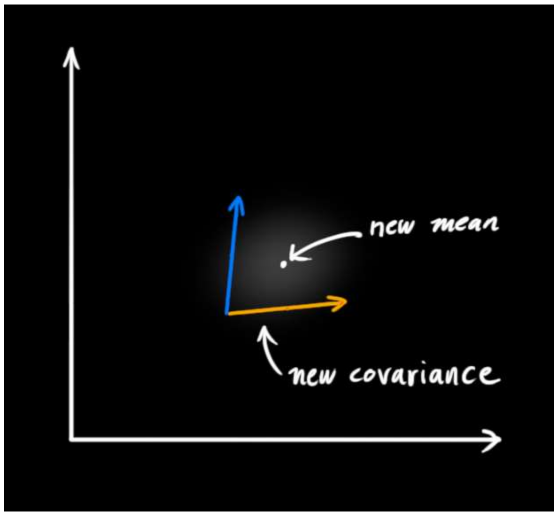
So what's our new most likely state? For any possible reading (*z*1,*z*2), we have two associated probabilities: (1) The probability that our sensor reading **z***k*→ is a (mis-)measurement of $(z1,z2)$, and (2) the probability that our previous estimate thinks $(z1,z2)$ is the reading we should see.

If we have two probabilities and we want to know the chance that *both* are true, we just multiply them together. So, we take the two Gaussian blobs and multiply them:



What we're left with is the overlap, the region where *both* blobs are bright/likely. And it's a lot more precise than either of our previous estimates. The mean of this distribution is the configuration for which both estimates are most likely, and is therefore the best guess of the true configuration given all the information we have.

Hmm. This looks like another Gaussian blob.



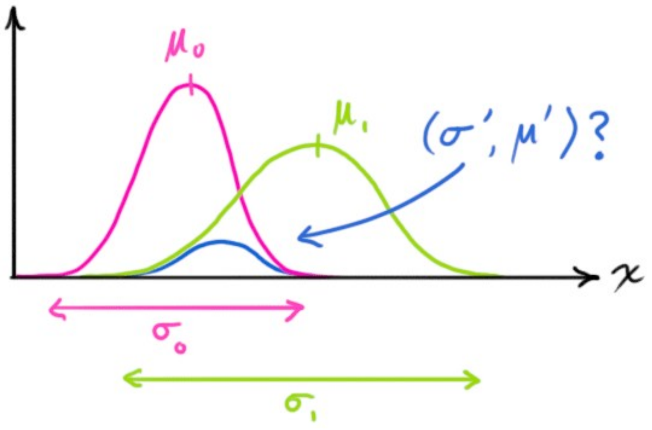
As it turns out, when you multiply two Gaussian blobs with separate means and covariance matrices, you get a *new* Gaussian blob with its own mean and covariance matrix! Maybe you can see where this is going: There's got to be a formula to get those new parameters from the old ones!

Combining Gaussians

Let's find that formula. It's easiest to look at this first in one dimension. A 1D Gaussian bell curve with variance $\sigma 2$ and mean μ is defined as:

$$\mathcal{N}(x, \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

We want to know what happens when you multiply two Gaussian curves together. The blue curve below represents the (unnormalized) intersection of the two Gaussian populations:



$$\mathcal{N}(x, \mu_0, \sigma_0) \cdot \mathcal{N}(x, \mu_1, \sigma_1) \stackrel{?}{=} \mathcal{N}(x, \mu', \sigma')$$

You can substitute equation (9) into equation (10) and do some algebra (being careful to renormalize, so that the total probability is 1) to obtain:

$$\mu' = \mu_0 + \frac{\sigma_0^2(\mu_1 - \mu_0)}{\sigma_0^2 + \sigma_1^2}$$
$$\sigma'^2 = \sigma_0^2 - \frac{\sigma_0^4}{\sigma_0^2 + \sigma_1^2}$$

We can simplify by factoring out a little piece and calling it **k**:

$$\mathbf{k} = \frac{\sigma_0^2}{\sigma_0^2 + \sigma_1^2}$$

$$\mu' = \mu_0 + \mathbf{k}(\mu_1 - \mu_0)$$
$$\sigma'^2 = \sigma_0^2 - \mathbf{k}\sigma_0^2$$

Take note of how you can take your previous estimate and add something to make a new estimate. And look at how simple that formula is!

But what about a matrix version? Well, let's just re-write equations (12) and (13) in matrix form. If Σ is the covariance matrix of a Gaussian blob, and $\vec{\mu}$ its mean along each axis, then:

$$\mathbf{K} = \Sigma_0(\Sigma_0 + \Sigma_1)^{-1}$$

$$\vec{\mu}' = \vec{\mu_0} + \mathbf{K}(\vec{\mu_1} - \vec{\mu_0})$$
$$\Sigma' = \Sigma_0 - \mathbf{K}\Sigma_0$$

K is a matrix called the Kalman gain, and we'll use it in just a moment.

Easy! We're almost finished!

Putting it all together

We have two distributions: The predicted measurement with $(\mu_0, \Sigma_0) = (\mathbf{H}_k\hat{\mathbf{x}}_k, \mathbf{H}_k\mathbf{P}_k\mathbf{H}_k^T)$, and the observed measurement with ($\mu 1$,$\Sigma 1$)=($\mathbf{z}k\rightarrow$,$\mathbf{R}k$). We can just plug these into equation (15) to find their overlap:

$$\begin{aligned}\mathbf{H}_k\hat{\mathbf{x}}_k' &= \mathbf{H}_k\hat{\mathbf{x}}_k && +\mathbf{K}(\vec{\mathbf{z}_k}-\mathbf{H}_k\hat{\mathbf{x}}_k)\\ \mathbf{H}_k\mathbf{P}_k'\mathbf{H}_k^T &= \mathbf{H}_k\mathbf{P}_k\mathbf{H}_k^T && -\mathbf{K}\mathbf{H}_k\mathbf{P}_k\mathbf{H}_k^T\end{aligned}$$

And from (14), the Kalman gain is:

$$\mathbf{K} = \mathbf{H}_k\mathbf{P}_k\mathbf{H}_k^T(\mathbf{H}_k\mathbf{P}_k\mathbf{H}_k^T + \mathbf{R}_k)^{-1}$$

We can knock an $\mathbf{H}k$ off the front of every term in (16) and (17) (note that one is hiding inside $\mathbf{K}$), and an $\mathbf{H}Tk$ off the end of all terms in the equation for $\mathbf{P}'k$.

$$\begin{aligned}\hat{\mathbf{x}}_k' &= \hat{\mathbf{x}}_k+\mathbf{K}'(\vec{\mathbf{z}_k}-\mathbf{H}_k\hat{\mathbf{x}}_k)\\ \mathbf{P}_k' &= \mathbf{P}_k-\mathbf{K}'\mathbf{H}_k\mathbf{P}_k\end{aligned}$$

$$\mathbf{K}' = \mathbf{P}_k\mathbf{H}_k^T(\mathbf{H}_k\mathbf{P}_k\mathbf{H}_k^T + \mathbf{R}_k)^{-1}$$

…giving us the complete equations for the update step.

And that's it! $\hat{\mathbf{x}}'k$ is our new best estimate, and we can go on and feed it (along with $\mathbf{P}'k$) back into another round of predict or update as many times as we like.

Conclusions

This paper has summarized the research efforts made over the past two decades about the application and the digital implementation of KFs in a significant number of industrial fields. In summary, one of the main issues of this recursive state estimator was the computational load requirement. Therefore, two research directions have been mainly investigated. The first one, which started in the 1970s, focused on factorization methods and fast algorithms. This paper was primarily motivated by aerospace applications. The second approach, which appeared later, focused on the design and implementation of highly sophisticated numerical architectures embedded on FPGAs. Nowadays, the integration of KFs or variants of the KF (e.g., UKFs) into industrial systems is not very widespread for two main reasons, i.e., the complexity of the algorithm compared with the classical Luenberger observers and the computational load requirement to be embedded on a low computational power processor. However, due to the availability of new lowcost and highly elaborate processors (such as floating-point DSPs targeted at real-time process control applications and system-on-chips), the KF is likely to spread more and more and still has a bright future ahead of it.

References:

1- Ghysels, Eric; Marcellino, Massimiliano (2018), Applied Economic Forecasting using Time Series Methods. New York, NY: Oxford University Press. p. 419.

2- Humpherys, Jeffrey (2012), "A Fresh Look at the Kalman Filter". Society for Industrial and Applied Mathematics. 54 (4): 801–823.

3- Kalman, R. E. (1960), "A New Approach to Linear Filtering and Prediction Problems". Journal of Basic Engineering. 82: 35–45.

4- N. Bergman, "Recursive Bayesian estimation (1999),: Navigation and tracking applications," Ph.D. dissertation, Linköping Univ., Linköping, Sweden.

5- RodrÌguez, A., and Ruiz, E. (2012), íBootstrap prediction mean squared errors of unobserved states based on the Kalman Ölter with estimated parametersí, Computational Statistics and Data Analysis, 56:62-74.

6- Rojas, A.J. (2011), íOn the discrete-time algebraic Riccati equation and its solution in closed-formí, Preprints of the 18th IFAC World Congress, Milan.

7- Simon D (2006), Optimal State Estimation: Kalman, H Infinity, and Nonlinear Approaches. Oxford: John Wiley & Sons;

8- Simon Julier and Jeffrey Uhlmann (1997), A new extension of the kalman filter to nonlinear systems. Int. Symp. Aerospace/Defense Sensing, Simul. And Controls, Orlando, FL.

9- Wolpert, Daniel; Ghahramani, Zoubin (2000), "Computational principles of movement neuroscience". Nature Neuroscience. 3: 1212–7.

10- Zarchan Paul; (2000), Fundamentals of Kalman Filtering: A Practical Approach. American Institute of Aeronautics and Astronautics, Incorporated.

11- Quenneville, B., and Singh (2000), íBayesian prediction mean squared error for state space models with estimated parametersí, Journal of Time Series Analysis, 21:219-236.