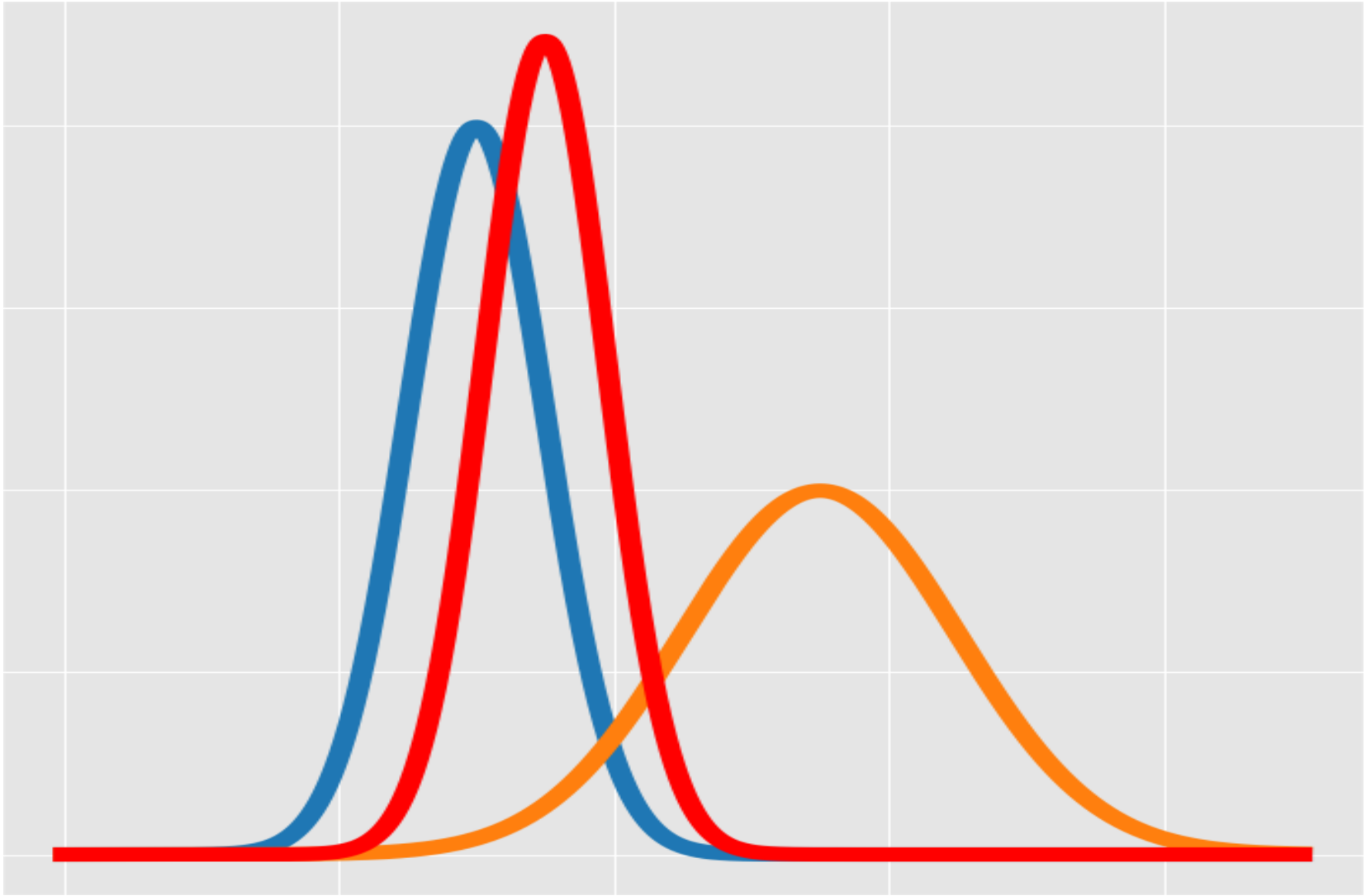
Kalman Filters

Noor Alhasani



Introduction

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. It can also 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.

Example

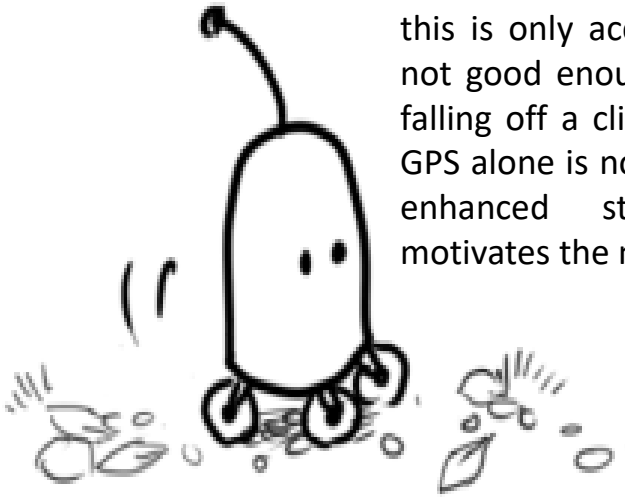
Gaussian distributions are exactly the same as normal distributions, just a different name.

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 $\overrightarrow{x_k}$ which is just a position and a velocity:

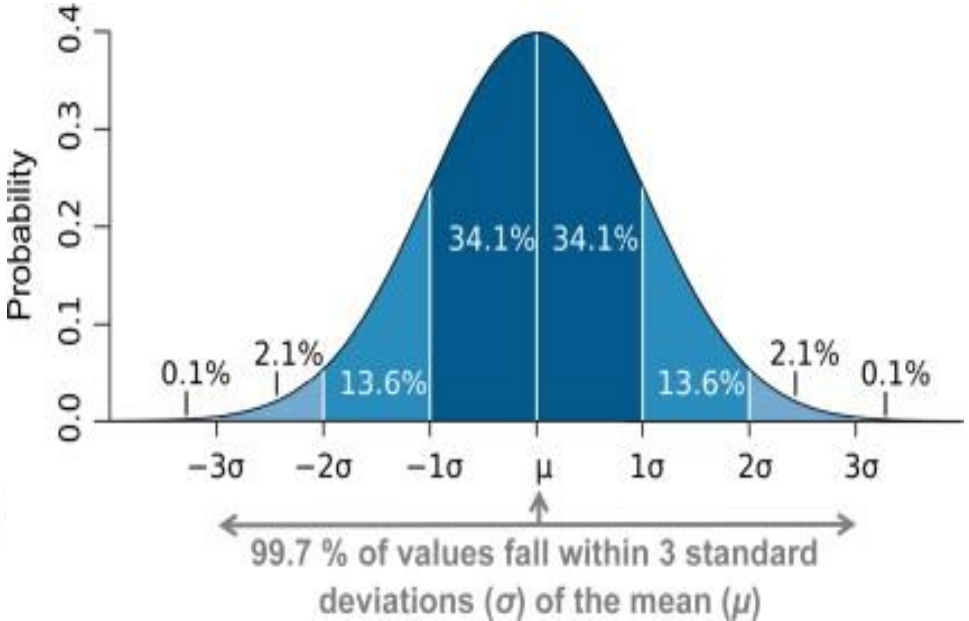
$$\overrightarrow{x_k} = (\vec{p}, \vec{v})$$

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.

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 (position and velocity, in our case) are random and *Gaussian distributed.* Each variable has a **mean** value μ, which is the center of the distribution (most likely state) & **variance** σ^2 which is the uncertainty.

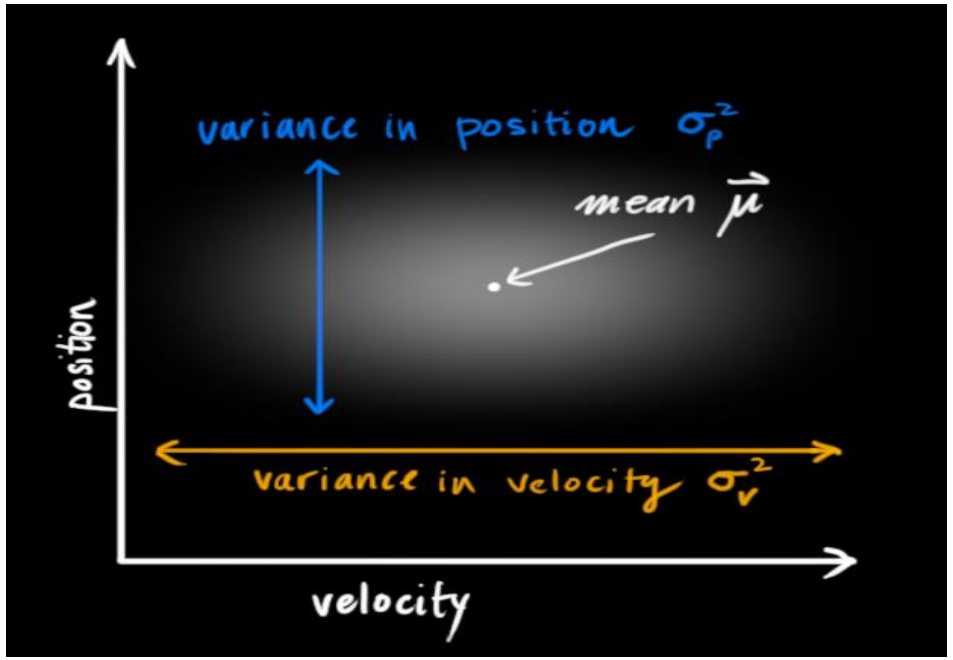


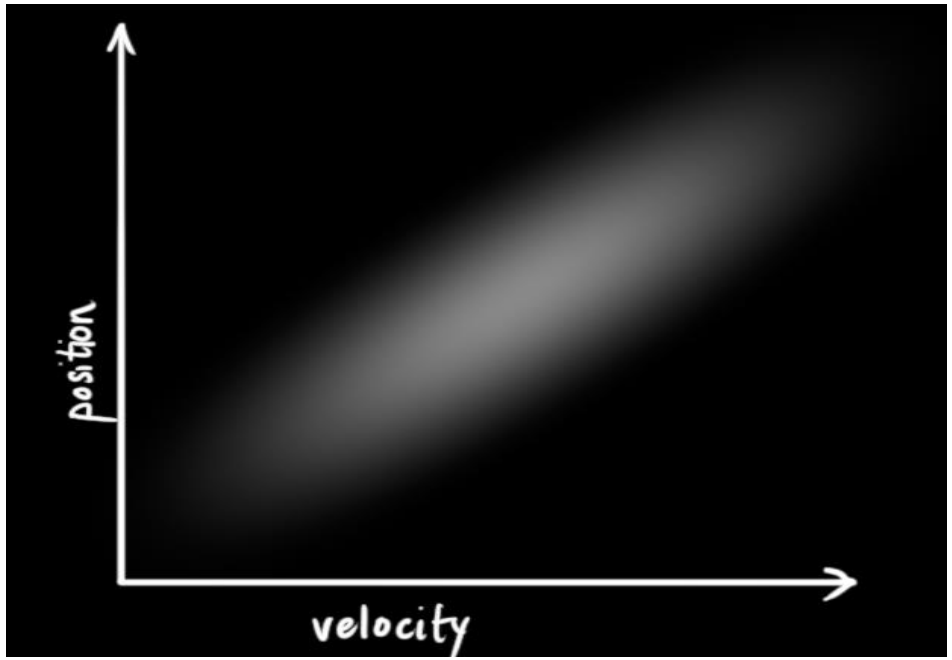
Whilst the robot may have a GPS sensor, this is only accurate to 10m, which is not good enough if we want to stop it falling off a cliff or into a ditch. Hence, GPS alone is not good enough, we need enhanced state estimation. This motivates the need for a Kalman filter.



State Correlation

In general, the states could be uncorrelated (left image), or correlated (right image). In this case, they are correlated as velocity is a time derivative of position.





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 Σ_{ij} is the degree of correlation between the i^{th} state variable and the j^{th} 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 "Σ_{ij}".

Mathematical Problem Description

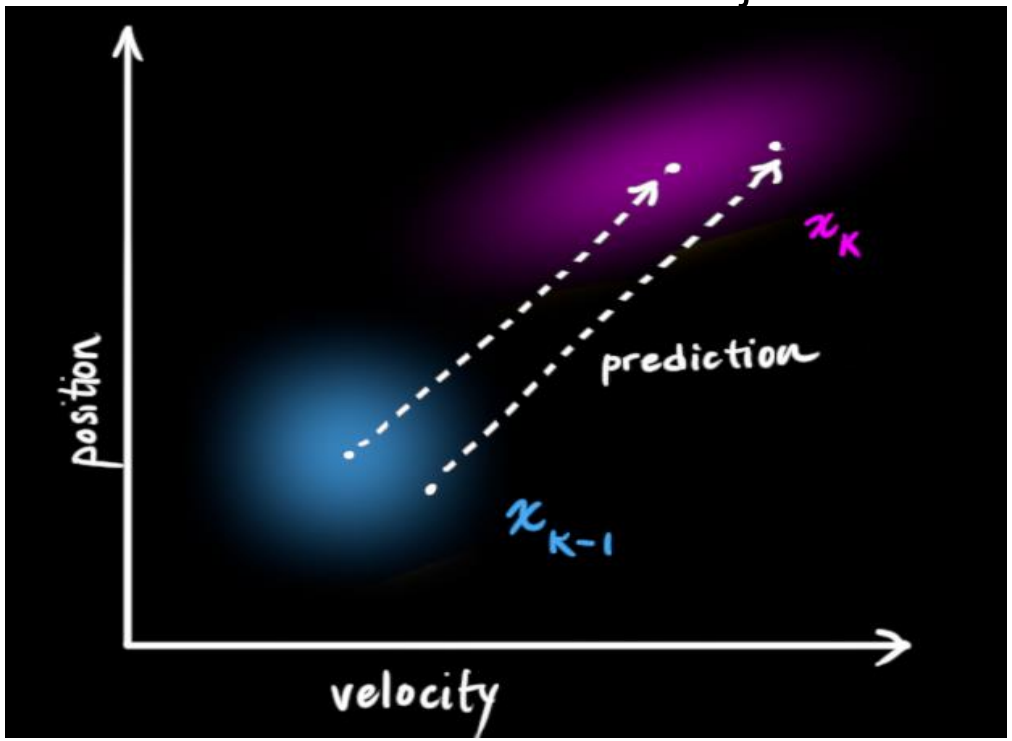
Current State
Future State

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 $\hat{x}_k$ (the mean, elsewhere named μ), and its covariance matrix P_k.

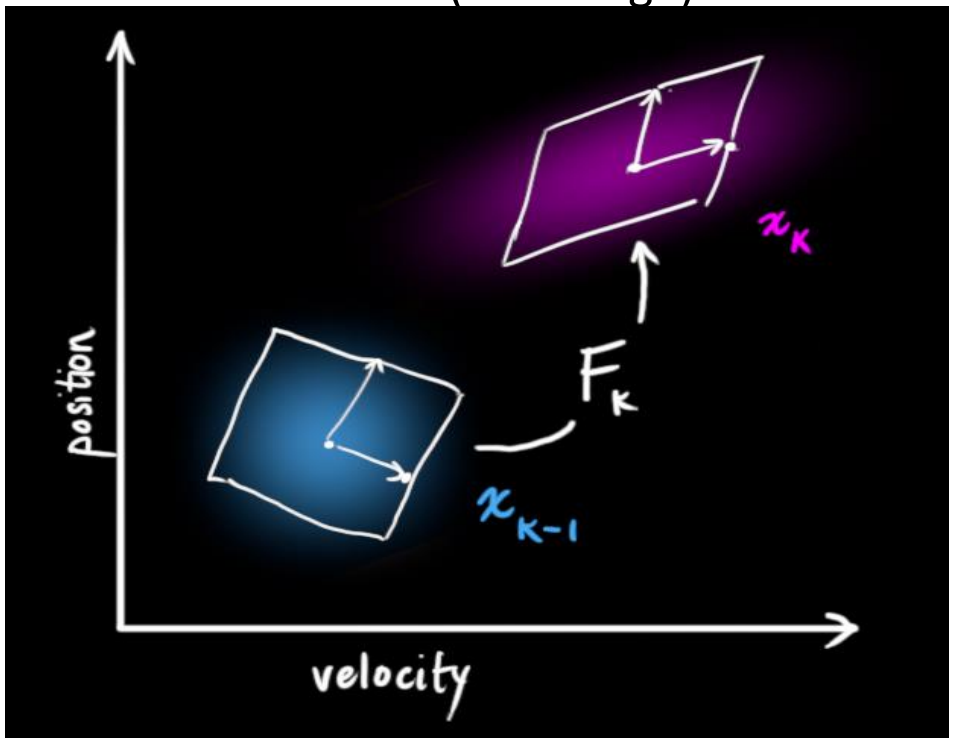
$$\overrightarrow{x_k} = \begin{bmatrix} position \\ velocity \end{bmatrix} \qquad P_k = \begin{bmatrix} \Sigma_{pp} & \Sigma_{pv} \\ \Sigma_{vp} & \Sigma_{vv} \end{bmatrix}$$

The covariance matrix is symmetric, hence $\Sigma_{pv} = \Sigma_{vp}$

(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 (left image).



We can represent the prediction step using a transformation matrix, F_k (right image), It takes every point in our original estimate and moves it to a new predicted location, where is where the system would move if the original estimate was correct.



Covariance Propagation

Current State

Future State

How would we use a matrix to predict the position and velocity at the next moment in the future?

We'll use a basic kinematic formula:

Note: $\mathbf{P}_k$ is the covariance matrix, p_k is the position (which can be a vector but is a scalar in this example)

$$p_k = p_{k-1} + v_{k-1}\,\Delta t$$
$$v_k = v_{k-1}$$

Where Δt is the time between state updates

Written in a more compact matrix form:

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

Where $F_k = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix}$

We now need to know how to update the covariance matrix $\mathbf{P}_k$. If we denote the covariance of $\vec{x}$ (where $\vec{x}$ is the state vector) as $\mathrm{Cov}(\vec{x}) = \Sigma$, we can write the covariance of $\mathbf{A}\vec{x}$ (where $\mathbf{A}$ is a matrix) as $\mathrm{Cov}(\mathbf{A}\vec{x}) = \mathbf{A}\vec{x}\mathbf{A}^T$. Combining this with $\hat{\boldsymbol{x}}_k = \boldsymbol{F}_k \hat{\boldsymbol{x}}_{k-1}$, we can write the covariance of $\hat{\boldsymbol{x}}_k$ as:

$$\mathbf{P}_k = \mathrm{Cov}(\hat{\boldsymbol{x}}_k) = \mathrm{Cov}(\boldsymbol{F}_k \hat{\boldsymbol{x}}_{k-1}) = \boldsymbol{F}_k \mathbf{P}_{k-1} \boldsymbol{F}_k{}^{\mathrm{T}}$$

To conclude this slide, the updated state prediction $\hat{\boldsymbol{x}}_k$ and the updated covariance matrix $\mathbf{P}_k$ can be expressed as:

$$\hat{\boldsymbol{x}}_k = \boldsymbol{F}_k \hat{\boldsymbol{x}}_{k-1}$$
$$\mathbf{P}_k = \boldsymbol{F}_k \mathbf{P}_{k-1} \boldsymbol{F}_k{}^{\mathrm{T}}$$

What about a dynamic system with external influence from a control system, say a motor or brakes ?

Control Inputs

Current State | Control Input | Future State

In this robot example, the navigation software may issue a command to turn the wheels or stop. If we know this additional information about external influences from the world, we can stuff it into a vector called $\vec{u}_k$, 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:

$$p_k = p_{k-1} + v_{k-1}\,\Delta t + \frac{1}{2}a(\Delta t)^2$$

$$v_k = v_{k-1} + a(\Delta t)$$

This is useful as typically, control systems do not directly control the position or velocity, but apply a force F, from which we can obtain the acceleration a.

Written in a more compact matrix form:

$$\hat{\boldsymbol{x}}_k = \boldsymbol{F}_k\hat{\boldsymbol{x}}_{k-1} + \begin{bmatrix}\frac{1}{2}(\Delta t)^2 \\ \Delta t\end{bmatrix} a$$

$$\hat{\boldsymbol{x}}_k = \boldsymbol{F}_k\hat{\boldsymbol{x}}_{k-1} + \boldsymbol{B}_k\,\vec{u}_k$$

Where $\boldsymbol{B}_K = \begin{bmatrix}(\Delta t)^2/2m \\ \Delta t/m\end{bmatrix}$ assuming $\vec{u}_k$ is a force input vector and f = ma applies here (which is only true in an inertial reference frame).

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

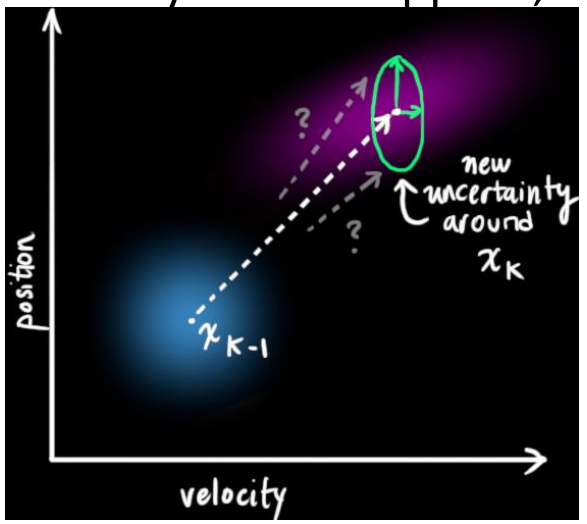
Let's add one more detail. What happens if our prediction is not a 100% accurate model of what's actually going on?

Model Uncertainty

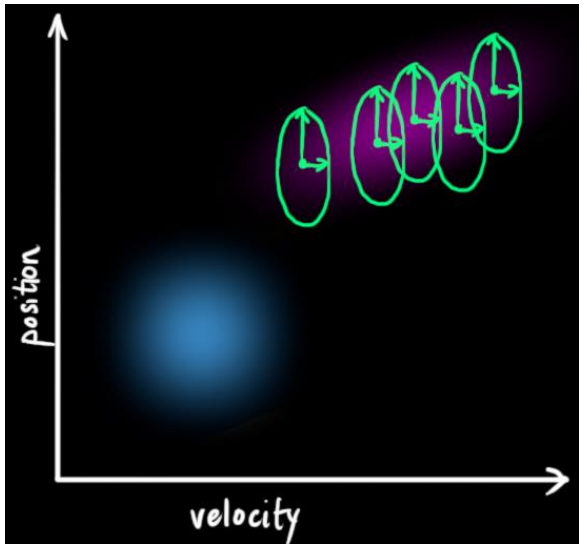
Current State | Control Input | Future State

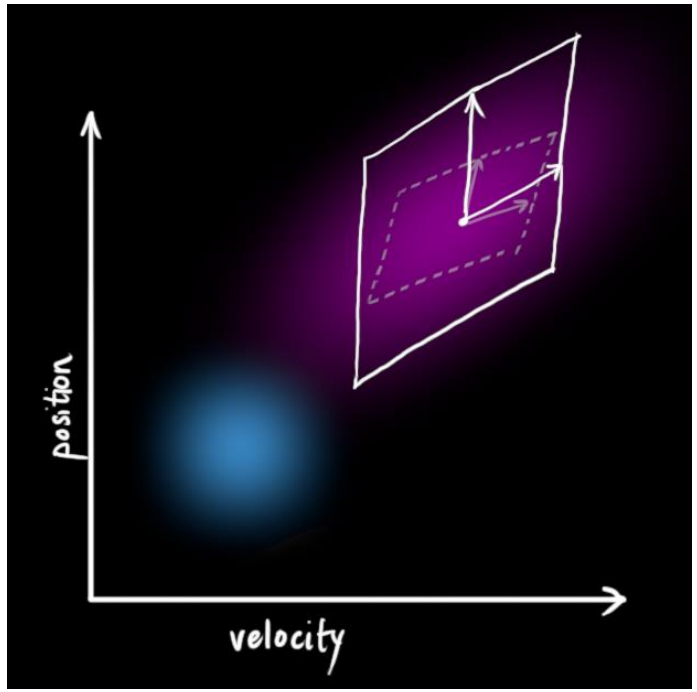
What about forces that we *don't* know about? If we're tracking a quadcopter, for example, it could be buffeted around by wind. If we're tracking a wheeled robot, the wheels could slip, or bumps on the ground could slow it down. We can't keep track of these things, and if any of this happens, our prediction 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 $\hat{x}_{k-1}$ is moved to somewhere inside a Gaussian blob with covariance Q_k. Another way to say this is that we are treating the untracked influences as **noise** with covariance Q_k.





This produces a new Gaussian blob, with a different covariance, but the same mean.

Model Uncertainty

Current State | Control Input | Future State | Uncertainty

We can model the expanded covariance by adding Q_k to our previous expression (end of slide 7) for the covariance matrix P_k.

This means our expressions for $\hat{x}_k$ and P_k become:

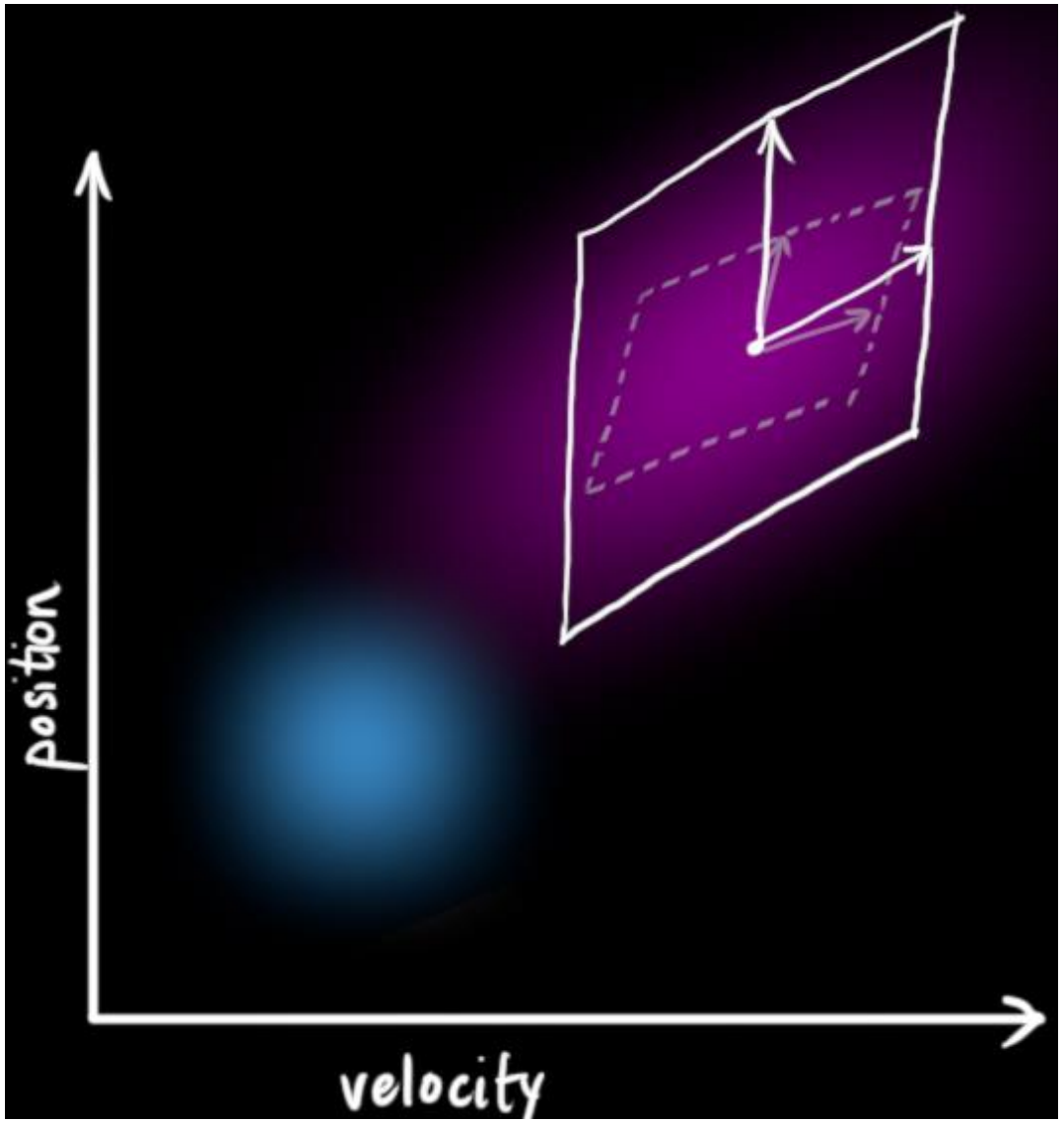
$$\hat{x}_k = F_K \hat{x}_{k-1} + B_k \vec{u}_k$$

$$P_k = F_k P_{k-1} F_k^{\mathrm{T}} + Q_k$$

In other words, the **new best estimate** $(\hat{x}_k)$ is a **prediction** made from **previous best estimate** $(\hat{x}_{k-1})$, plus a **correction** for **known external influences** $(\vec{u}_k)$.

The **new uncertainty** (P_k) **is predicted from the old uncertainty** (P_{k-1}), 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 $\hat{x}_k$ and P_k. What happens when we get some data from our sensors?



Measurements

Current State | Control Input | Future State | Uncertainty

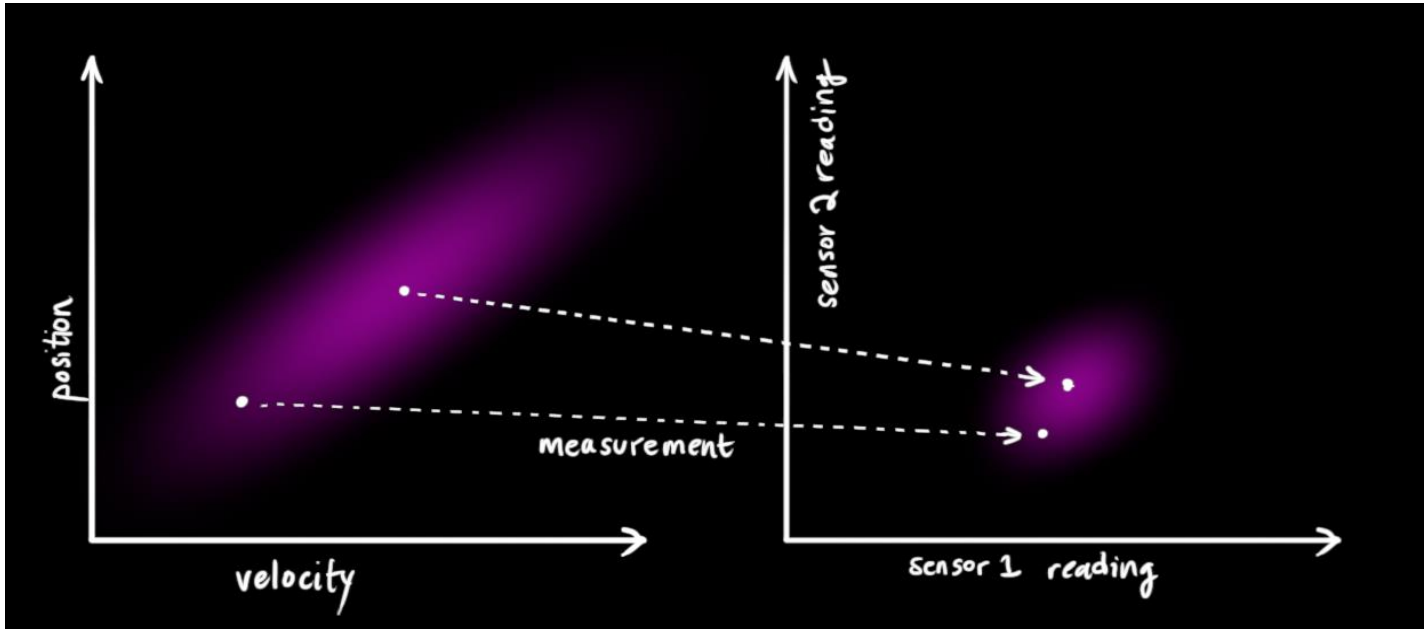
Now that we've made our 'prediction' step, now it's time to 'correct' that (or atleast improve it) using sensor data. We might have several sensors which give us information about the state of our system. For the time being it doesn't matter what they measure; perhaps one reads position and the other 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**.

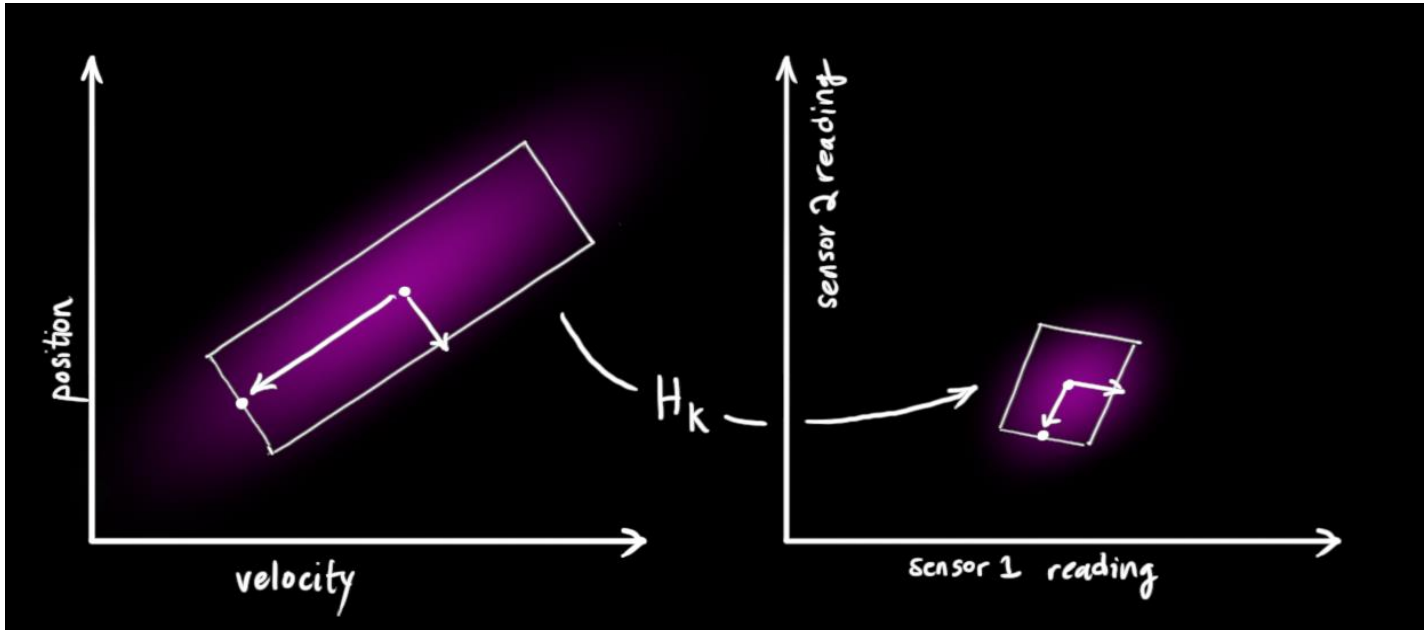
Notice that the units and scale of the reading might not be the same as the units and scale of the state we're keeping track of. You might be able to guess where this is going: We'll model the sensors with a matrix, H_k.

We can figure out the (Gaussian) distribution of the sensor readings as such:

$$\vec{\mu}_{expected} = H_k \hat{\boldsymbol{x}}_k \text{ (see bottom figure)}$$

$$\vec{\Sigma}_{expected} = H_k \mathbf{P}_k H_k^{\mathsf{T}} \text{ (Cov. equation on slide 7)}$$

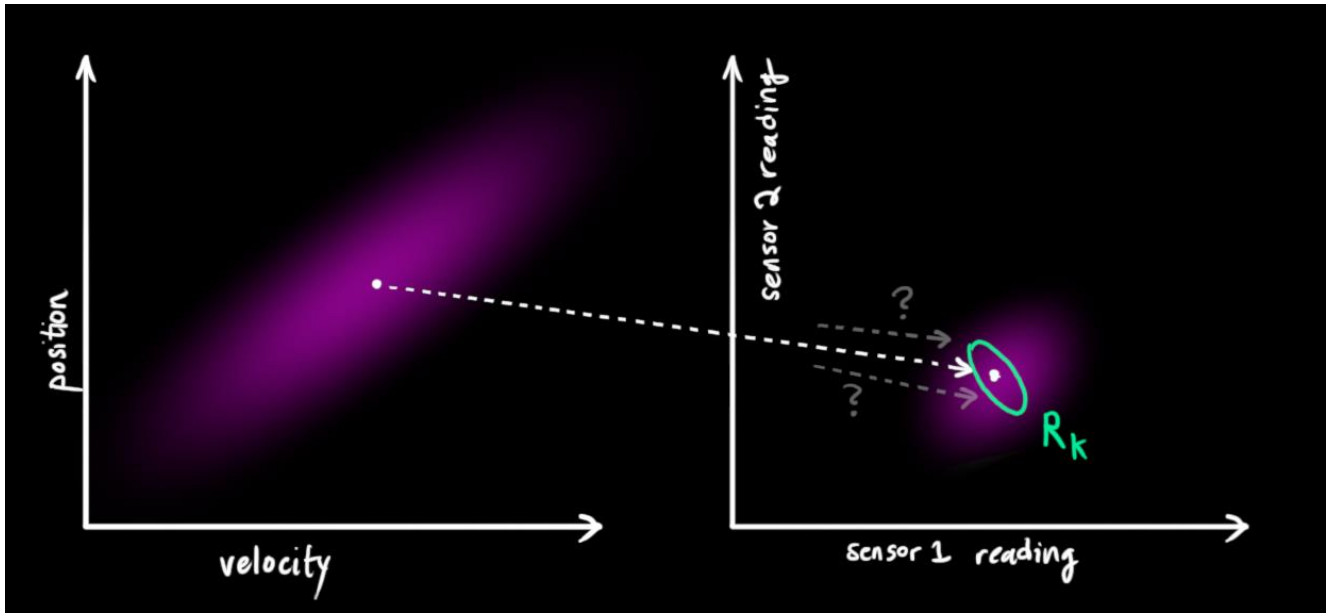




Sensor Noise

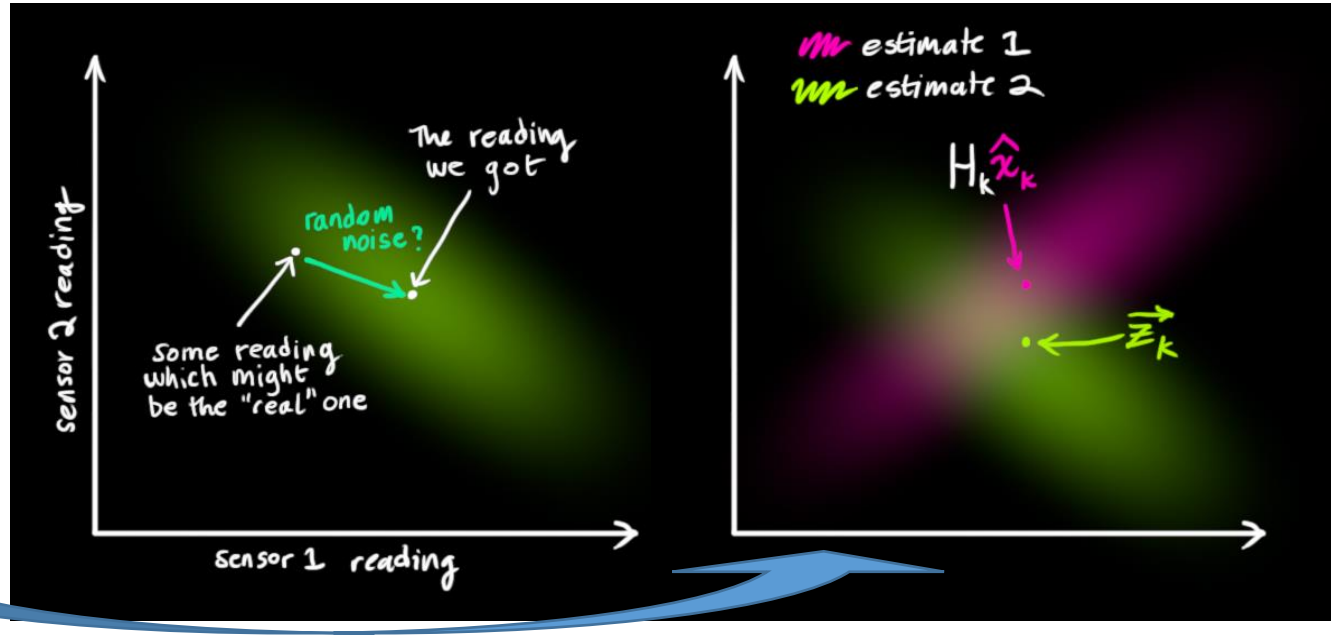
Future Measurement | Current State | Control Input
Future State | Uncertainty

Kalman filters are great for dealing with sensor noise. All sensors (even the best ones) have noise (whether from the environment, i.e. wind, or from within the sensor itself, i.e. thermal noise in an IR sensor). Thus, each individual state in our original estimate ($\hat{x}_{k-1}$) will result in a range of sensor readings. From each reading we observe, we might guess that our system was in a particular state.



However, 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 $\vec{z}_k$.

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



Sensor Noise

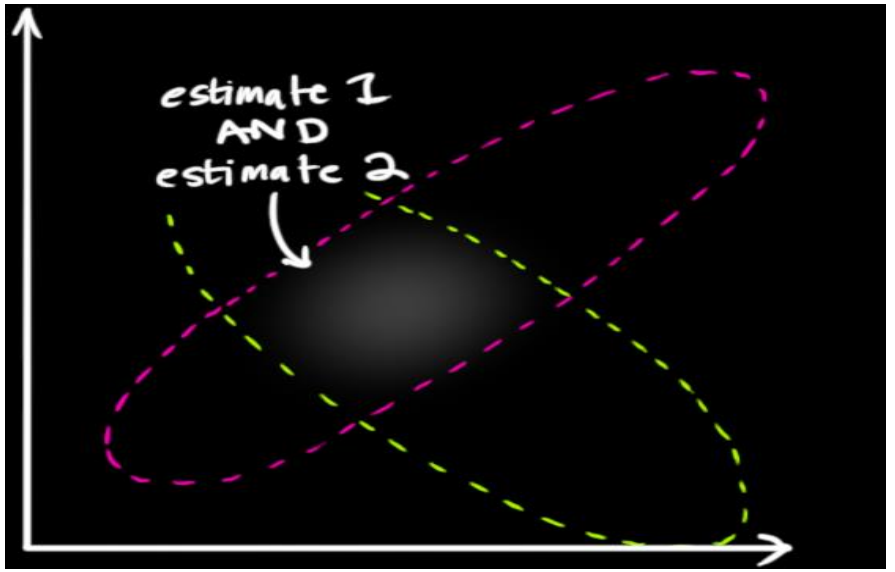
Future Measurement | Current State | Control Input
Future State | Uncertainty

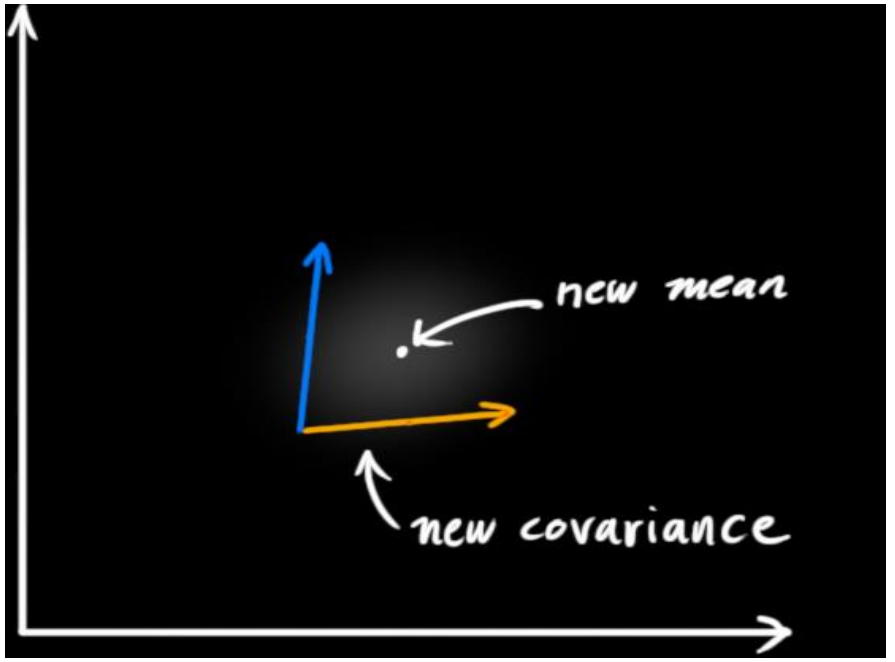
We must try to reconcile our guess about the readings we'd see based on the **predicted state ($\hat{x}_k$)** with a *different* guess based on our **sensor readings** that we actually observed.

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 $\overrightarrow{z_k}$ 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. Hmmm… this looks like another Gaussian blob.





Statistics (eww, I know right)

Future Measurement | Current State | Future State | Control Input | Uncertainty

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. In one dimension, the probability density function (pdf) is given by:

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

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

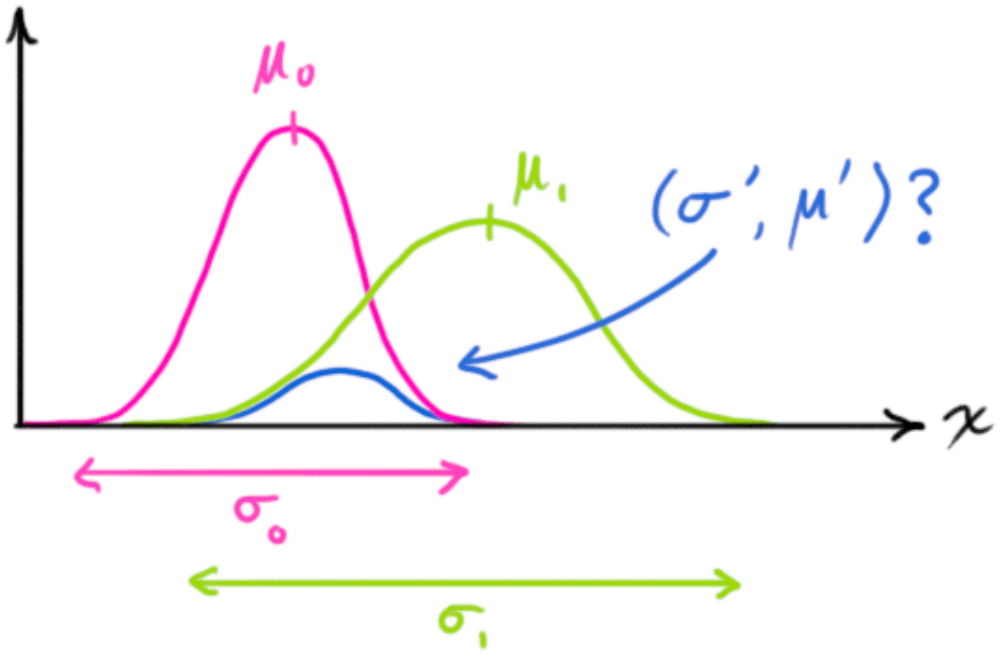
$$\boldsymbol{N(x, \mu_0, \sigma_0) \bullet N(x, \mu_1, \sigma_1)} \stackrel{?}{=} N(x, \mu', \sigma')$$

By substituting the Gaussian pdf (top of slide) into the equation just above, doing some algebra and being careful to normalise (so as to keep the total probability equal to 1), we obtain:

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

If we let $\mathbf{k} = \frac{\sigma_0^2}{\sigma_0^2 + \sigma_1^2}$:

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



More Statistics (sorry I promise it's almost over)

On the last slide, we saw that for scalar values of mean and standard deviation, we obtained:

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

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

What about multi-variate Gaussian distributions? The means each become a vector and the standard deviations each become a matrix (called the Covariance matrix), denoted by $\vec{\mu}_0, \vec{\mu}_1, \Sigma_0 \& \Sigma_1$ respectively. In this case we obtain the following:

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

Where $\mathbf{K} = \Sigma_0\,(\Sigma_0 + \Sigma_1)^{-1}$, this matrix is called the **Kalman gain matrix**, we will use it in a moment.

Putting it all together

Future Measurement | Current State | Control Input | Future State | Uncertainty

We have 2 distributions. First, the predicted measurement with $\boldsymbol{\mu_0} = H_k\hat{\boldsymbol{x}}_k$ and $\Sigma_0 = H_k P_k H_k^T$. Secondly, we have the observed measurement with $\mu_1 = \overrightarrow{z_k}$ and $\boldsymbol{\Sigma_1} = \mathbf{R_k}$ (in bold purely for visibility reasons). Substituting these equations into the equations for $\vec{\mu}' = \overrightarrow{\mu_0} + \mathbf{K}(\overrightarrow{\mu_1} - \overrightarrow{\mu_0})$ and $\Sigma' = \Sigma_0 - \mathbf{K}\Sigma_0$, where $\mathbf{K} = \Sigma_0 (\Sigma_0 + \Sigma_1)^{-1}$, we obtain:

$$H_k\hat{x}'_k = H_k\hat{\boldsymbol{x}}_k + \mathbf{K}(\overrightarrow{z_k} - H_k\hat{\boldsymbol{x}}_k) \quad \text{and} \quad H_k P'_k H_k^T = H_k P_k H_k^T - \mathbf{K}(H_k P_k H_k^T)$$

Substituting $\Sigma_0 = H_k P_k H_k^T$ and $\boldsymbol{\Sigma_1} = \mathbf{R_k}$ into $\mathbf{K} = \Sigma_0 (\Sigma_0 + \Sigma_1)^{-1}$, we can express the Kalman gain matrix as follows:

$$\mathbf{K} = H_k P_k H_k^T (H_k P_k H_k^T + \mathbf{R_k})^{-1}$$

Removing H_k from $H_k\hat{x}'_k = H_k\hat{\boldsymbol{x}}_k + \mathbf{K}(\overrightarrow{z_k} - H_k\hat{\boldsymbol{x}}_k)$, we obtain $\hat{x}'_k = \hat{\boldsymbol{x}}_k + \mathbf{K'}(\overrightarrow{z_k} - H_k\hat{\boldsymbol{x}}_k)$, where $\mathbf{K'} = P_k H_k^T (H_k P_k H_k^T + \mathbf{R_k})^{-1}$

Removing H_k from $H_k P'_k H_k^T = H_k P_k H_k^T - \mathbf{K}(H_k P_k H_k^T)$, we obtain $P'_k H_k^T = P_k H_k^T - \mathbf{K'}(H_k P_k H_k^T)$. Removing H_k^T from the end of each term of this, we obtain $P'_k = P_k - \mathbf{K'}(H_k P_k)$.

The 3 highlighted equations give us the complete equations for the update step!

$$\hat{x}'_k = \hat{\boldsymbol{x}}_k + \mathbf{K'}(\overrightarrow{z_k} - H_k\hat{x}_k)$$
$$P'_k = P_k - \mathbf{K'}(H_k P_k)$$
$$\mathbf{K'} = P_k H_k^T (H_k P_k H_k^T + \mathbf{R_k})^{-1}$$

$\hat{x}'_k$ is our new best estimate, we can then move on in time (k becomes k+1, k-1 becomes k) and feed $\hat{x}'_k$ & P'_k back into another round of predict and update. That's it!

Final Equations

Future Measurement | Current State | Future State | Control Input | Uncertainty

Of all the equations in this presentation, the only ones you actually need to implement are:

$$\hat{x}_k = F_K \hat{x}_{k-1} + B_k \vec{u}_k$$

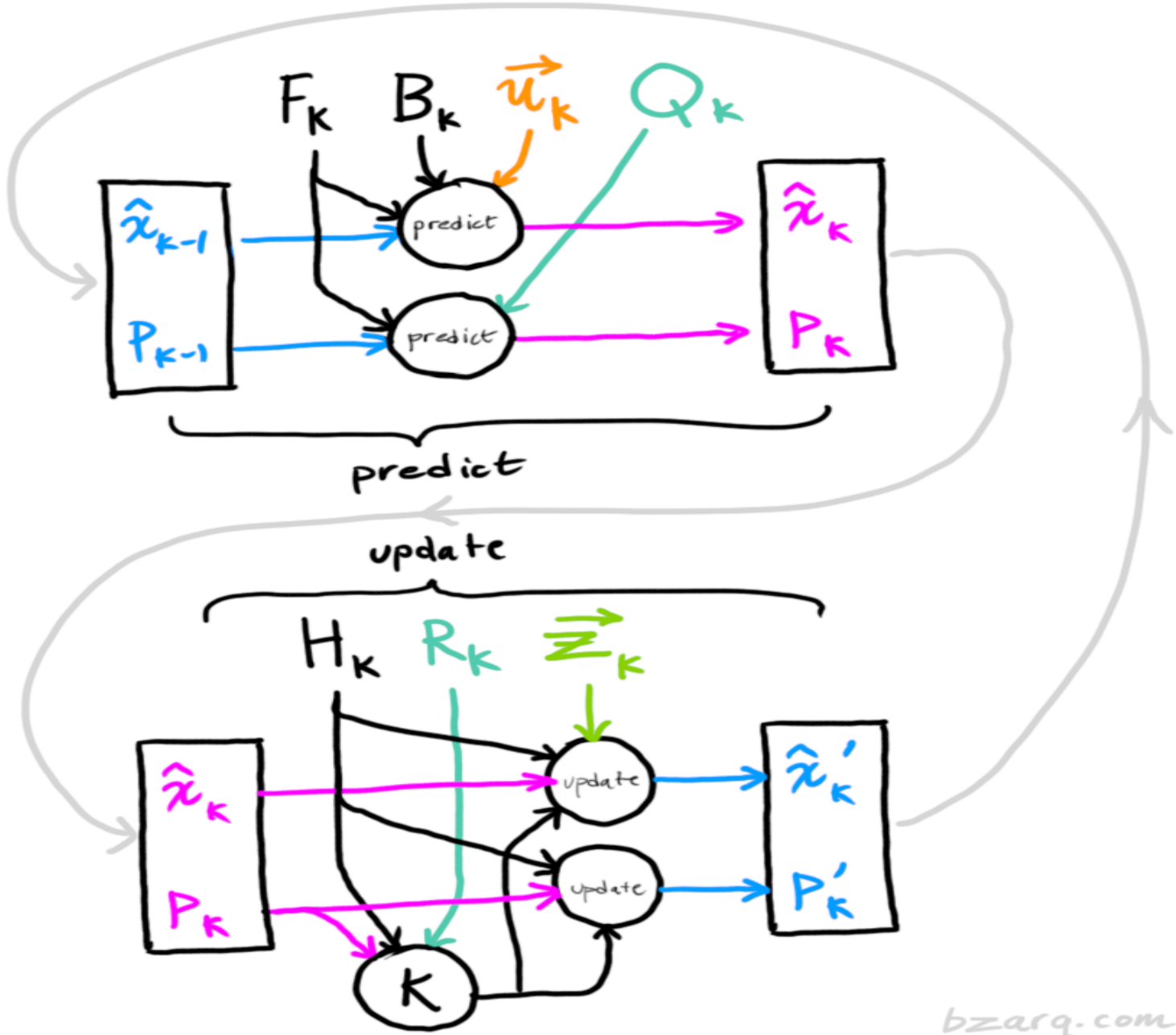
$$P_k = F_k P_{k-1} F_k^T + Q_k$$

$$\hat{x}'_k = \hat{x}_k + K'(\vec{z}_k - H_k \hat{x}_k)$$

$$P'_k = P_k - K'(H_k P_k)$$

$$K' = P_k H_k^T (H_k P_k H_k^T + R_k)^{-1}$$

At the end of the loop, $\hat{x}'_k$ & P'_k become the new values for $\hat{x}_{k-1}$ & P_{k-1} .



Kalman Filter Overview

Advantages	Disadvantages
Enables Accurate State Modelling.	Only works for linear systems, linearization can mitigate this (which is what the extended Kalman filter does), however this won't work for highly non-linear systems. For this, an unscented Kalman filter is suitable.
Low computational overhead, can be implemented on slow hardware as you only need to store the current variables, previous variables can be wiped on the fly.	Complex, this makes learning it difficult and hence good implementation can be difficult without a solid theoretical understanding.
Removes the requirement for high end hardware (i.e. sensors and electronics), thus reduces cost	May be difficult to accurately obtain values for modelling uncertainties (Q_k) and sensor uncertainties (R_k). This will require hours of laboratory testing if existing data is not available. Educated guesses close enough to the true distribution should work in most cases.
Very robust, can deal with a wide variety of uncertainties, relies on relatively few modelling assumptions.	Assumptions must be validated, if noise or covariance are not Gaussian processes, the true statistical distribution must be obtain for the filter to work optimally. If it is very different in reality, the Kalman filter will not converge.
A lot of flight heritage, first major application in the Apollo guidance computer (and the applications in the modern day cannot be understated, it is used everywhere).	There is no perfect formula to get it working immediately, often, in practise, the parameters will require constant tweaking and it will require many hours of testing and analysis to determine what the optimum trade-offs between running speed, convergence time, stability and computational overhead are.

Sources

- This presentation is adapted from the excellent content from https://www.bzarg.com/p/how-a-kalman-filter-works-in-pictures/ , which provides a useful and intuitive explanation.

- Any further sources used are indicated within the slides.

Thank you for reading!

For any queries, please contact me at:

- Email: alnoor587@gmail.com
- LinkedIN: https://www.linkedin.com/in/noor-alhasani-a2443a195/