Kalman Filters for Software Engineers

A deep dive into Kalman Filters, one of the most widespread and useful algorithms of all times.

Lorenzo Peppoloni Jan 27, 2020 · 12 min read ★

Photo by Clem Onojeghuo on Unsplash

Get started Open in app

(KF) are one of the most widespread algorithms in the world (if you look around your house, 80% of the tech you have probably has some sort of KF running inside), let's try and make them clear once and for all.

By the end of this post you will have an intuitive and detailed understanding of how a KF works, what's the idea behind it, why you need multiple variants and what are its most common ones.

. . .

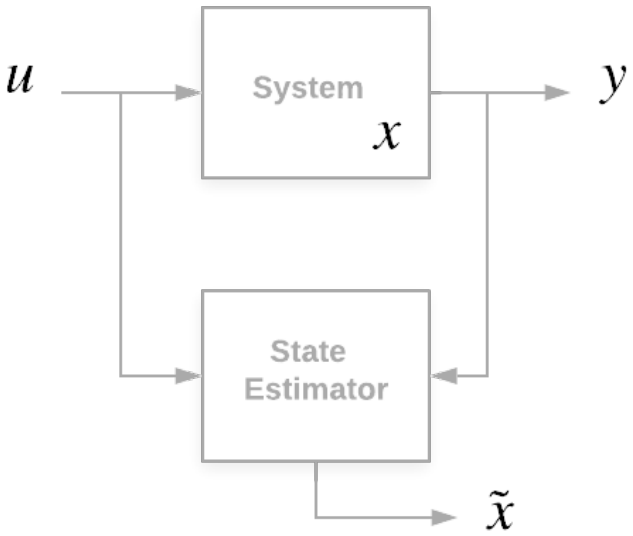
State Estimation

KFs are part of what is called State Estimation algorithms. What's state estimation? Imagine you have a system (let's treat it as a black box). This black box can be anything: your fan, a chemical system, a mobile robot. For each of these systems we can define a state. A state is a vector of variables that we care to know and that can describe the "state" (here's why it's called state) in which the system is at a specific point in time. What does "can describe" means? It means that if you know the state vector at time `k` and the input given to the system, you can know (using some sort knowledge of the system workings as well) the state of the system at time `k+1`.

For example, let's say we have a moving robot and we care about knowing its position (and we don't care about its orientation) in space. If we define the state to be the robot position `(x, y)` and its velocities `(v_x, v_y)` and we have a model of how the robot moves, that would be enough to pinpoint where the robot is and where it will be at the next time instant.

So a state estimation algorithm estimates the state of a system. Why do you want to estimate it? Because in real-life scenarios the real state of the system is never accessible by an external observer. Usually, there are two cases: you can measure the state but measurements are affected by noise (each sensor can only produce readings up to a certain precision which might not be enough for you) or you just cannot directly measure the state. An example could be computing the position of the aforementioned mobile robot using GPS (we decided position to be part of the state), that would give you probably up to 10 meters of measurement error which is probably not enough for any application you can think of.

it is also affected by a certain measurement noise. From this, we define a state estimator as a system which takes in the input and the output of the system you want to estimate the state of and outputs an estimation of the system state.
Traditionally, the state is indicated with `x`, the outputs with `y` or `z`, `u` is the input and `tilde_x` is the estimate state.



System and State Estimator block diagram.

Kalman Filters

As you may have noticed, we already discussed a bit about errors:

- you can measure the output of the system, but you have a measurement error given by your sensor
- you can estimate the state, but being it an estimation it has a certain level of confidence.

In addition to that, I said that you need some sort of knowledge of the system, you need to know a model of the "behaviour" of the system (more on this later), your

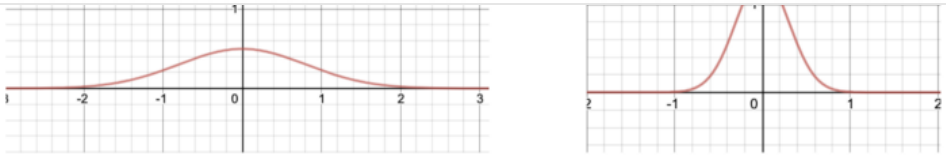
in KFs, you treat all these uncertainties using Gaussian distributions. A Gaussian distribution is a nice way of representing something you are not really sure about. Your current belief can be represented by the mean of the distribution, while the standard deviation will say how confident you are in your belief.

In a KF:

- your estimated state will be a Gaussian random variable with a certain mean and covariance (which will tell us how much the algorithm "trusts" its current estimation)
- your uncertainty on the measurements of the output of the original system will be represented with a random variable with mean 0 and a certain covariance (which will tell us how much we trust the measurement itself)
- your uncertainty of your system model will be represented with a random variable with mean 0 and a certain covariance (which will tell us how much we trust the model we are using).

Let's make some examples to understand what's the idea behind this.

1. **Bad model, good sensors**
   Let's assume again you want to track the position of a robot and you spent a lot of money in your sensors and they give you cm-precision accuracy. On the other hand, you are not into Robotics at all, you googled a bit and you found a very basic motion model: the random walk (basically a particle whose motion is just given by noise). It is pretty clear that your model is not really good and cannot really be trusted, while your measurements are really good. In this case, you are probably going to model your measurements noise with a very narrow Gaussian distribution (small variance), while your model uncertainty with a very wide Gaussian distribution (big variance).
2. **Bad sensors, good model**
   The opposite stands if you have bad sensors (e.g., GPS) but you spent lots of time modelling your system. In that case, you are probably going to model your model uncertainty with a very narrow Gaussian distribution (small variance), while your measurement noise with a very wide Gaussian distribution (big variance).



Wide variance vs. small variance Gaussian distributions.

What about the estimated state uncertainty?
The KF will take care of updating it according to what is happening during the estimation, the only thing you have to do is initializing it to a good-enough value. "Good enough" depends on your application, your sensors, your model, etc… In general, KF takes a bit to converge to the right estimation.

How does KF work?

As we said, for KF to work you need to have "some knowledge" about the system (an "uncertain", aka not perfect, model). In particular with KF you need two models:

- **the state transition model**: some function that, given the state and the input at a time `k`, gives you the state at time `k+1`.

$$x(k+1) = f(x(k), u(k))$$

- **the measurement model**: some function that, given the state at a time `k`, gives you the measurement for that same time instant

$$y(k) = h(x(k))$$

Later, we'll see why we need these functions, let's first look at some examples to understand what they mean.

State transition model
This model tells you how your system evolves in time (if you remember, earlier we touched on how a state has to be descriptive enough to infer the system behaviour in

Get started Open in app

moving objects (if measured at a decent sampling rate), you can use constant velocity models, which assume the object is moving with constant speed, for vehicles, you can use the unicycle model, etc… Let's assume that, one way or another, we got to a model. We are making an important assumption here, that is necessary for the KF to work: your current state only depends on the precedent. In other words, the "history" of the system state is condensed in the previous state, that is to say each state is independent from the past, given the precedent state. This is also known as **Markov assumption**. If this doesn't hold, you cannot express the current state in terms of the precedent one alone.

Measurement model
The measurement model tells you how output (which you can measure) and state are tied together. Intuitively, you need this because you know the measured output and you want to infer the state from it during the estimation. Again, this model changes from case to case. For example, in the mobile robot example, if you care about the position and you have GPS, your model is the identity function, because you are already measuring a noisy version of the state.

The mathematical formulation and explanation of each step follows:

$$\tilde{x}^-(k) = f(\tilde{x}(k-1)) \quad (1)$$
$$\tilde{y}(k) = h(\tilde{x}^-(k)) \quad (2)$$
$$\tilde{x}(k) = \tilde{x}^-(k) + K(y(k) - \tilde{y}(k)) \quad (3)$$

So how does the KF actually work? The algorithm works in two steps called predict and update. Let's assume we are at time `k` and we have our estimated state at that time. First, we use the state transition model and we make the estimated state evolve to the next time instant `(1)`. This is equivalent to saying: given my current belief about the state, the input I have and my knowledge about the system, I expect my next state to be this. This is the predict step.

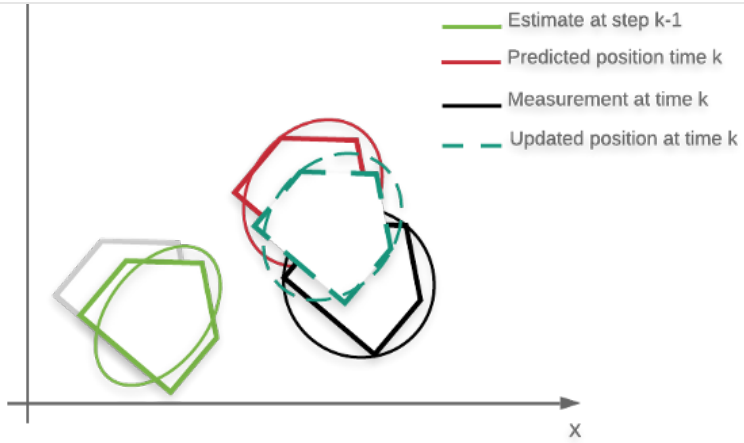
expected state, we compute the output (with the measurement model) `(2)` and we compare it with the real measured output. We then use the difference between the two in a "smart way" to correct our estimation of the state `(3)`.

Usually, we indicate with apex `-`, the estimation of the state coming from the predict step, before the correction. *K* is called the Kalman Gain. That's where the smartness really is: the *K* depends on how much we trust the measurement, how much we trust our current estimation (which depends on how much we trust the model) and according to this information *K* "decides" how much the predicted estimation is corrected with the measurement. If our measurement noise is "small" compared to how much we are trusting our estimation coming from the predict step, we are going to correct the estimation a lot using the measurement, if the reverse is true, we are going to correct it minimally.

A note: for the sake of simplicity, I wrote the equations as if we were dealing with normal variables, but you have to consider that at each step we're dealing with random Gaussian variables, thus we need to propagate through the functions also the covariances of the variables, not only the means.

Let's exemplify. Let's imagine that we are tracking the position of a robot (again). The real position is shown in grey, at time `k` we believe that the robot is in the position in green, with an estimation covariance represented as an ellipse (if you are not familiar with this type of representation have a look here). Roughly speaking, you can see by the shape of the ellipse, that our filter is at this step more "confident" about the lateral positioning, compared to the positioning in the direction of forward motion. After the predict step, where we let the system evolve using the state transition model, we think that the new position is the one in red. Since the ellipse grew bigger in the lateral direction, we are now less certain of the new estimated position (for example because we don't trust the model much). Then we read our GPS and we get the position in black. With the update step, the actual position estimation will be the dark green dotted one. The estimation will be closer to the red if we trust the model more (lower covariance compared to the measurement noise covariance) or closer to the measurement if we trust the measurement more (lower noise measurement covariance compared to the model uncertainty).

Get started Open in app



Example of real position and estimation at each step of the KF algorithm.

Families of KFs

KFs can be classified in two big families according to the type of models (state transition and measurement) they use: if the models are linear you have a Linear Kalman Filter, while if they are nonlinear you have Nonlinear Kalman Filters.

Why the distinction? Well, KFs assume that your variables are Gaussian, when passed through a linear function a Gaussian variable remains a Gaussian variable, this is not true if you pass it through a nonlinear function. This breaks the Kalman assumption, thus we need to find ways to fix it.

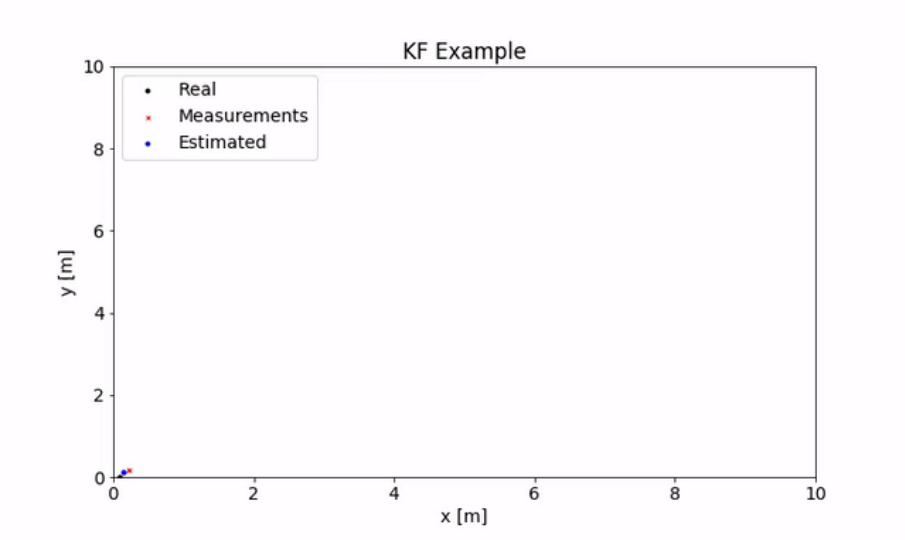
Historically, people have found two main approaches: cheating with the model and cheating with the data. If you cheat with the model you basically linearize the nonlinear functions around your current estimation, in that way you brought yourself back to the linear case that works. This approach is called **Extended Kalman Filter** (**EKF**). The main disadvantage of this method is that you have to be able to compute the Jacobians of `f()` and `h()`. Alternatively, if you cheat with the data, you use your nonlinear functions, but then you try and "Gaussianize" (if that word even exists) the distributions that you made non Gaussian. This is done via a smart sampling technique, called Unscented Transform. This transformation allows you to describe (approximately) a distribution in terms of mean and covariance (only Gaussian

Unscented Transform gives a better approximation of the resulting distributions compared to the approximation you get linearizing the models. In practice, you have to have quite heavy nonlinearities to actually see big differences.

KF in action

Since I spoke so much about a mobile robot with GPS, I made a short demo about this case (you can find the code here if you want to play with it). The robot motion is generated using a unicycle model. The state transition model used for the KF is a constant velocity model, with a state containing the x and y positions, the steering angle and their derivatives.

The robot is moving in time (the real positions shown in black), at each step you get a very noisy GPS measurement, which gives the `x` and `y` (in red) and estimates the position (blue). You can play with the different parameters and see how they affect the state estimation. As you can see, we can take very noisy measurements and get a nice estimation of the real positions.



KF in action: a robot real path (black) is tracked with a KF (blue) from noisy measurements (red).

Bonus: intuitive meaning of the Kalman Gain

Get started Open in app

$$K = P_k^{-1} C^T (C P_k C^T + R)^{-1} = \frac{P_k C^T}{C P_k C^T + R}$$

where `P_k` is the covariance of the current estimated state (how confident we are about the estimation), `C` is the linear transformation for the measurement model such that `y(k) = Cx(k)` and R is the covariance matrix of the measurement noise. Note that the fraction notation is not really correct but makes visualizing what happens easier.

From the equation, if R becomes 0, we have:

$$K \simeq C^{-1}$$
$$\tilde{x}(k) = x^- + C^{-1}(y(k) - C\tilde{x}^-) = C^{-1}y(k) \qquad (3)$$

Substituting in what we defined step `(3)` of the algorithm, we can see that we will completely disregard the predict step results and we use the inverse transformation of the measurement model to obtain a state estimation coming only from the measurements.

If instead we trust the model/estimation a lot, `P_k` will tend to 0, giving:

$$K \simeq 0$$
$$\tilde{x}(k) = x^- \qquad (3)$$

Thus we have a final estimation that is the same as the predict step output.

It is to be noted that I'm using "trusting the model" and "trusting the current estimation" interchangeably. They are not the same but they are related, since how

the estimation of the previous step of filtering.

Bonus 2: Libraries

There are a bunch of nice libraries to compute KFs online, here are some of my favourites.

Being a GO enthusiast, I'll start with this very nice GO library with several pre-implemented models:

rosshemsley/kalman

A package implementing Kalman filtering and smoothing for continuous time-indexed models with non-uniform time...

github.com

For Python, you can have a look at https://pykalman.github.io/.

. . .

Conclusions: We had an in-depth look at what state estimation is, how Kalman Filters work, what's the intuition behind them, how to use them and when. We introduced a toy (but real life) problem and saw how you can solve it with a Kalman Filter. Then, we had a more in-depth look at what the Kalman Filter actually does under the hood.

Cheers!

Get started Open in app

Kalman Filter | Robotics | Data Science | Tech | Technology