Bzarg

phi{losophy,sics,ction}

How a Kalman filter works, in pictures

I have to tell you about the Kalman filter, because what it does is pretty damn amazing.

Surprisingly few software engineers and scientists seem to know about it, and that makes me sad because it is such a general and powerful tool for **combining information** in the presence of uncertainty. At times its ability to extract accurate information seems almost magical— and if it sounds like I'm talking this up too much, then take a look at this previously posted video where I demonstrate a Kalman filter figuring out the *orientation* of a free-floating body by looking at its *velocity*. Totally neat!

What is it?

You can use a Kalman filter in any place where you have **uncertain information** about some dynamic system, and you can make an **educated guess** about what the system is going to do next. Even if messy reality comes along and interferes with the clean motion you guessed about, the Kalman filter will often do a very good job of figuring out what actually happened. And it can take advantage of correlations between crazy phenomena that you maybe wouldn't have thought to exploit!

Kalman filters are ideal for systems which are **continuously changing**. They have the advantage that they are light on memory (they don't need to keep any history other than the previous state), and they are very fast, making them well suited for real time problems and embedded systems.

The math for implementing the Kalman filter appears pretty scary and opaque in most places you find on Google. That's a bad state of affairs, because

the Kalman filter is actually super simple and easy to understand if you look at it in the right way. Thus it makes a great article topic, and I will attempt to illuminate it with lots of clear, pretty pictures and colors. The prerequisites are simple; all you need is a basic understanding of probability and matrices.

I'll start with a loose example of the kind of thing a Kalman filter can solve, but if you want to get right to the shiny pictures and math, feel free to jump ahead.

What can we do with a Kalman filter?

Let's make a toy example: You've built a little robot that can wander around in the woods, and the robot needs to know exactly where it is so that it can navigate.

We'll say our robot has a state $\vec{x_k}$, which is just a position and a velocity:

$$\vec{x_k} = (\vec{p}, \vec{v})$$

Note that the state is just a list of numbers about the underlying configuration of your system; it could be anything. In our example it's position and velocity, but it could be data about the amount of fluid in a tank, the temperature of a car engine, the position of a user's finger on a touchpad, or any number of things you need to keep track of.

Our robot also has a GPS sensor, which is accurate to about 10 meters, which is good, but it needs to know its location more precisely than 10 meters.

There are lots of gullies and cliffs in these woods, and if the robot is wrong by more than a few feet, it could fall off a cliff. So GPS by itself is not good enough.

We might also know something about how the robot moves: It knows the commands sent to the wheel motors, and its knows that if it's headed in one direction and nothing interferes, at the next instant it will likely be further along that same direction. But of course it doesn't know everything about its motion: It might be buffeted by the wind, the wheels might slip a little bit, or roll over bumpy terrain; so the amount the wheels have turned might not exactly represent how far the robot has actually traveled, and the prediction won't be perfect.

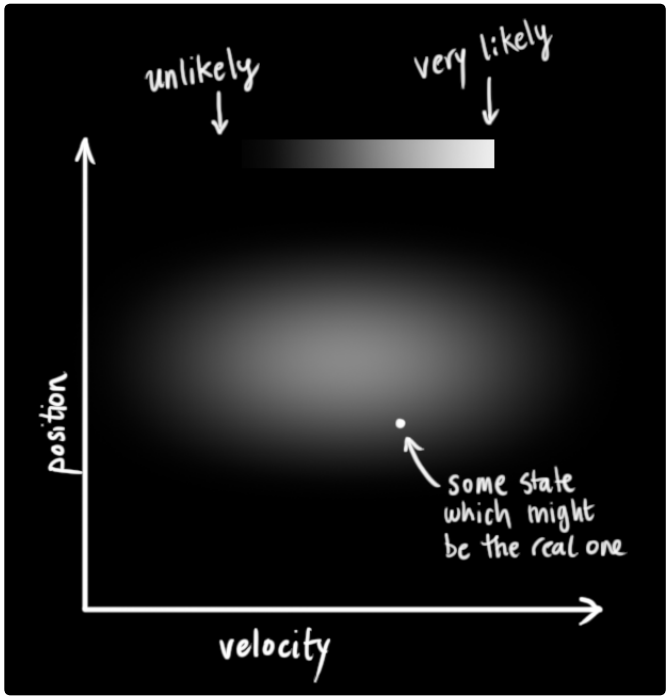
The GPS **sensor** tells us something about the state, but only indirectly, and with some uncertainty or inaccuracy. Our **prediction** tells us something about how the robot is moving, but only indirectly, and with some uncertainty or inaccuracy.

But if we use all the information available to us, can we get a better answer than **either estimate would give us by itself**? Of course the answer is yes, and that's what a Kalman filter is for.

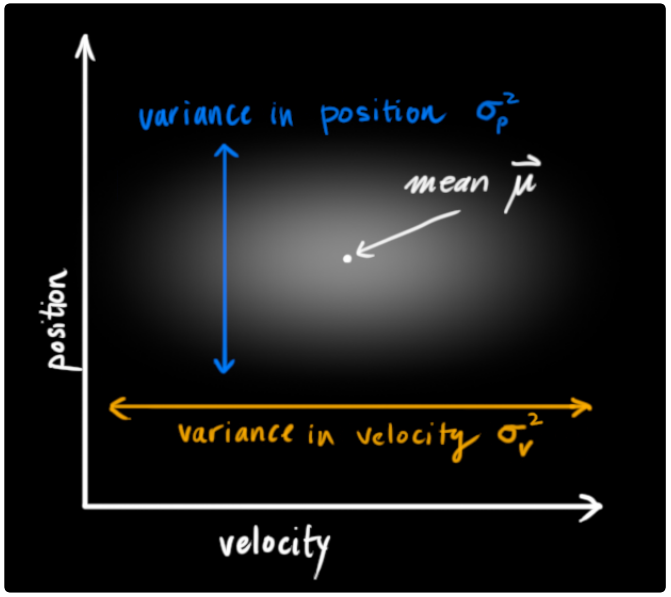
How a Kalman filter sees your problem

Let's look at the landscape we're trying to interpret. We'll continue with a simple state having only position and velocity. $\vec{x} = \begin{bmatrix} p\\ v \end{bmatrix}$

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:

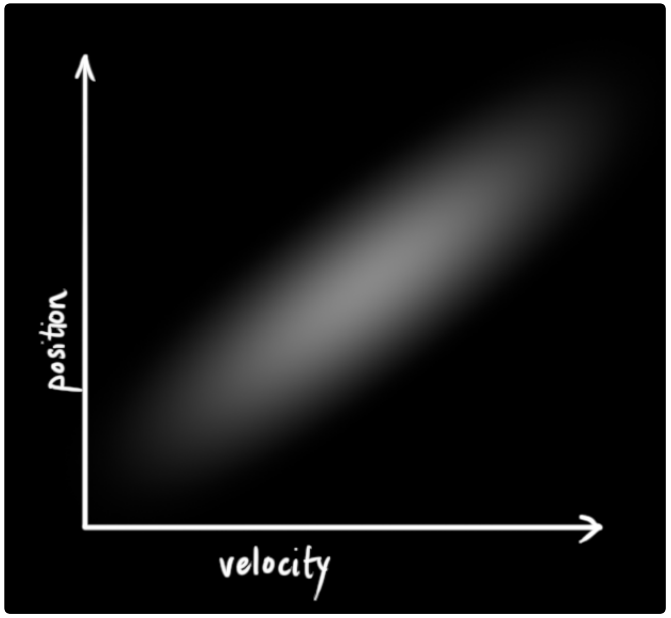


The Kalman filter assumes that both variables (postion and velocity, in our case) are random and *Gaussian distributed.* Each variable has a **mean** value μ, which is the center of the random distribution (and its most likely state), and a **variance** σ^2, which is the uncertainty:



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.

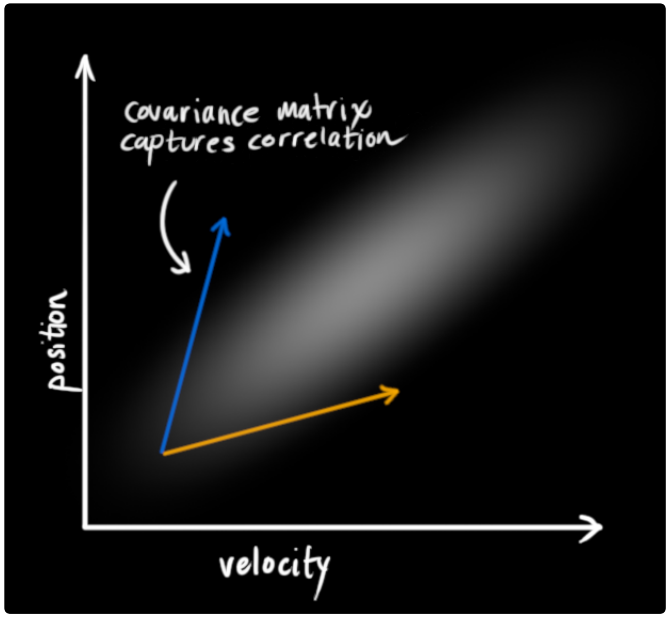
The example below shows something more interesting: Position and velocity are **correlated**. The likelihood of observing a particular position depends on what velocity you have:



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 \Sigma_{ij} 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 "\mathbf{\Sigma}", so we call their elements "\Sigma_{ij}".

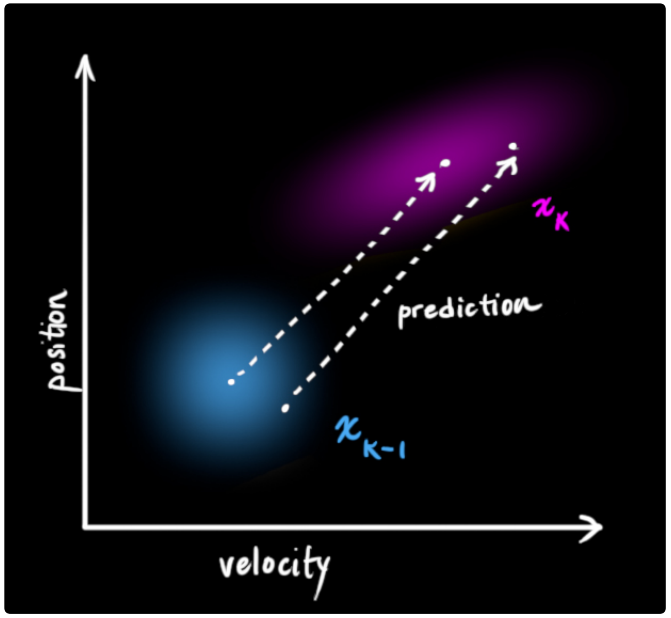


Describing the problem with matrices

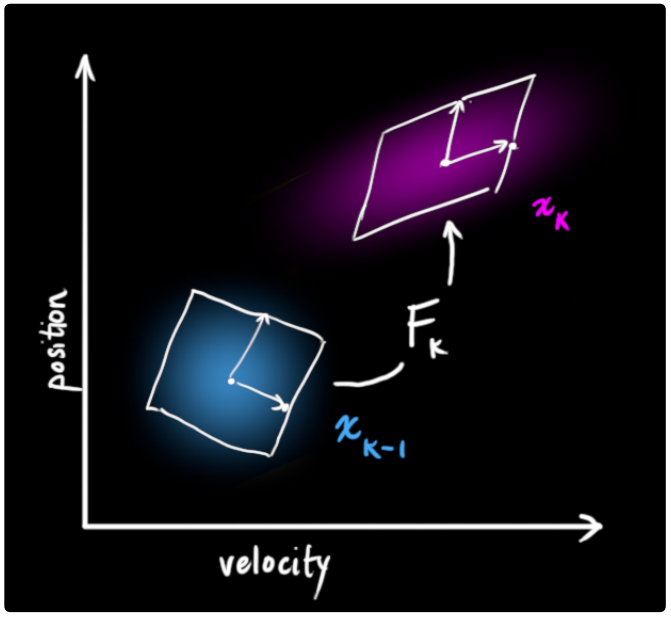
We're modeling our knowledge about the state as a Gaussian blob, so we need two pieces of information at time k: We'll call our best estimate $\mathbf{\hat{x}_k}$ (the mean, elsewhere named μ), and its covariance matrix $\mathbf{P_k}$. $$\begin{aligned} \mathbf{\hat{x}}_k &= \begin{bmatrix} \text{position}\\ \text{velocity} \end{bmatrix}\\ \mathbf{P}_k &= \begin{bmatrix} \Sigma_{pp} & \Sigma_{pv} \\ \Sigma_{vp} & \Sigma_{vv} \\ \end{bmatrix} \end{aligned}$$

(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, \mathbf{F_k}:



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: $$\begin{split} \color{deeppink}{p_k} &= \color{royalblue}{p_{k-1}} + \Delta t &\color{royalblue}{v_{k-1}} \\ \color{deeppink}{v_k} &= &\color{royalblue}{v_{k-1}} \end{split}$$ In other words: $$\begin{align} \color{deeppink}{\mathbf{\hat{x}}_k} &= \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix} \color{royalblue}{\mathbf{\hat{x}}_{k-1}} \\ &= \mathbf{F}_k \color{royalblue}{\mathbf{\hat{x}}_{k-1}} \label{statevars} \end{align}$$

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 $\color{firebrick}{\mathbf{A}}$, then what happens to its covariance matrix Σ?

Well, it's easy. I'll just give you the identity: $$\begin{equation} \begin{split} Cov(x) &= \Sigma\\ Cov(\color{firebrick}{\mathbf{A}}x) &= \color{firebrick}{\mathbf{A}} \Sigma \color{firebrick}{\mathbf{A}}^T \end{split} \label{covident} \end{equation}$$

So combining $\eqref{covident}$ with equation $\eqref{statevars}$: $$\begin{equation} \begin{split} \color{deeppink}{\mathbf{\hat{x}}_k} &= \mathbf{F}_k \color{royalblue}{\mathbf{\hat{x}}_{k-1}} \\ \color{deeppink}{\mathbf{P}_k} &= \mathbf{F_k} \color{royalblue}{\mathbf{P}_{k-1}} \mathbf{F}_k^T \end{split} \end{equation}$$

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 $\color{darkorange}{\vec{\mathbf{u}}_k}$, do something with it, and add it to our prediction as a correction.

Let's say we know the expected acceleration $\color{darkorange}{a}$ due to the throttle setting or control commands. From basic kinematics we get: $\begin{split} \color{deeppink}{p_k} &= \color{royalblue}{p_{k-1}} + {\Delta t} &\color{royalblue}{v_{k-1}} + &\frac{1}{2} \color{darkorange}{a} {\Delta t}^2 \\ \color{deeppink}{v_k} &= &\color{royalblue}{v_{k-1}} + & \color{darkorange}{a} {\Delta t} \end{split}$ In matrix form: $\begin{equation} \begin{split} \color{deeppink}{\mathbf{\hat{x}}_k} &= \mathbf{F}_k \color{royalblue}{\mathbf{\hat{x}}_{k-1}} + \begin{bmatrix} \frac{\Delta t^2}{2} \\ \Delta t \end{bmatrix} \color{darkorange}{a} \\ &= \mathbf{F}_k \color{royalblue}{\mathbf{\hat{x}}_{k-1}} + \mathbf{B}_k \color{darkorange}{\vec{\mathbf{u}_k}} \end{split} \end{equation}$

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

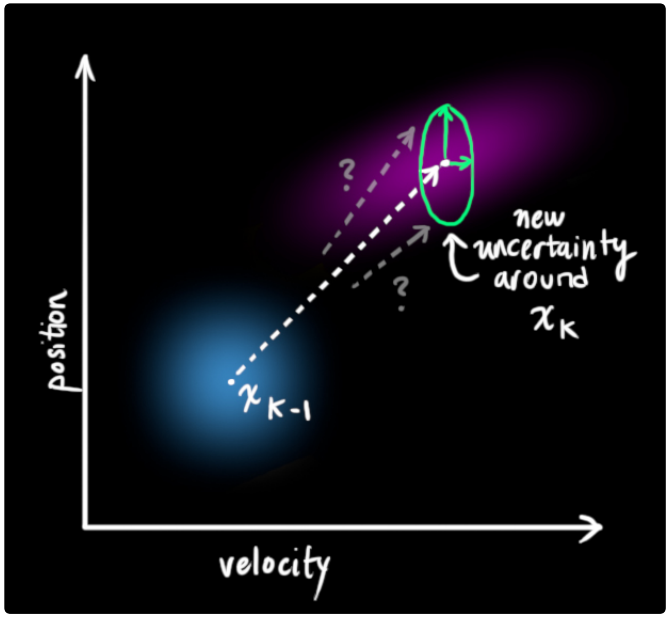
Let's add one more detail. What happens if our prediction is not a 100% accurate model of what's actually going on?

External uncertainty

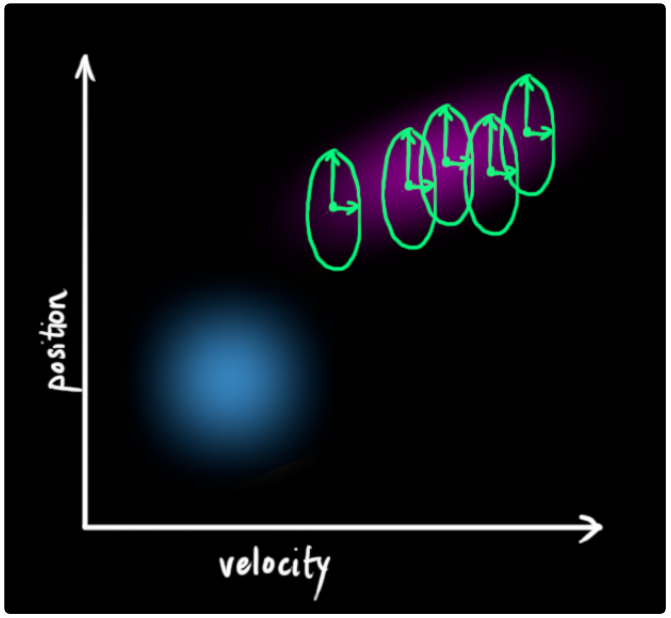
Everything is fine if the state evolves based on its own properties. Everything is *still* fine if the state evolves based on external forces, so long as we know what those external forces are.

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.

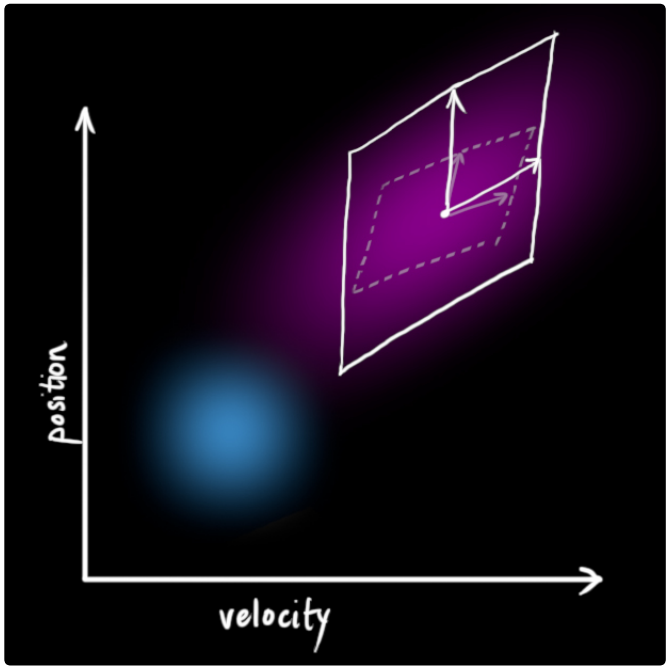
We can model the uncertainty associated with the "world" (i.e. things we aren't keeping track of) by adding some new uncertainty after every prediction step:



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 $\color{royalblue}{\mathbf{\hat{x}}_{k-1}}$ is moved to somewhere inside a Gaussian blob with covariance $\color{mediumaquamarine}{\mathbf{Q}_k}$. Another way to say this is that we are treating the untracked influences as **noise** with covariance $\color{mediumaquamarine}{\mathbf{Q}_k}$.



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



We get the expanded covariance by simply **adding** {\color{mediumaquamarine}{\mathbf{Q}_k}}, giving our complete expression for the **prediction step**: \begin{equation} \begin{split} \color{deeppink}{\mathbf{\hat{x}}_k} &= \mathbf{F}_k \color{royalblue}{\mathbf{\hat{x}}_{k-1}} + \mathbf{B}_k \color{darkorange}{\vec{\mathbf{u}_k}} \\ \color{deeppink}{\mathbf{P}_k} &= \mathbf{F_k} \color{royalblue}{\mathbf{P}_{k-1}} \mathbf{F}_k^T + \color{mediumaquamarine}{\mathbf{Q}_k} \end{split} \label{kalpredictfull} \end{equation}

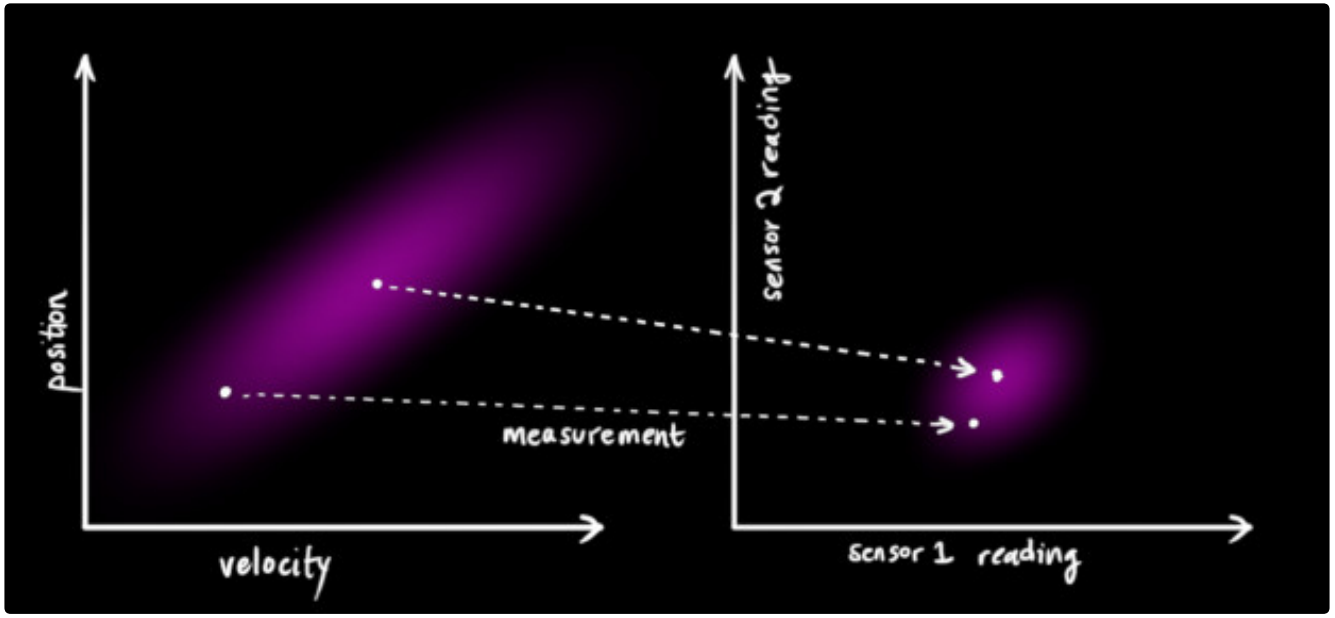
In other words, the **new best estimate** is a **prediction** made from **previous best estimate**, plus a **correction** for **known external influences**.

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

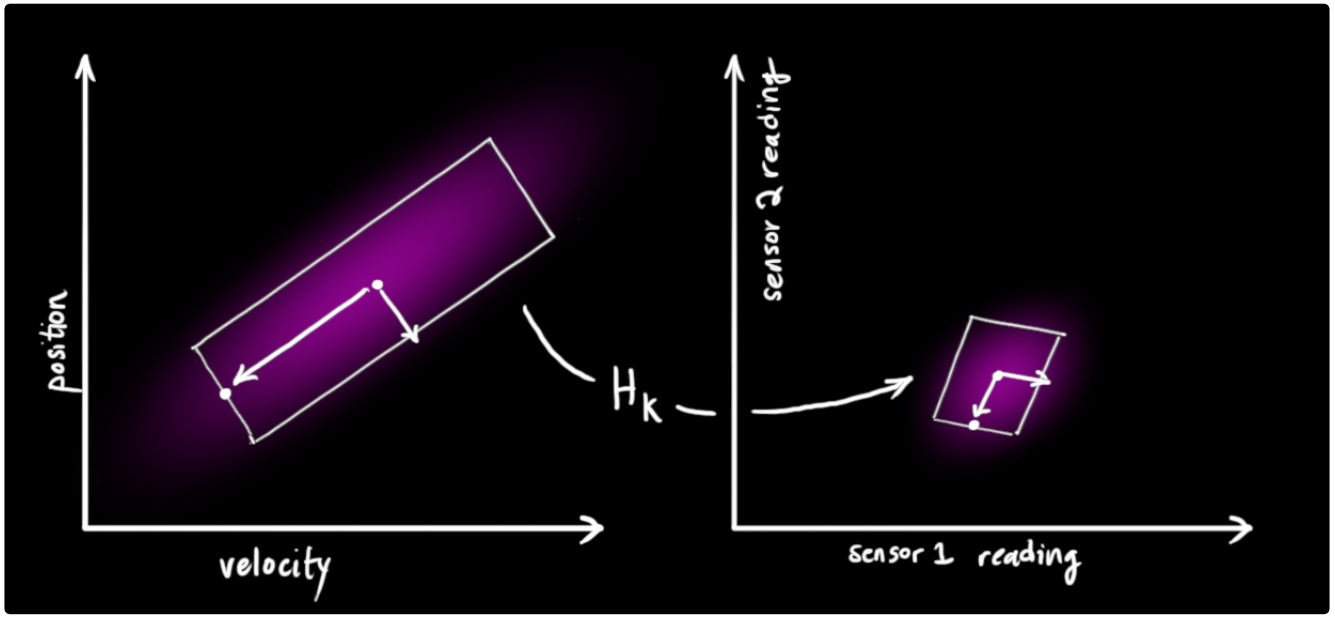
All right, so that's easy enough. We have a fuzzy estimate of where our system might be, given by \color{deeppink}{\mathbf{\hat{x}}_k} and \color{deeppink}{\mathbf{P}_k}. What happens when we get some data from our sensors?

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**.

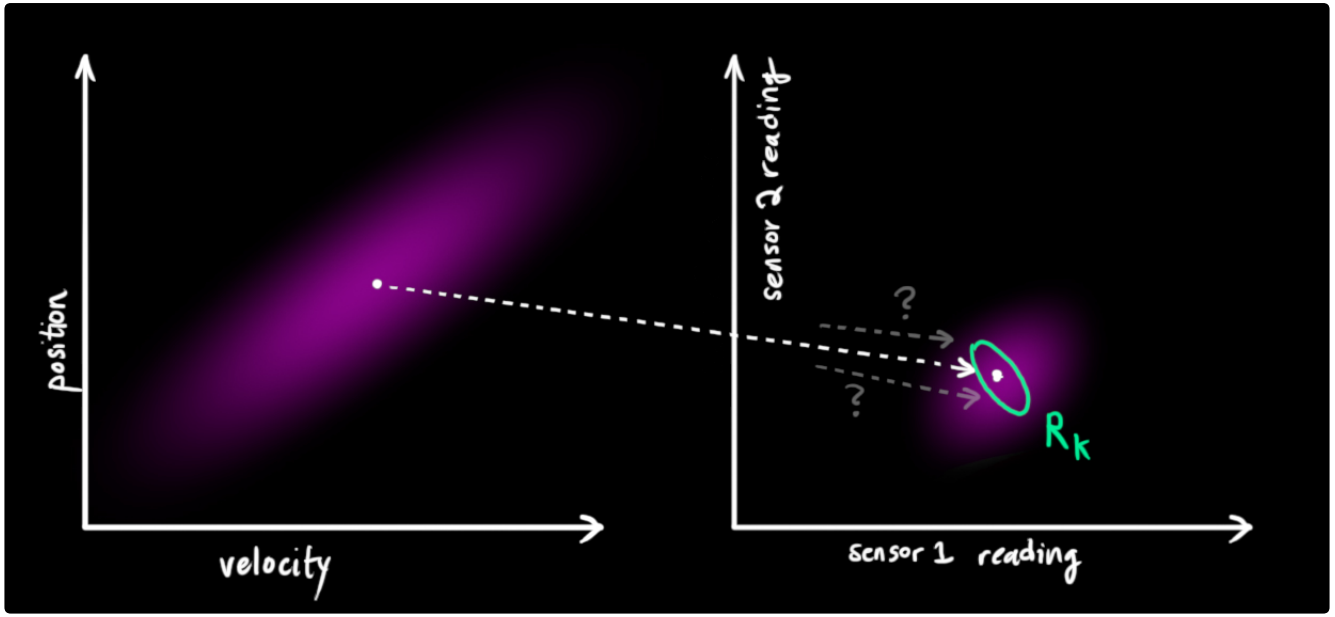


Notice that the units and scale of the reading might not be the same as the units and scale of the state we're keeping track of. You might be able to guess where this is going: We'll model the sensors with a matrix, $\mathbf{H}_k$.

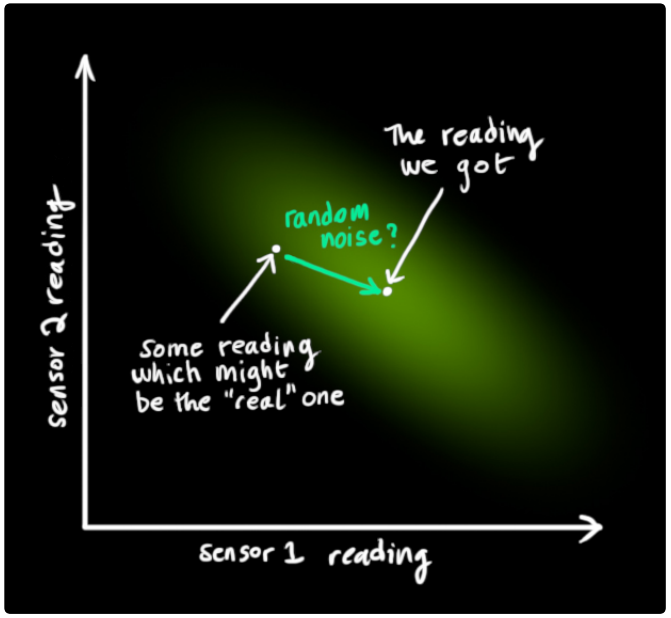


We can figure out the distribution of sensor readings we'd expect to see in the usual way: $$\begin{aligned} \vec{\mu}_{\text{expected}} &= \mathbf{H}_k \color{deeppink}{\mathbf{\hat{x}}_k} \\ \mathbf{\Sigma}_{\text{expected}} &= \mathbf{H}_k \color{deeppink}{\mathbf{P}_k} \mathbf{H}_k^T \end{aligned}$$

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.

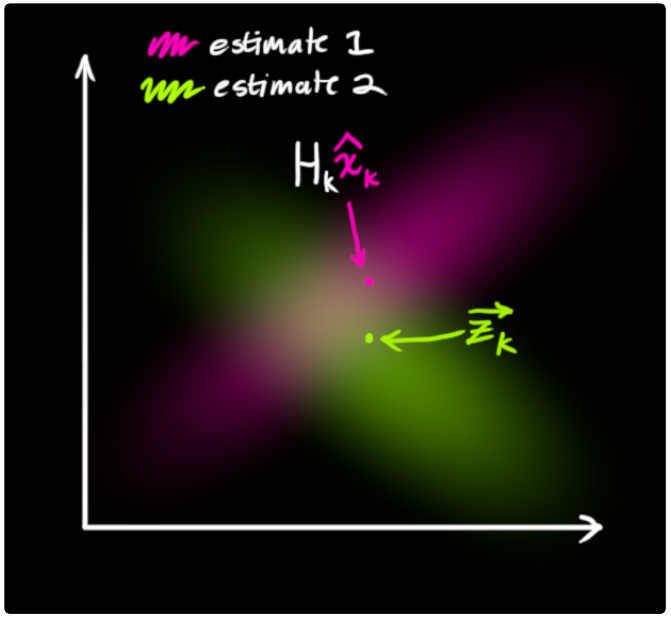


From each reading we observe, we might guess that our system was in a particular state. But because there is uncertainty, **some states are more likely than others** to have have produced the reading we saw:



We'll call the **covariance** of this uncertainty (i.e. of the sensor noise) \color{mediumaquamarine}{\mathbf{R}_k}. The distribution has a **mean** equal to the reading we observed, which we'll call \color{yellowgreen}{\vec{\mathbf{z}_k}}.

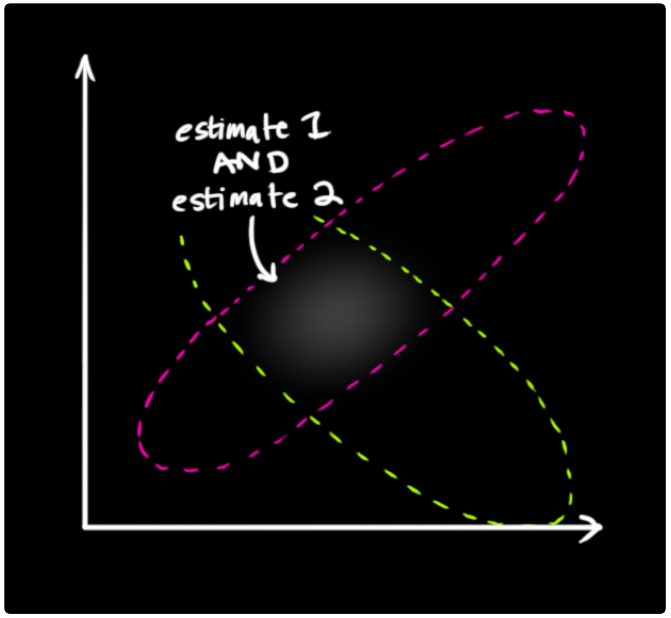
So now we have two Gaussian blobs: One surrounding the mean of our transformed prediction, and one surrounding the actual sensor reading we got.



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.

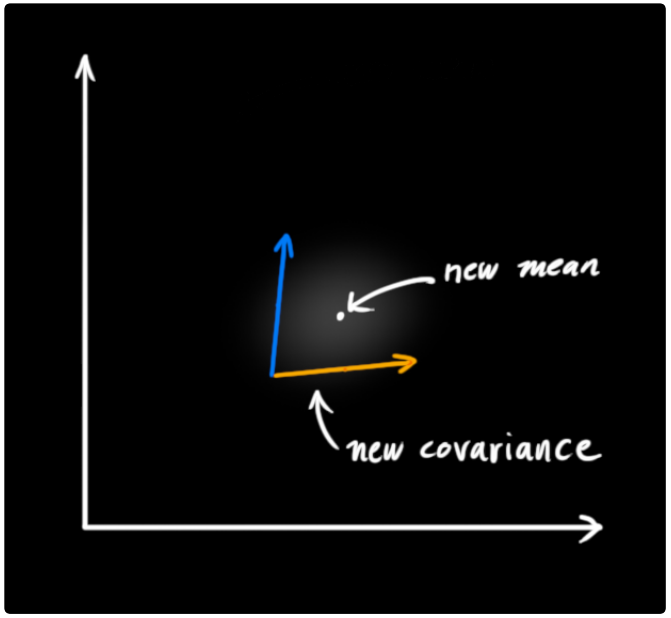
So what's our new most likely state? For any possible reading (z_1,z_2), we have two associated probabilities: (1) The probability that our sensor reading \color{yellowgreen}{\vec{\mathbf{z}_k}} is a (mis-)measurement of (z_1,z_2), and (2) the probability that our previous estimate thinks (z_1,z_2) is 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.



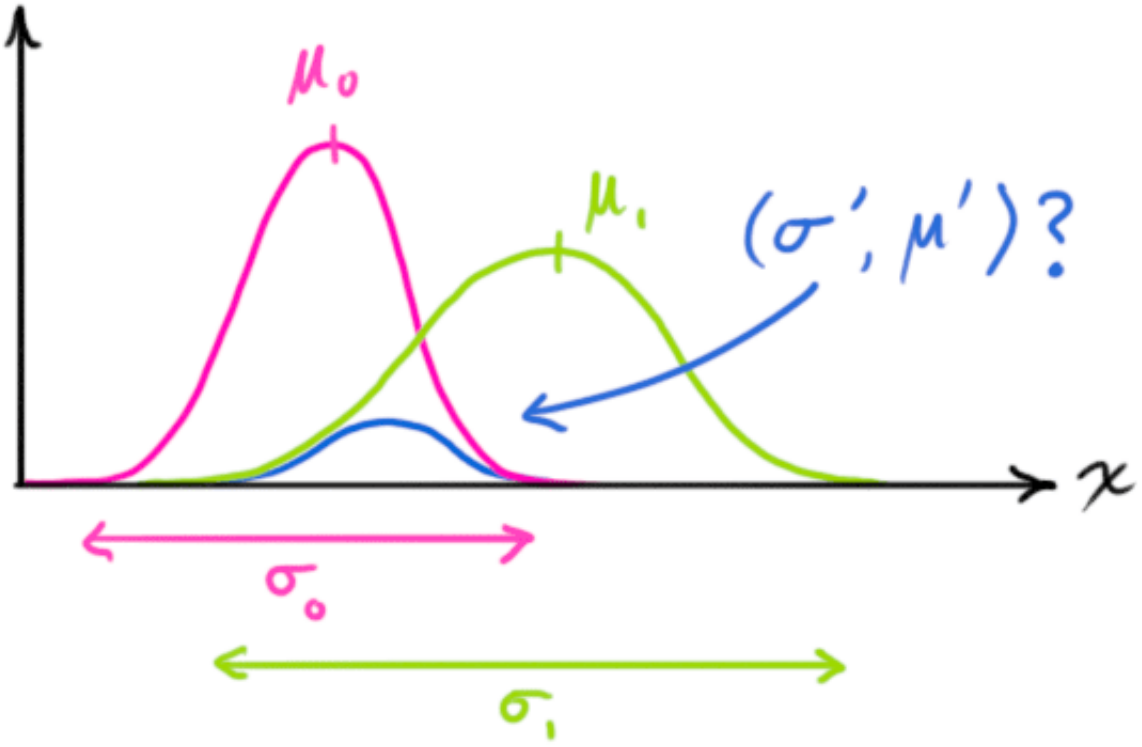
As it turns out, when you multiply two Gaussian blobs with separate means and covariance matrices, you get a *new* Gaussian blob with its **own** mean and covariance matrix! Maybe you can see where this is going: There's got to be a formula to get those new parameters from the old ones!

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 σ^2 and mean μ is defined as:

$$\begin{equation} \label{gaussformula} \mathcal{N}(x, \mu,\sigma) = \frac{1}{ \sigma \sqrt{ 2\pi } } e^{ -\frac{ (x – \mu)^2 }{ 2\sigma^2 } } \end{equation}$$

We want to know what happens when you multiply two Gaussian curves together:



$$\begin{equation} \label{gaussequiv} \mathcal{N}(x, \color{fuchsia}{\mu_0}, \color{deeppink}{\sigma_0}) \cdot \mathcal{N}(x, \color{yellowgreen}{\mu_1}, \color{mediumaquamarine}{\sigma_1}) \stackrel{?}{=} \mathcal{N}(x, \color{royalblue}{\mu'}, \color{mediumblue}{\sigma'}) \end{equation}$$

You can substitute equation \eqref{gaussformula} into equation \eqref{gaussequiv} and do some algebra (being careful to renormalize, so that the total probability is 1) to obtain: $$\begin{equation} \label{fusionformula} \begin{aligned} \color{royalblue}{\mu'} &= \mu_0 + \frac{\sigma_0^2 (\mu_1 - \mu_0)}{\sigma_0^2 + \sigma_1^2}\\ \color{mediumblue}{\sigma'}^2 &= \sigma_0^2 - \frac{\sigma_0^4}{\sigma_0^2 + \sigma_1^2} \end{aligned} \end{equation}$$

We can simplify by factoring out a little piece and calling it \color{purple}{\mathbf{k}}: $$\begin{equation} \label{gainformula} \color{purple}{\mathbf{k}} = \frac{\sigma_0^2}{\sigma_0^2 + \sigma_1^2} \end{equation}$$ $$\begin{equation} \begin{split} \color{royalblue}{\mu'} &= \mu_0 + &\color{purple}{\mathbf{k}} (\mu_1 - \mu_0)\\ \color{mediumblue}{\sigma'}^2 &= \sigma_0^2 - &\color{purple}{\mathbf{k}} \sigma_0^2 \end{split} \label{update} \end{equation}$$

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 \eqref{gainformula} and \eqref{update} in matrix form. If \Sigma is the covariance matrix of a Gaussian blob, and \vec{\mu} its mean along each axis, then: \begin{equation} \label{matrixgain} \color{purple}{\mathbf{K}} = \Sigma_0 (\Sigma_0 + \Sigma_1)^{-1} \end{equation} \begin{equation} \begin{split} \color{royalblue}{\vec{\mu}'} &= \vec{\mu_0} + &\color{purple}{\mathbf{K}} (\vec{\mu_1} – \vec{\mu_0})\\ \color{mediumblue}{\Sigma'} &= \Sigma_0 – &\color{purple}{\mathbf{K}} \Sigma_0 \end{split} \label{matrixupdate} \end{equation}

\color{purple}{\mathbf{K}} is a matrix called the **Kalman gain**, and we'll use it in just a moment.

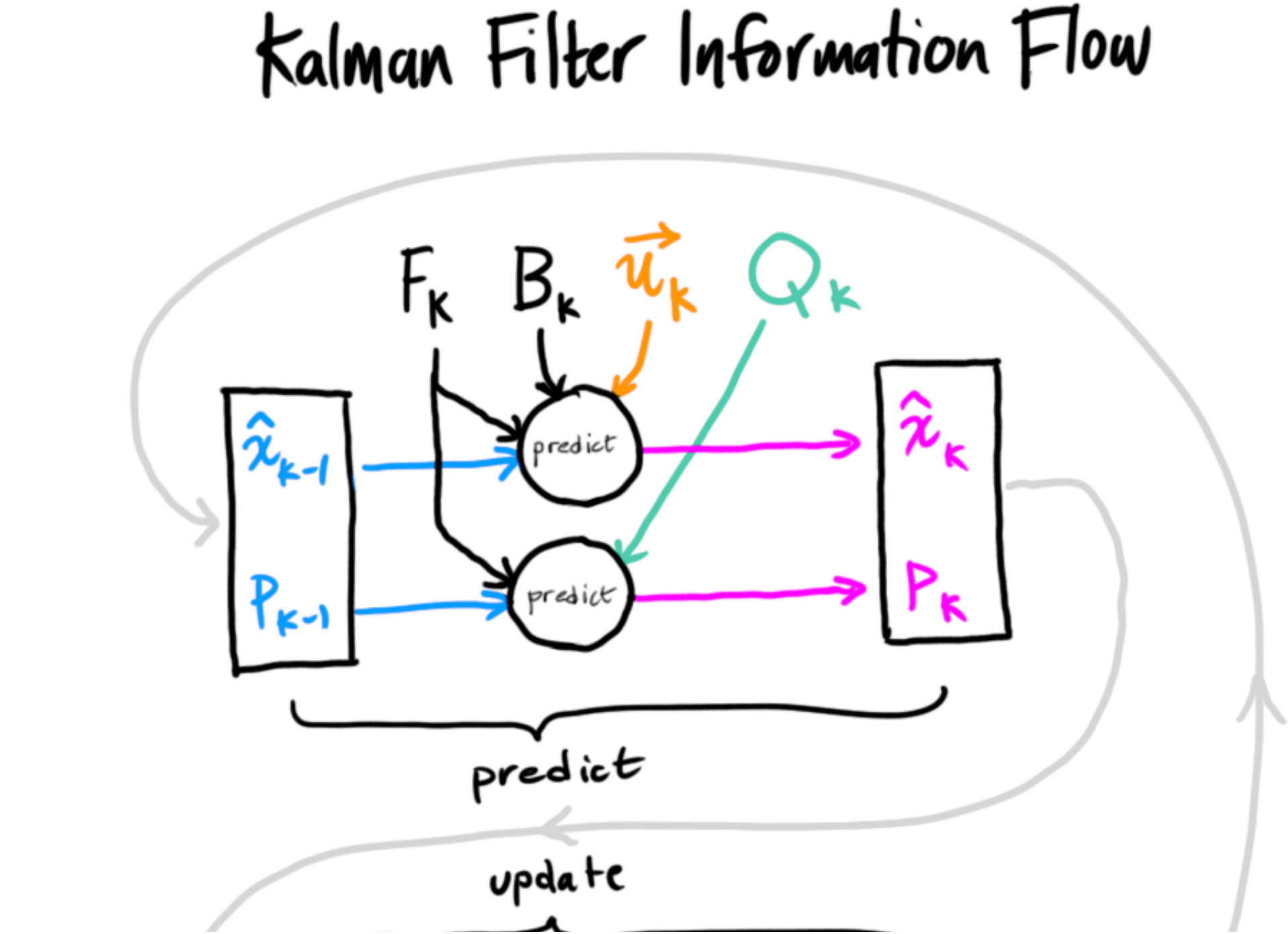
Easy! We're almost finished!

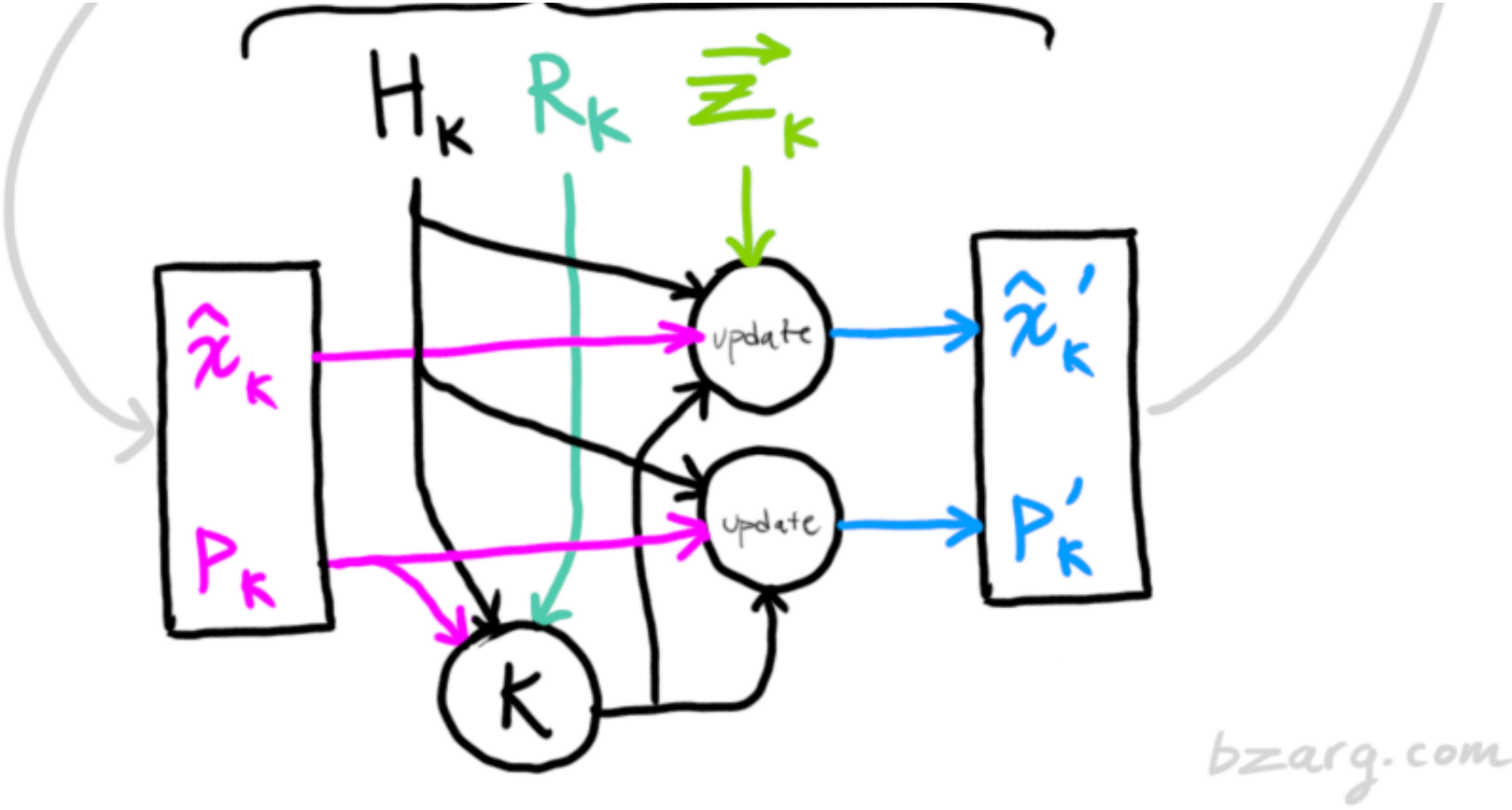
Putting it all together

We have two distributions: The predicted measurement with (\color{fuchsia}{\mu_0}, \color{deeppink}{\Sigma_0}) = (\color{fuchsia}{\mathbf{H}_k \mathbf{\hat{x}}_k}, \color{deeppink}{\mathbf{H}_k \mathbf{P}_k \mathbf{H}_k^T}) , and the observed measurement with (\color{yellowgreen}{\mu_1}, \color{mediumaquamarine}{\Sigma_1}) = (\color{yellowgreen}{\vec{\mathbf{z}_k}}, \color{mediumaquamarine}{\mathbf{R}_k}). We can just plug these into equation \eqref{matrixupdate} to find their overlap: \begin{equation} \begin{aligned} \mathbf{H}_k \color{royalblue}{\mathbf{\hat{x}}_k'} &= \color{fuchsia}{\mathbf{H}_k \mathbf{\hat{x}}_k} & + & \color{purple}{\mathbf{K}} (\color{yellowgreen}{\vec{\mathbf{z}_k}} – \color{fuchsia}{\mathbf{H}_k \mathbf{\hat{x}}_k}) \\ \mathbf{H}_k \color{royalblue}{\mathbf{P}_k'} \mathbf{H}_k^T &= \color{deeppink}{\mathbf{H}_k \mathbf{P}_k \mathbf{H}_k^T} & – & \color{purple}{\mathbf{K}} \color{deeppink}{\mathbf{H}_k \mathbf{P}_k \mathbf{H}_k^T} \end{aligned} \label{kalunsimplified} \end{equation} And from \eqref{matrixgain}, the Kalman gain is: \begin{equation} \label{eq:kalgainunsimplified} \color{purple}{\mathbf{K}} = \color{deeppink}{\mathbf{H}_k \mathbf{P}_k \mathbf{H}_k^T} (\color{deeppink}{\mathbf{H}_k \mathbf{P}_k \mathbf{H}_k^T} + \color{mediumaquamarine}{\mathbf{R}_k})^{-1} \end{equation} We can knock an \mathbf{H}_k off the front of every term in \eqref{kalunsimplified} and \eqref{eq:kalgainunsimplified} (note that one is hiding inside \color{purple}{\mathbf{K}}), and an \mathbf{H}_k^T off the end of all terms in the equation for \color{royalblue}{\mathbf{P}_k'}. \begin{equation} \begin{split} \color{royalblue}{\mathbf{\hat{x}}_k'} &= \color{fuchsia}{\mathbf{\hat{x}}_k} & + & \color{purple}{\mathbf{K}'} (\color{yellowgreen}{\vec{\mathbf{z}_k}} – \color{fuchsia}{\mathbf{H}_k \mathbf{\hat{x}}_k}) \\ \color{royalblue}{\mathbf{P}_k'} &= \color{deeppink}{\mathbf{P}_k} & – & \color{purple}{\mathbf{K}'} \color{deeppink}{\mathbf{H}_k \mathbf{P}_k}

\end{split} \label{kalupdatefull} \end{equation} \begin{equation} \color{purple}{\mathbf{K}'} = \color{deeppink}{\mathbf{P}_k \mathbf{H}_k^T} (\color{deeppink}{\mathbf{H}_k \mathbf{P}_k \mathbf{H}_k^T} + \color{mediumaquamarine}{\mathbf{R}_k})^{-1} \label{kalgainfull} \end{equation} ... giving us the complete equations for the **update step.**

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


Kalman Filter Information Flow
F_k
B_k
$\vec{u}_k$
Q_k
$\hat{x}_{k-1}$
P_{k-1}
predict
predict
$\hat{x}_k$
P_k
predict
update



Wrapping up

Of all the math above, all you need to implement are equations \eqref{kalpredictfull}, \eqref{kalupdatefull}, and \eqref{kalgainfull}. (Or if you forget those, you could re-derive everything from equations \eqref{covident} and \eqref{matrixupdate}.)

This will allow you to model any linear system accurately. For nonlinear systems, we use the **extended Kalman filter**, which works by simply linearizing the predictions and measurements about their mean. (I may do a second write-up on the EKF in the future).

If I've done my job well, hopefully someone else out there will realize how cool these things are and come up with an unexpected new place to put

them into action.

Some credit and referral should be given to this fine document, which uses a similar approach involving overlapping Gaussians. More in-depth derivations can be found there, for the curious.

Share this article:

This entry was posted in Mini-courses, Programming on August 11, 2015 [http://www.bzarg.com/p/how-a-kalman-filter-works-in-pictures/] .

89 thoughts on “How a Kalman filter works, in pictures”

Valen
August 11, 2015 at 5:51 pm

Great article! Loving your other posts as well.

Camilla
August 11, 2015 at 8:42 pm

Wow! This article is amazing. Thank you very much for putting in the time and effort to produce this.

george
August 11, 2015 at 9:34 pm

This is a nice and straight forward explanation .
Thanks.

sobhan
January 5, 2016 at 3:04 pm

I do agree...that is so great and I find it interesting and I will do it in other places .....and mention your name dude.....thanks a lot.

Justin
August 11, 2015 at 9:35 pm

Hey Tim what did you use to draw this illustration?
http://www.bzarg.com/wp-content/uploads/2015/08/kalflow.png

tbabb Post author
August 13, 2015 at 12:07 am

All the illustrations are done primarily with Photoshop and a stylus.

Philip Tellis
August 11, 2015 at 10:30 pm

Nice explanation. Looks like someone wrote a Kalman filter implementation in Julia: https://github.com/wkearn/Kalman.jl

Paul Masurel
August 11, 2015 at 11:22 pm

Great post! Keep up the good work!

Ben Jackson
August 12, 2015 at 12:10 am

Nice work! I literally just drew half of those covariance diagrams on a whiteboard for someone. Now I can just direct everyone to your page.

Maxime Caron
August 12, 2015 at 12:10 am

Great post. Keep up the good work! I am hoping for the Extended Kalman filter soon.

Sebastian
August 12, 2015 at 1:49 am

What happens if your sensors only measure one of the state variables. Do you just make the H matrix to drop the rows you don't have sensor data for and it all works out?

Raj Samant
December 12, 2015 at 5:22 pm

You reduce the rank of H matrix, omitting row will not make Hx multiplication possible. If in above example only position is measured state u make H = [1 0; 0 0]. If both are measurable then u make H = [1 0; 0 1];

Jim Hejl
August 12, 2015 at 2:09 am

great write up! Thx!!

John Shea
August 12, 2015 at 2:19 am

Very nice, but are you missing squares on those variances in (1)?

Ilya Kavalerov
August 12, 2015 at 2:34 am

Nice post!

Near 'You can use a Kalman filter in any place where you have uncertain information' shouldn't there be a caveat that the 'dynamic system' obeys the markov property? I.e. a process where given the present, the future is independent of the past (not true in financial data for example).

Also just curious, why no references to hidden markov models, the Kalman filter's discrete (and simpler) cousin?

Jan Galkowski
August 12, 2015 at 2:10 pm

Don’t know if this question was answered, but, yes, there is a Markovian assumption in the model, as well as an assumption of linearity. But, at least in my technical opinion, that sounds much more restrictive than it actually is in practice. If the system (or “plant”) changes its internal “state” smoothly, the linearization of the Kalman is nothing more than using a local Taylor expansion of that state behavior, and, to some degree, a faster rate of change can be compensated for by increasing sampling rate. As far as the Markovian assumption goes, I think most models which are not Markovian can be transformed into alternate models which are Markovian, using a change in variables and such.

tom
August 12, 2015 at 3:24 am

Awesome post! I’m impressed.

Kalen
August 12, 2015 at 5:57 am

Aaaargh!
I wish I’d known about these filters a couple years back – they would have helped me solve an embedded control problem with lots of measurement uncertainty.

Thanks for the great post!

Mike McRoberts
August 12, 2015 at 7:33 am

Great article and very informative. Love the use of graphics. I would love to see another on the ‘extended Kalman filter’.

Thanks,

Mike

Pedrow
August 12, 2015 at 10:54 am

Great article, thanks!

It would be great if you could repeat all the definitions just after equations (18) and (19) – I found myself constantly scrolling up & down because I couldn’t remember what z was, etc. etc.

Greg Yaks
August 12, 2015 at 12:20 pm

Just before equation (2), the kinematics part, shouldn't the first equation be about p_k rather than x_k, i.e., position and not the state?

santaraxita
August 12, 2015 at 9:10 pm

This is an excellent piece of pedagogy. Every step in the exposition seems natural and reasonable. I just chanced upon this post having the vaguest idea about Kalman filters but now I can pretty much derive it. The only thing I have to ask is whether the control matrix/vector must come from the second order terms of the taylor expansion or is that a pedagogical choice you made as an instance of external influence? Also, I guess in general your prediction matrices can come from a one-parameter group of diffeomorphisms.

tbabb Post author
August 13, 2015 at 12:09 am

Nope, using acceleration was just a pedagogical choice since the example was using kinematics. The control matrix need not be a higher order Taylor term; just a way to mix "environment" state into the system state.

Jai
August 12, 2015 at 9:30 pm

I wish there were more posts like this. That explain how amazing and simple ideas are represented by scary symbols. Loved the approach. Can you please do one on Gibbs Sampling/Metropolis Hastings Algorithm as well?

Eric O. LEBIGOT
August 12, 2015 at 10:31 pm

Very nice write up! The use of colors in the equations and drawings is useful.

Small nitpick: an early graph that shows the uncertainties on x should say that sigma is the *standard deviation*, not the “variance”.

tbabb Post author
August 12, 2015 at 11:12 pm

@Eric Lebigot: Ah, yes, the diagram is missing a ‘squared’ on the sigma symbols. I’ll fix that when I next have access to the source file for that image.

Nico Galoppo
August 12, 2015 at 11:34 pm

Eye opening. The only part I didn’t follow in the derivation, is where the left hand side of (16) came from… until I realized that you defined x’_k and P’_k in the true state space coordinate system, not in the measurement coordinate system – hence the use of H_k!

Antti
August 13, 2015 at 6:19 am

Hmm, I didn't think this through yet, but don't you need to have a pretty good initial guess for your orientation (in the video example) in order for the future estimates to be accurate? Please show this is not so :)

Great illustration and nice work! Thanks!

Antti
August 13, 2015 at 6:21 am

(Or is it all "hidden" in the "velocity constrains acceleration" information?)

tbabb Post author
August 13, 2015 at 6:56 am

The Kalman filter is quite good at converging on an accurate state from a poor initial guess. Representing the uncertainty accurately will help attain convergence more quickly– if your initial guess overstates its confidence, the filter may take awhile before it begins to "trust" the sensor readings instead.

In the linked video, the initial orientation is completely random, if I recall correctly. I think it actually converges quite a bit before the first frame even renders. :)

Dayne Batten
August 13, 2015 at 12:32 pm

"The math for implementing the Kalman filter appears pretty scary and opaque in most places you find on Google." Indeed. I've tried to puzzle my way through the Wikipedia explanation of Kalman filters on more than one occasion, and always gave up.

I was able to walk through your explanation with no trouble. Thank you.

Ju
August 24, 2015 at 2:39 pm

I'm a PhD student in economics and decided a while back to never ask Wikipedia for anything related to economics, statistics or mathematics because you will only leave feeling inadequate and confused. Seriously, concepts that I know and understand perfectly well look like egyptian hieroglyphs when I look at the wikipedia representation. I would ONLY look at the verbal description and introduction, the formulas seem to all be written by a wizard savant.

Alex Polotsk

August 13, 2015 at 12:34 pm

The article has a perfect balance between intuition and math! This is the first time I actually understood Kalman filter. =)

I have a couple of questions.

The way we got second equation in (4) wasn't easy for me to see until I manually computed it from the first equation in (4). Is it meant to be so, or did I missed a simple relation? When you say "I'll just give you the identity", what "identity" are you referring to? Are you referring to given equalities in (4)?

So, sensors produce:
– observed noisy mean and covariance (z and R) we want to correct, and
– an additional info 'control vector' (u) with known relation to our prediction.
Correct?

Does H in (8) maps physical measurements (e.g. km/h) into raw data readings from sensors (e.g. uint32, as described in some accelerometer's reference manual)?

tbabb Post author
August 13, 2015 at 7:13 pm

(4) was not meant to be derived by the reader; just given.

Z and R are sensor mean and covariance, yes. The control vector 'u' is generally not treated as related to the sensors (which are a transformation of the system state, not the environment), and are in some sense considered to be "certain". For example, the commands issued to the motors in a robot

are known exactly (though any uncertainty in the execution of that motion could be folded into the process covariance Q).

Yes, H maps the units of the state to any other scale, be they different physical units or sensor data units. I suppose you could transform the sensor measurements to a standard physical unit before it's input to the Kalman filter and let H be the some permutation matrix, but you would have to be careful to transform your sensor covariance into that same space as well, and that's basically what the Kalman filter is already doing for you by including a term for H. (That would also assume that all your sensors make orthogonal measurements, which not necessarily true in practice).

Nico Galoppo
August 13, 2015 at 4:39 pm

So what happens if you don't have measurements for all DOFs in your state vector? I'm assuming that means that H_k isn't square, in which case some of the derivation doesn't hold, right? What do you do in that case?

Robert
August 13, 2015 at 7:44 pm

Excellent explanation. Thanks!

John Dabbar
August 15, 2015 at 12:07 am

Kalman filters are used in dynamic positioning systems for offshore oil drilling.

Istvan Hajnal
August 15, 2015 at 3:13 pm

Great write up. Very helpful. Thanks.

Just one question. Shouldn't it be p_k in stead of x_k (and p_k-1 instead of x_k-1) in the equation right before equation (2)? Also, in (2), that's the transpose of x_k-1, right?
I guess the same thing applies to equation right before (6)?

Regards,
Istvan Hajnal

tbabb Post author
August 17, 2015 at 8:21 pm

Yes, my thinking was to make those kinematic equations look "familiar" by using x (and it would be understood where it came from), but perhaps the inconsistency is worse. :\

Sourjya Sarkar

August 17, 2015 at 2:30 pm

Great ! Really interesting and comprehensive to read.

Li

August 18, 2015 at 5:28 pm

Thanks for the post, I have learnt a lot. My background is signal processing, pattern recognition.

One question:

If we have two probabilities and we want to know the chance that both are true, we just multiply them together.

Why not use sum or become Chi-square distribution?

Because from http://math.stackexchange.com/questions/101062/is-the-product-of-two-gaussian-random-variables-also-a-gaussian

The product of two Gaussian random variables is distributed, in general, as a linear combination of two Chi-square random variables.

Thanks,
Regards,
Li

tbabb Post author
August 20, 2015 at 6:48 pm

Ah, not quite. Let X and Y both be Gaussian distributed. We are not doing $\text{pdf}(X \cdot Y)$, we are doing $\text{pdf}(X) \cdot \text{pdf}(Y)$!

Raphael Michel
August 19, 2015 at 4:33 pm

Really interesting article. Clear and simple. Exactly what I needed.

Stephane
August 20, 2015 at 6:23 am

Thank you very much for this very clear article!

Qu
August 20, 2015 at 8:43 am

Great post ! I have a question about fomula (7), How to get Qk genenrally ?

Prof. Vadlamani Ravi
August 21, 2015 at 11:42 pm

Great post. It demystifies the Kalman filter in simple graphics. A great teaching aid. Thx.

Chintan
August 25, 2015 at 4:32 pm

Hello Tim,

Very nice article. I had read the signal processing article that you cite and had given up half way.

This article clears many things. I will now have to implement it myself.

It would be nice if you could write another article with an example or maybe provide Matlab or Python code.

Keep up the good work.

Best Regards

Chintan

Naseem Makiya
August 26, 2015 at 3:49 am

Awesome post!!! Great visuals and explanations.

Paul
October 7, 2015 at 7:34 pm

Can you realy knock an Hk off the front of every term in (16) and (17) ?
I think this operation is forbidden for this matrix.

Insik
October 13, 2015 at 8:48 am

Wow! This post is amazing. It really helps me to understand true meaning behind equations.

rosie

October 28, 2015 at 5:48 am

This is simplyy awesum!!!! this demonstration has given our team a confidence to cope up with the assigned project

---

Debo
November 5, 2015 at 12:52 pm

Great post! Thanks

hec
November 13, 2015 at 1:54 am

Amazing article, I struggled over the textbook explanations. This article summed up 4 months of graduate lectures, and i finally know whats going on. Thank you.

Nezih
November 13, 2015 at 10:01 am

Greta article, thank you!

Mel
November 18, 2015 at 5:52 pm

Great work. Thanks!

Jose Kurian
November 19, 2015 at 1:26 pm

Hello,

This is indeed a great article. I have been trying to understand this filter for some time now. This article makes most of the steps involved in developing the filter clear.

I how ever did not understand equation 8 where you model the sensor. What does the parameter H do here. How do you obtain the components of H.

Thanks in advance,

Jose

carolinux
November 30, 2015 at 1:13 pm

Very good job explaining and illustrating these! Now I understand how the Kalman gain equation is derived. It was hidden inside the properties of Gaussian probability distributions all along!

Sam
December 5, 2015 at 4:09 pm

This is the greatest explanation ever!

Vidhya Venugopal
December 10, 2015 at 1:37 am

Great explanation! I have a question though just to clarify my understanding of Kalman Filtering. In the above example (position, velocity), we are providing a constant acceleration value ‘a’. Assuming this is a car example, let’s say the driver decides to change the acceleration during the trip. From what I understand of the filter, I would have to provide this value to my Kalman filter for it to calculated the predicted state every time I change the acceleration. Kalman filter would be able to “predict” the state without the information that the acceleration was changed. Is this correct? Also, would this be impractical in a real world situation, where I may not always be aware how much the control (input) changed?
Can anyone help me with this?

javad
January 2, 2016 at 4:56 pm

you are the best Tim!
thank you very much

minh
January 8, 2016 at 12:05 am

hey, my kalman filter output is lagging the original signal. However it does a great job smoothing. How does lagging happen

---

Sujay
January 21, 2016 at 7:23 am

I must say the best link in the first page of google to understand Kalman filters. I guess I read around 5 documents and this is by far the best one. Well done and thanks!! cheers!! :D

Salman
February 4, 2016 at 5:47 am

Excellent explanation.

After reading many times about Kalman filter and giving up on numerous occasions because of the complex probability mathematics, this article certainly keeps you interested till the end when you realize that you just understood the entire concept.

Keep up the good work.

Ben
February 4, 2016 at 11:01 am

Thank you for your excelent work!
There is no doubt, this is the best tutorial about KF !
Many thanks!

William A
February 17, 2016 at 9:23 pm

Great article ! Clear and easy to understand.

Paul
February 24, 2016 at 12:40 pm

This is by far the best explanation of a Kalman filter I have seen yet. Very well done.

Baljit Kaur
March 1, 2016 at 10:19 am

v.nice explanation. Actually I have something different problem if you can provide a solution to me. In my system, I have starting and end position of a robot. I need to find angle if robot needs to rotate and velocity of a robot. Can I get solution that what will be Transition matrix, x(k-1), b(k), u(k).
Thanks Baljit

Maneesha K
March 20, 2016 at 11:40 am

Such an amazing explanation of the much scary kalman filter. Kudos to the author. Thanks very much Sir. Expecting such explanation for EKF, UKF and Particle filter as well.

Ali
March 23, 2016 at 6:11 pm

Hello There!

Great article but I have a question. Why did you consider acceleration as external influance? Could we add the acceleration inside the F matrix directly e.g. x=[position, velocity, acceleration]' ?

Thanks!

Clive Myrie
May 6, 2016 at 2:47 pm

I think that acceleration was considered an external influence because in real life applications acceleration is what the controller has (for lack of a better word) control of. In other words, acceleration and acceleration commands are how a controller influences a dynamic system.

Minh
March 28, 2016 at 1:54 am

Thank you so so much Tim. The math in most articles on Kalman Filtering looks pretty scary and obscure, but you make it so intuitive and accessible (and fun also, in my opinion). Again excellent job! Would you mind if I share part of the particles to my peers in the lab and maybe my students in problem sessions? I'll certainly mention the source

---

Jeroen
April 1, 2016 at 12:01 pm

Best explanation I've read so far on the Kalman filter. Far better than many textbooks. Thank you.

Graeme
April 3, 2016 at 8:43 pm

Without doubt the best explanation of the Kalman filter I have come across! Often in DSP, learning materials begin with the mathematics and don't give you the intuitive understanding of the problem you need to fully grasp the problem. This is a great resource. Thanks.

vishwanath
April 9, 2016 at 11:30 am

amazing...simply simplified.you saved me a lot of time...thanks for the post.please update with nonlinear filters if possible that would be a great help.

Laurent Vosgien
April 23, 2016 at 4:31 pm

Your original approach (is it ?) of combining Gaussian distributions to derive the Kalman filter gain is elegant and intuitive. All presentations of the Kalman filter that I have read use matrix algebra to derive the gain that minimizes the updated covariance matrix to come to the same result. That was satisfying enough to me up to a point but I felt i had to transform X and P to the measurement domain (using H) to be able to convince myself that the gain was just the barycenter between the a priori prediction distribution and the measurement distributions weighted by their covariances. This is

where other articles confuse the reader by introducing Y and S which are the difference z-H*x called innovation and its covariance matrix. Then they have to call S a “residual” of covariance which blurs understanding of what the gain actually represents when expressed from P and S. Good job on that part !

I will be less pleasant for the rest of my comment, your article is misleading in the benefit versus effort required in developing an augmented model to implement the Kalman filter. By the time you invested the research and developing integrated models equations for errors of your sensors which is what the KF filter is about, not the the recursive algorithm principle presented here which is trivial by comparison.

There is nothing magic about the Kalman filter, if you expect it to give you miraculous results out of the box you are in for a big disappointment. By the time you have developed the level of understanding of your system errors propagation the Kalman filter is only 1% of the real work associated to get those models into motion. There is a continuous supply of serious failed Kalman Filters papers where greedy people expect to get something from nothing implement a EKF or UKF and the result are junk or poor. All because of article like yours give the false impression that understanding a couple of stochastic process principles and matrix algebra will give miraculous results. The work in not where you insinuate it is. Understanding the Kalman filter predict and update matrix equation is only opening a door but most people reading your article will think it’s the main part when it is only a small chapter out of 16 chapters that you need to master and 2 to 5% of the work required.

Byambajav
April 27, 2016 at 11:27 am

Great article I’ve ever been reading on subject of Kalman filtering. Thanks !!!

Edu
April 28, 2016 at 10:58 pm

fantastic | thanks for the outstanding post !

Wanjohi
May 5, 2016 at 9:54 am

First time am getting this stuff…..it doesn’t sound Greek and Chinese…..greekochinese…..
on point….and very good work…..

Mohamad
May 8, 2016 at 2:32 am

thank you Tim for your informative post, I did enjoy when I was reading it, very easy and logic… good job

---

Kurt Ramsdale
May 11, 2016 at 5:08 am

Equation 18 (measurement variable) is wrong.
Equation 16 is right. Divide all by H.

tbabb Post author
May 11, 2016 at 5:43 am

What's the issue? Note that K has a leading H_k inside of it, which is knocked off to make K'.

Kurt Ramsdale
May 11, 2016 at 8:23 am

x has the units of the state variables.
z has the units of the measurement variables.
K is unitless 0-1.
The units don't work unless the right term is K(z/H-x).

Mehdi
May 11, 2016 at 12:06 pm

Excellent Post! Kalman Filter has found applications in so diverse fields. A great one to mention is as a online learning algorithm for Artificial Neural Networks.

vishnu
May 12, 2016 at 10:21 am

Great Article. Nicely articulated. Do you recommened any C++ or python implementation of kalman filter? I know there are many in google but your recommendation is not the same which i choose.

Assume that every car is connected to internet. I am trying to predict the movement of bunch of cars, where they probably going in next ,say 15 min. you can assume like 4 regions A,B,C,D (5-10km of radius) which are close to each other. How can I make use of kalman filter to predict and say, so many number cars have moved from A to B.

I am actullay having trouble with making the Covariance Matrix and Prediction Matrix. In my case I know only position. Velocity of the car is not reported to the cloud. So First step could be guessing the velocity from 2 consecutive position points, then forming velocity vector and position vector.Then applying your equations. Is my assumption is right? Thanks

P.S: sorry for the long comment.Need Help. Thanks

Ebrahim Mirzaei
May 17, 2016 at 3:51 pm

Excellent Post! thanks alot.
I want to use kalman Filter to auto correct 2m temperature NWP forecasts.
Could you please help me to get a solution or code in R, FORTRAN or Linux Shell Scripts(bash,perl,csh,...) to do this.

omid
May 20, 2016 at 8:24 pm

Hi
I'm kinda new to this field and this document helped me a lot
I just have one question and that is what is the value of the covariance matrix at the start of the process?

Will
May 20, 2016 at 9:29 pm

This is the best article I've read on Kalman filter so far by a long mile!

Will
May 20, 2016 at 9:32 pm

Btw, will there be an article on Extend Kalman Filter sometime in the future, soon hopefully?

Thanks! Great work

---

urwa

May 22, 2016 at 1:52 pm

Thanks a bunch. :) Very helpful indeed