Kalman Filter Part I: Kalman Filter

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.

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.

Example

Let's assume we've built a robot that can wander around in the woods, and the robot needs to know exactly where it is so it can navigate. We will consider that the robot has a state $\boldsymbol{x}_k$ = (**p, v**), which is just a position and velocity

Our robot also has a GPS sensor, which is accurate to about 10 meters. It's not precise 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.

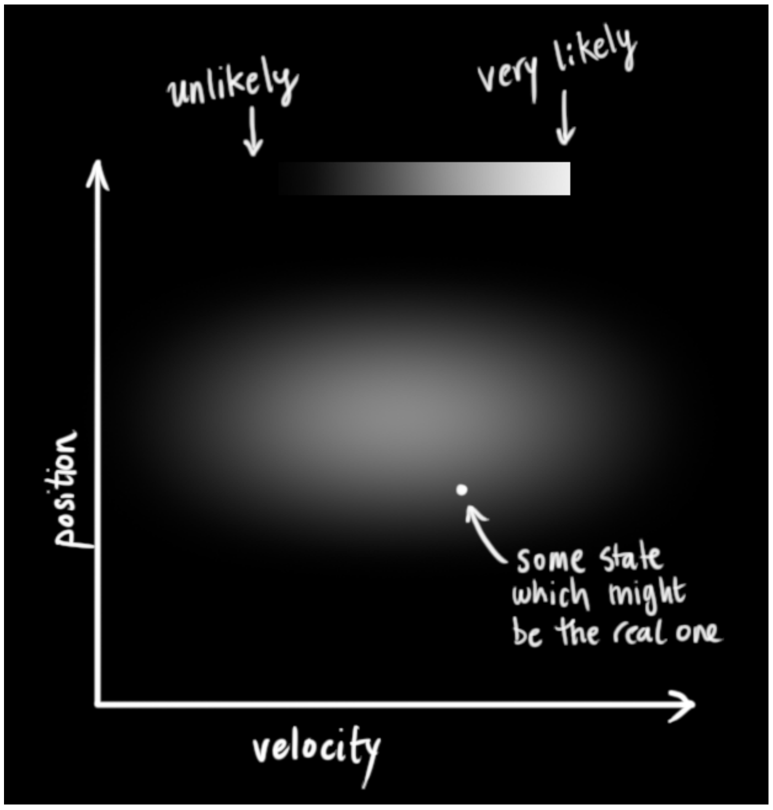
The GPS **sensor** tells us something about the state, but only indirectly, and with some uncertainty or inaccuracy. Our **prediction** tells us something about how the robot is moving, but only indirectly, and with some uncertainty or inaccuracy. But if we use all the information available to us, can 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 Kalman Filter works?

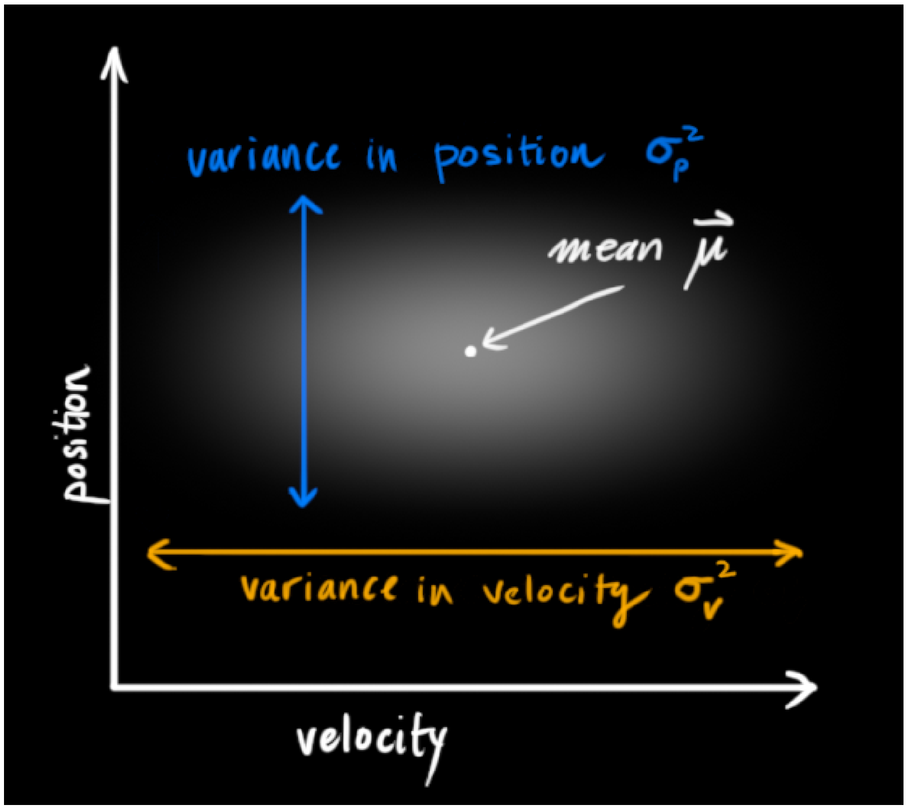
Let's look at the landscape we're trying to interpret. We'll continue with a simple state having only position and velocity.

x =[p,v]

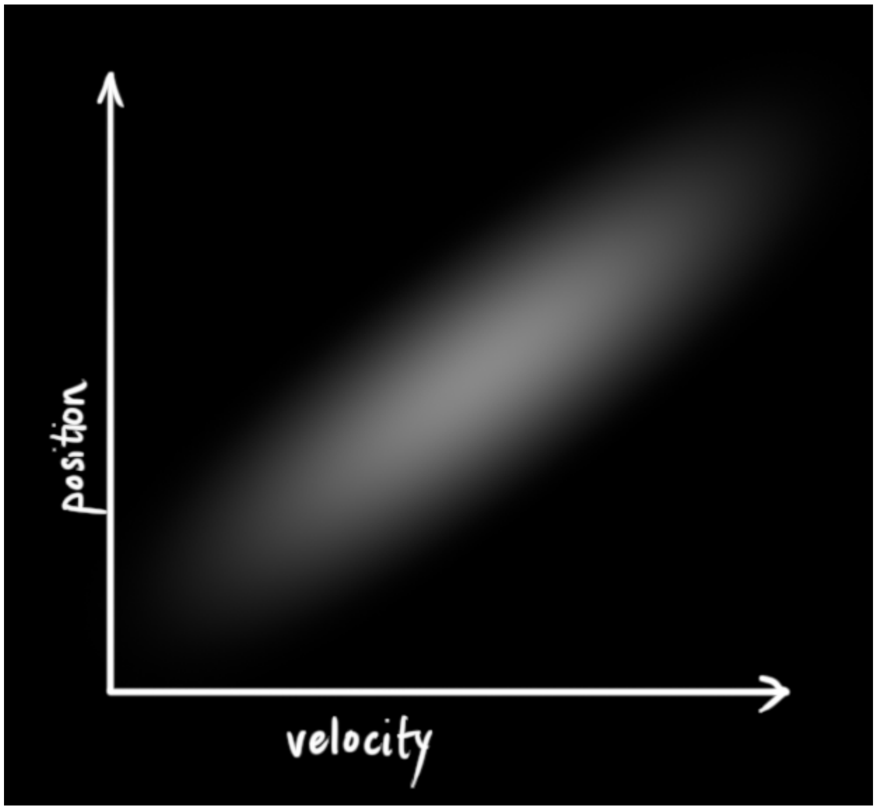
We don't know what the *actual* position and velocity are; there are a whole range of possible combinations of position and velocity that might be true, but some of them are more likely than others:



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


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

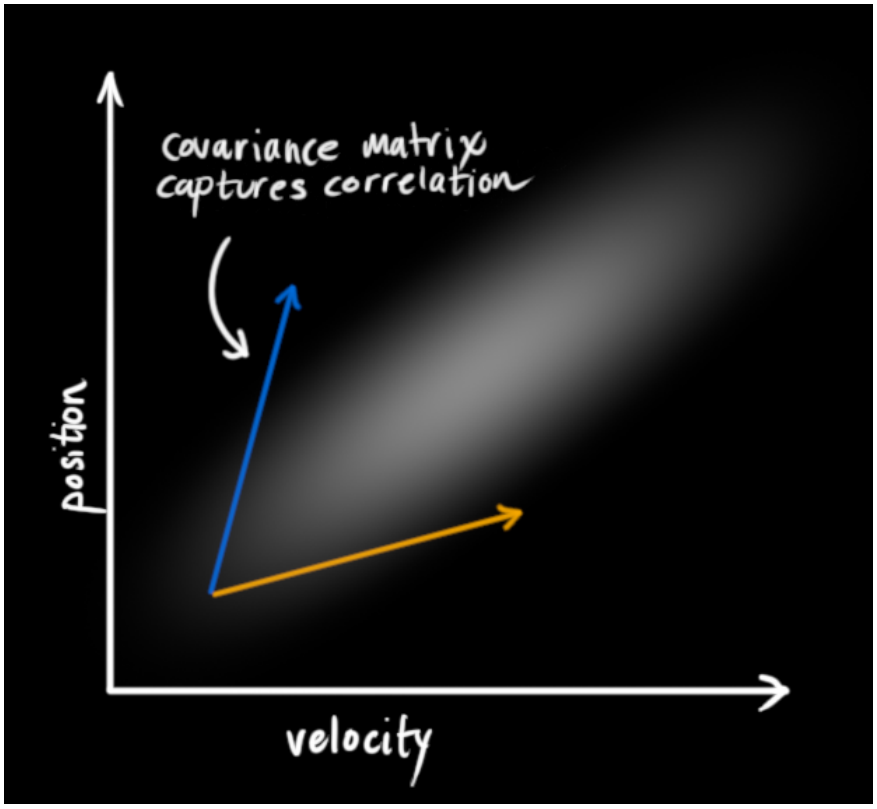
In the above picture, position and velocity are **uncorrelated**, which means that the state of one variable tells you nothing about what the other might be. The example below shows something more interesting: Position and velocity are **correlated**. The likelihood of observing a particular position depends on what velocity you have:



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

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

This correlation is captured by something called a covariance matrix. In short, each element of the matrix 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 "", so we call their elements "ij".

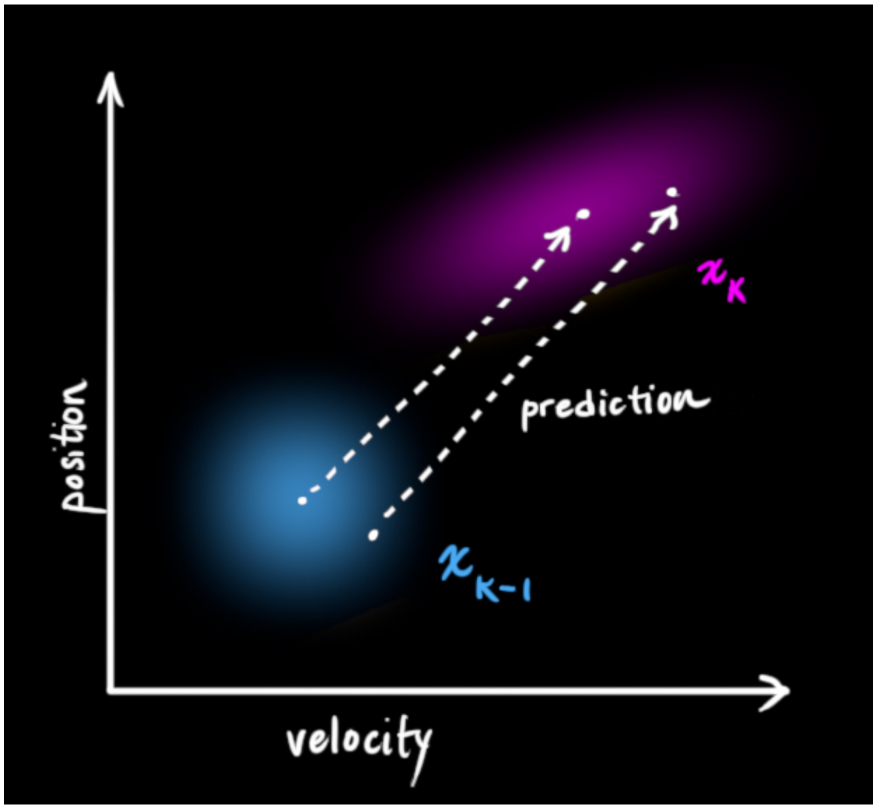


Describing the problem with matrices

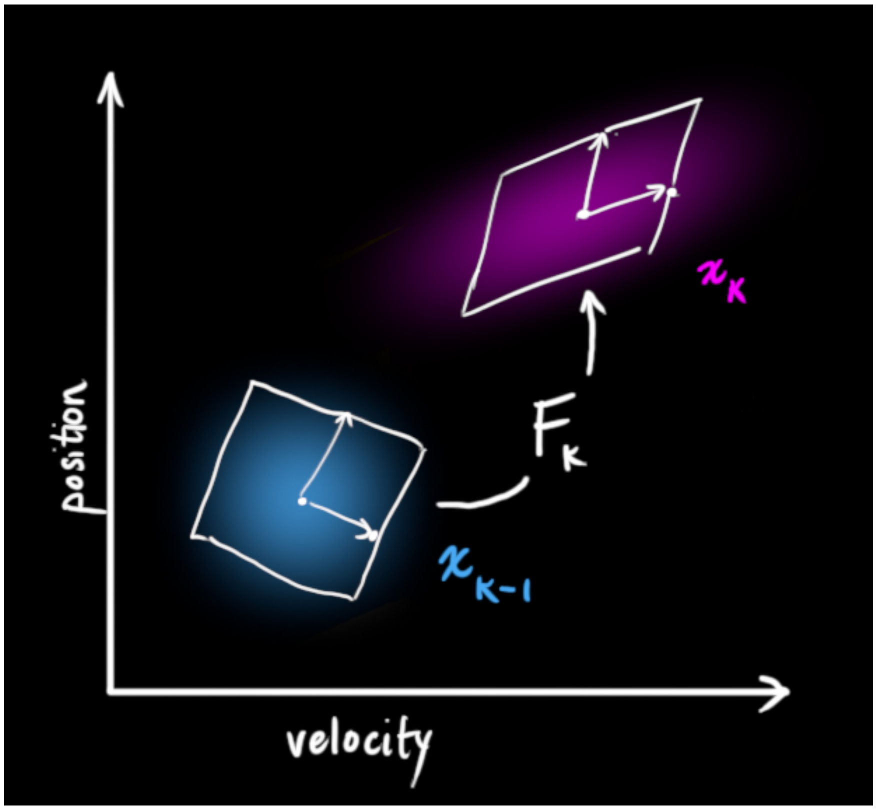
We’re modeling our knowledge about the state as a Gaussian blob, so we need two pieces of information at time k: We’ll call our best estimate x k (the mean, elsewhere named), and its covariance matrix Pk.

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

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



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



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

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

In other words:

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

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

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

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

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

So combining the upper equations:

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

External influence

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

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

In matrix form:

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

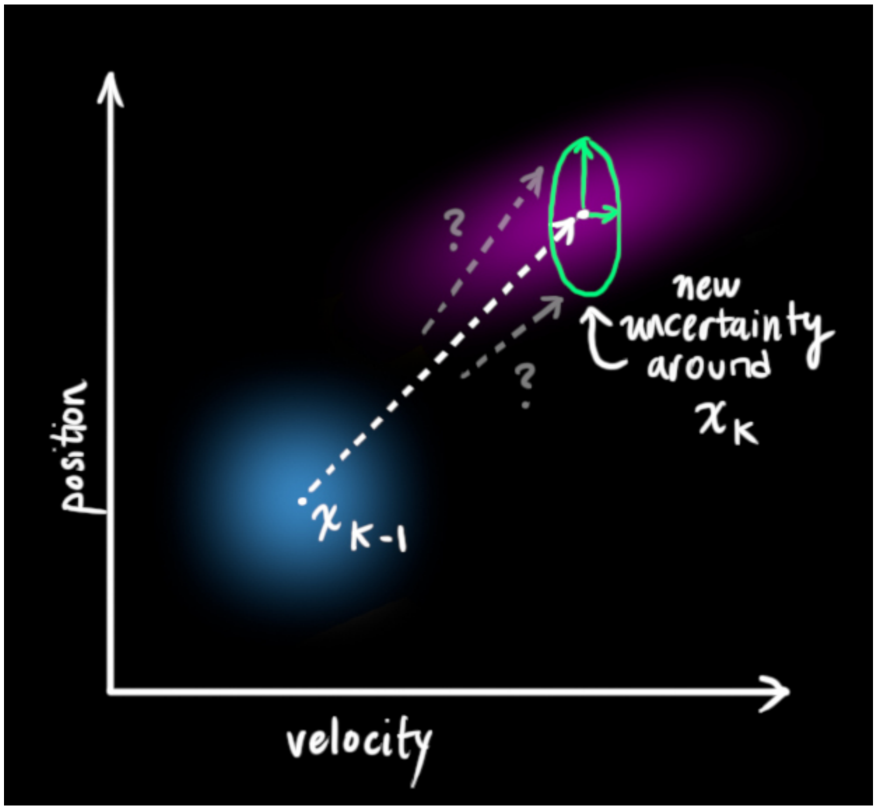
Bk is called the **control matrix** and uk the **control vector.** (For very simple systems with no external influence, you could omit these).

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

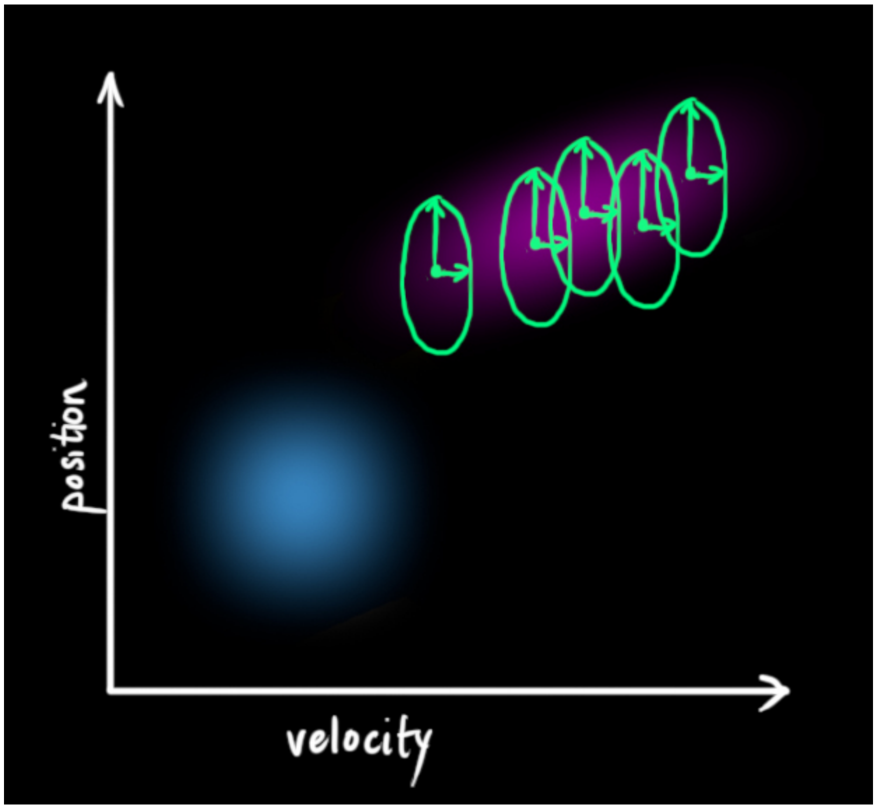
External uncertainty

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 x k1 is moved to somewhere inside a Gaussian blob with covariance Qk. Another way to say this is that we are treating the untracked influences as **noise** with covariance Qk.

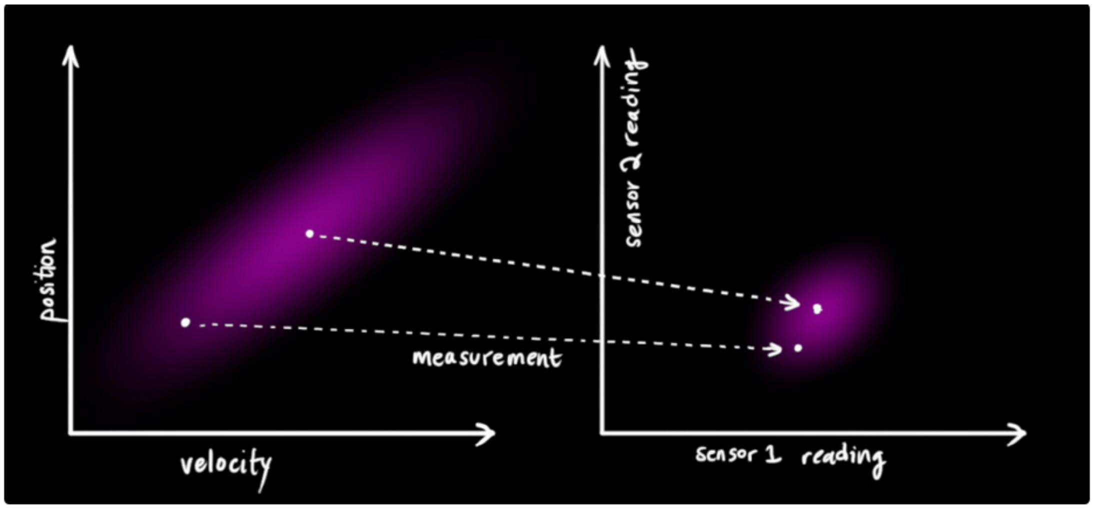


We get the expanded covariance by simply **adding** Qk, giving our complete expression for the **prediction step**:

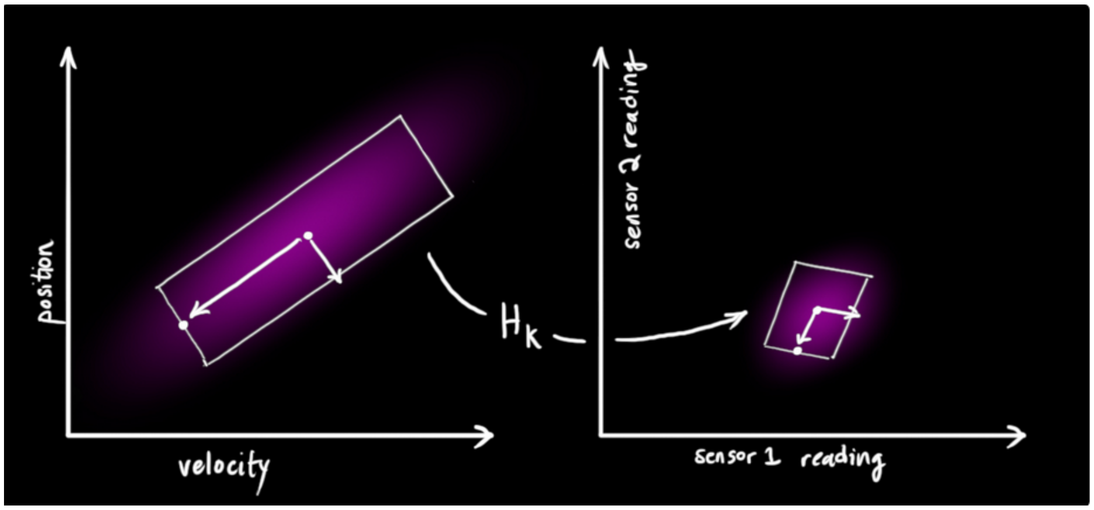
$$\begin{aligned}\hat{\mathbf{x}}_k &= \mathbf{F}_k \hat{\mathbf{x}}_{k-1} + \mathbf{B}_k \vec{\mathbf{u}_k} \\ \mathbf{P}_k &= \mathbf{F}_k \mathbf{P}_{k-1} \mathbf{F}_k^T + \mathbf{Q}_k\end{aligned}$$

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**.



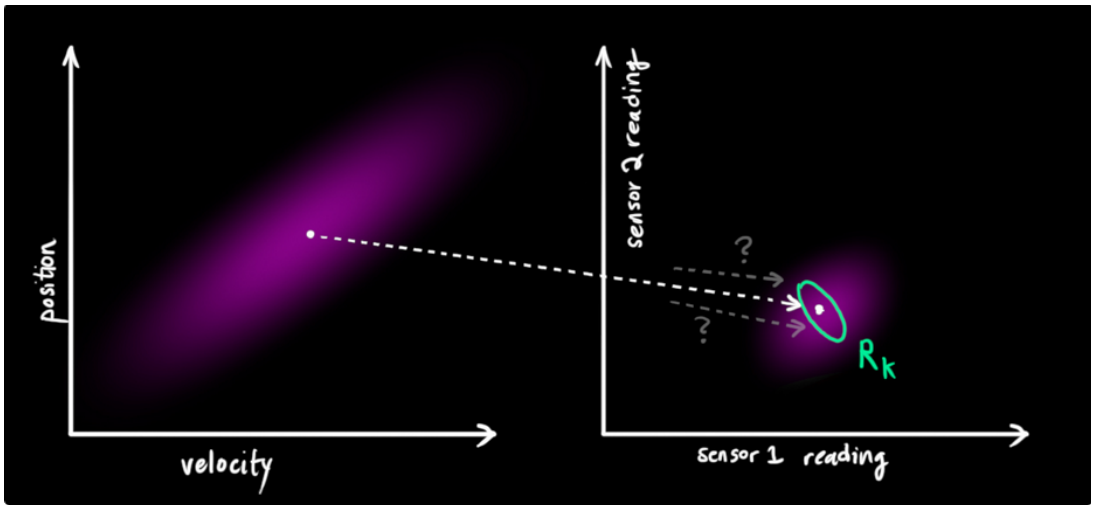
Notice that the units and scale of the reading might not be the same as the units and scale of the state we're keeping track of. You might be able to guess where this is going: We'll model the sensors with a matrix, Hk.



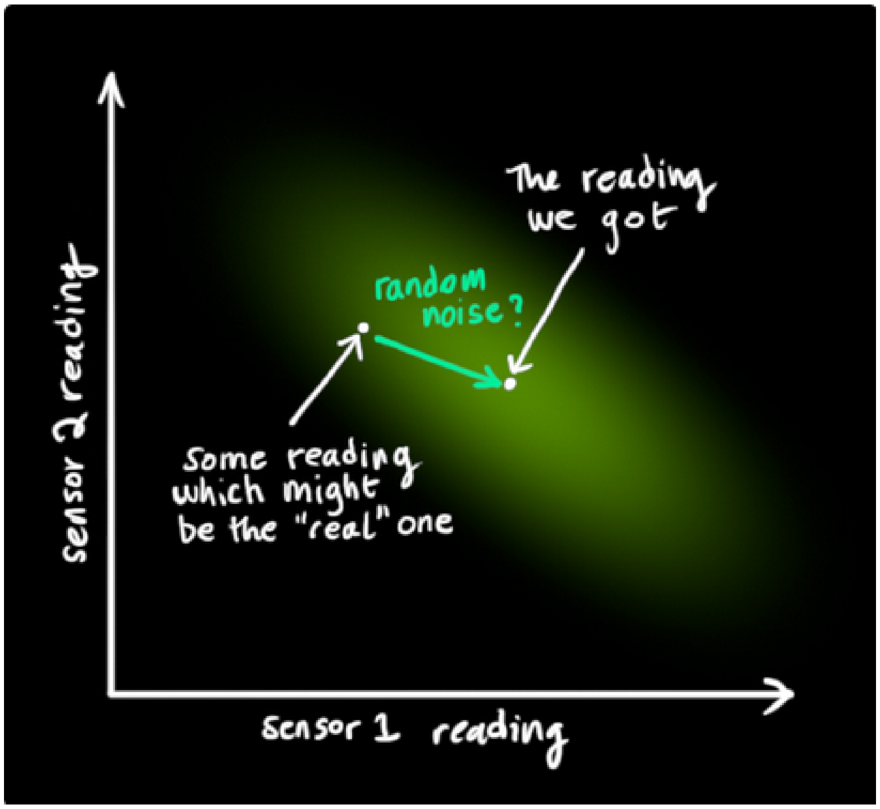
We can figure out the distribution of sensor readings we'd expect to see in the usual way:

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

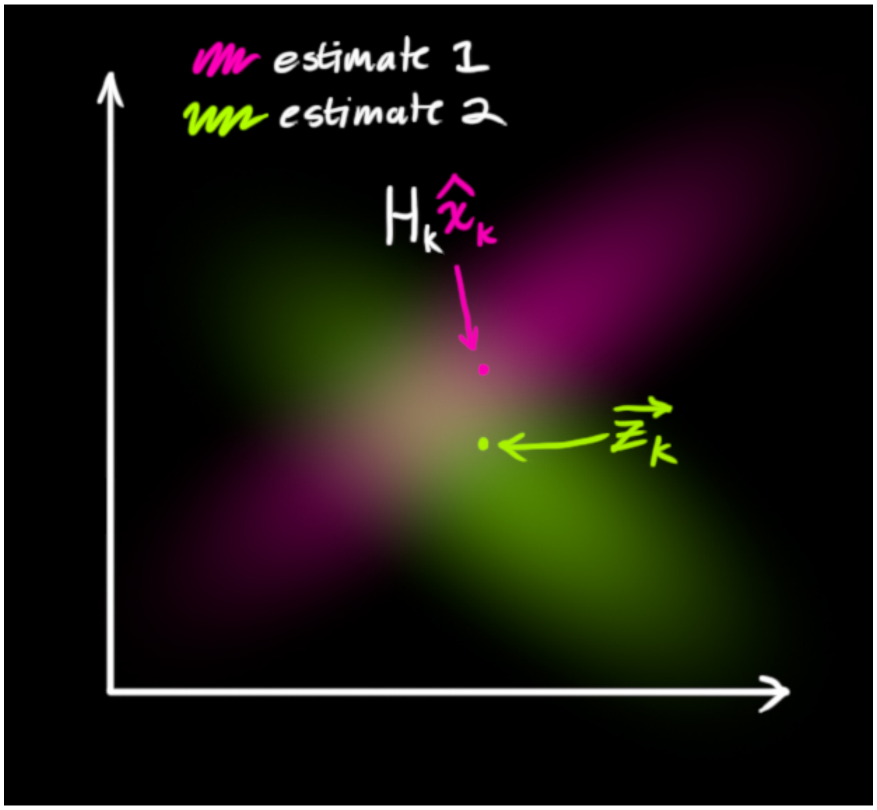
One thing that Kalman filters are great for is dealing with *sensor noise*. In other words, our sensors are at least somewhat unreliable, and every state in our original estimate might result in a *range* of sensor readings.



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



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

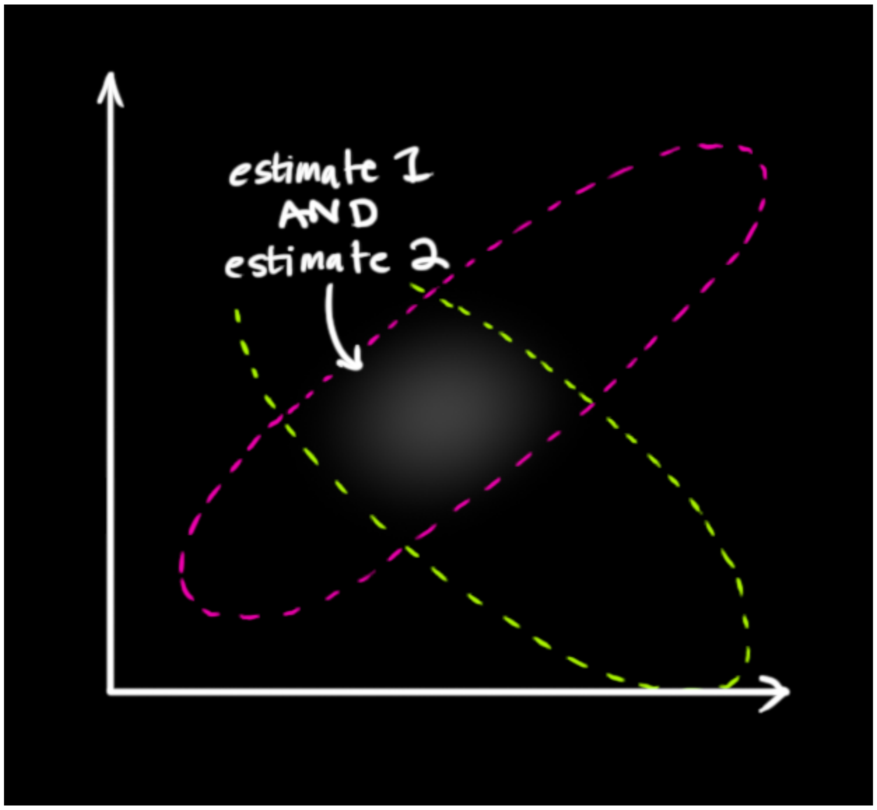
readings (**green**) that we actually observed.

So what’s our new most likely state? For any possible reading (z1,z2), we have two associated probabilities: (1) The probability that our sensor reading

zk is a (mis-)measurement of (z1,z2), and (2) the probability that our previous estimate thinks (z1,z2) is the reading we should see.

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

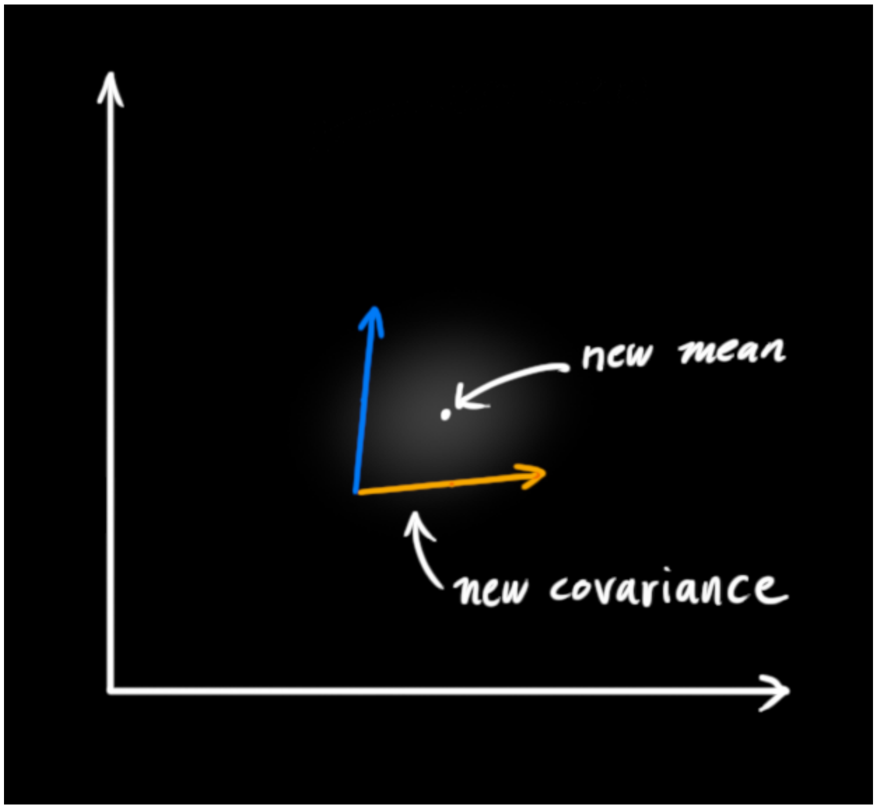
blobs and multiply them:



What we’re left with is the **overlap**, the region where *both* blobs are bright/likely. And it’s a lot more precise than either of our previous estimates.

The mean of this distribution is the configuration for which **both estimates are most likely**, and is therefore the **best guess** of the true configuration

given all the information we have.



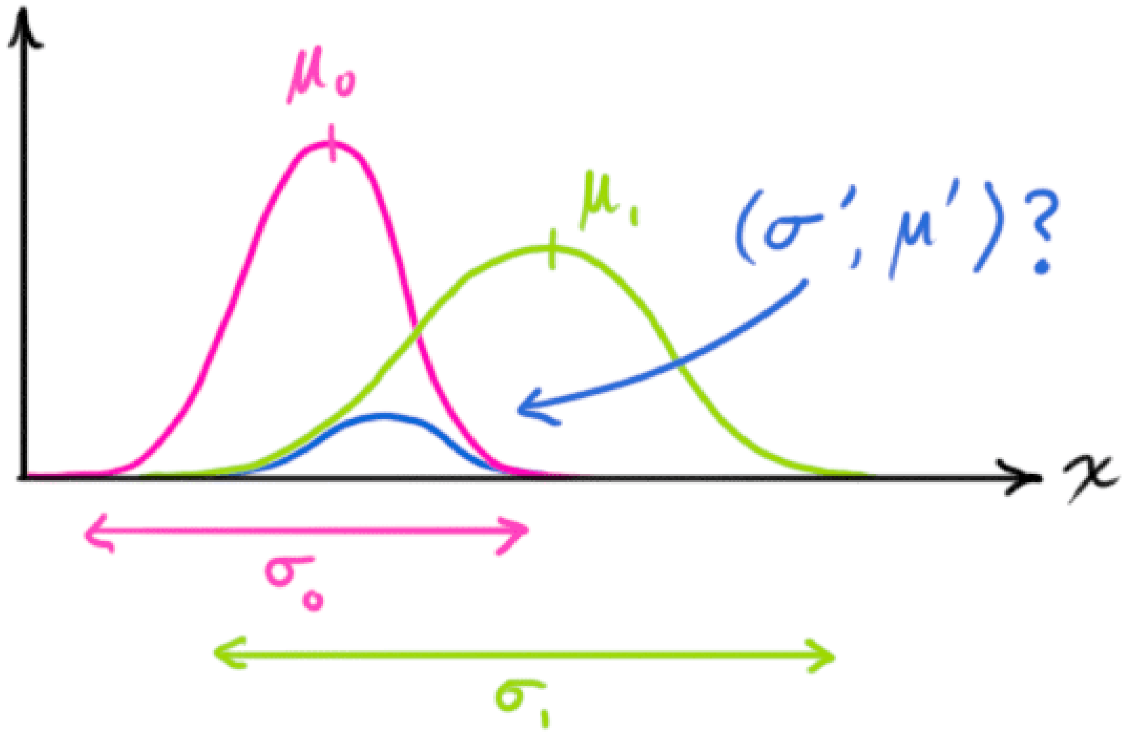
Combining Gaussians

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

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

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

of the two Gaussian populations:



Do some algebra (being careful to renormalize, so that the total probability is 1) to obtain:

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

$$\sigma'^2 = \sigma_0^2 - \frac{\sigma_0^4}{\sigma_0^2 + \sigma_1^2}$$

The matrix version:

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

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

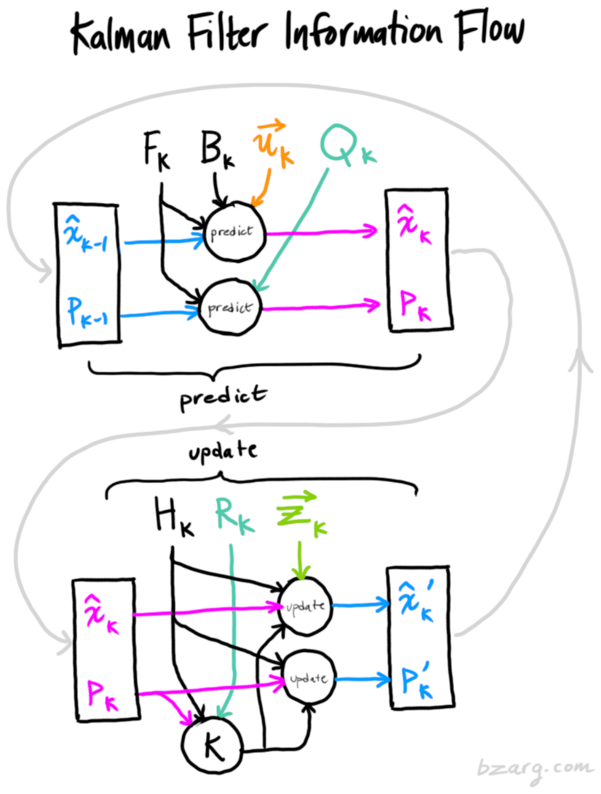
$$\Sigma' = \Sigma_0 - \mathbf{K}\Sigma_0$$

Putting it all together

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

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

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

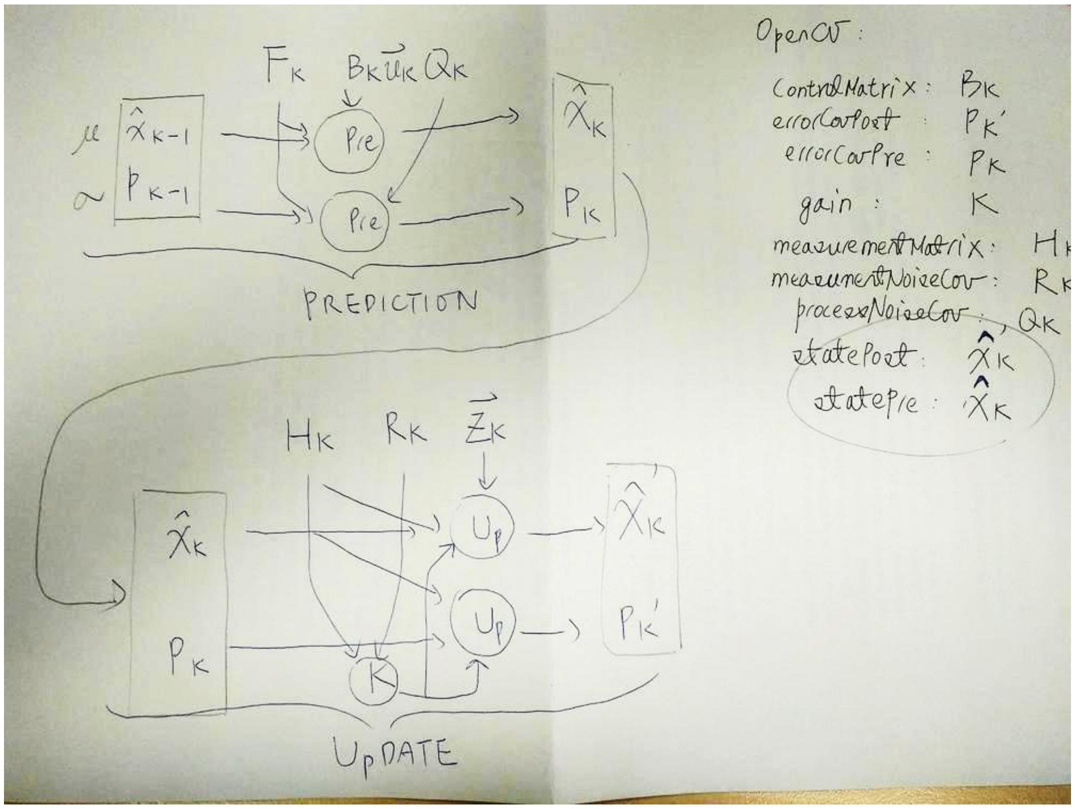


OpenCV API of Kalman Filter:

The link to the official document is shown below:

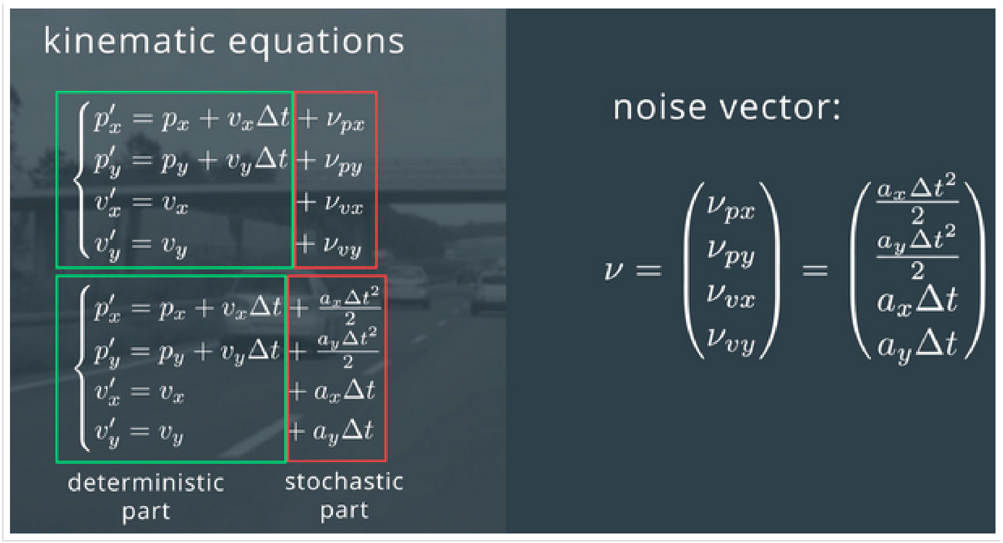
https://docs.opencv.org/trunk/dd/d6a/classcv_1_1KalmanFilter.html

The following figure shows the relationship between the OpenCV API and the kalman filter working flow:



Qk in Constant Velocity Motion Model

If we consider acceleration as a random noise with mean = 0, covariance matrix = Q



kinematic equations

$$\left\{\begin{aligned} \boxed{\begin{aligned} p'_x &= p_x + v_x\Delta t \\ p'_y &= p_y + v_y\Delta t \\ v'_x &= v_x \\ v'_y &= v_y \end{aligned}} \boxed{\begin{aligned} &+ \nu_{px} \\ &+ \nu_{py} \\ &+ \nu_{vx} \\ &+ \nu_{vy} \end{aligned}} \end{aligned}\right.$$

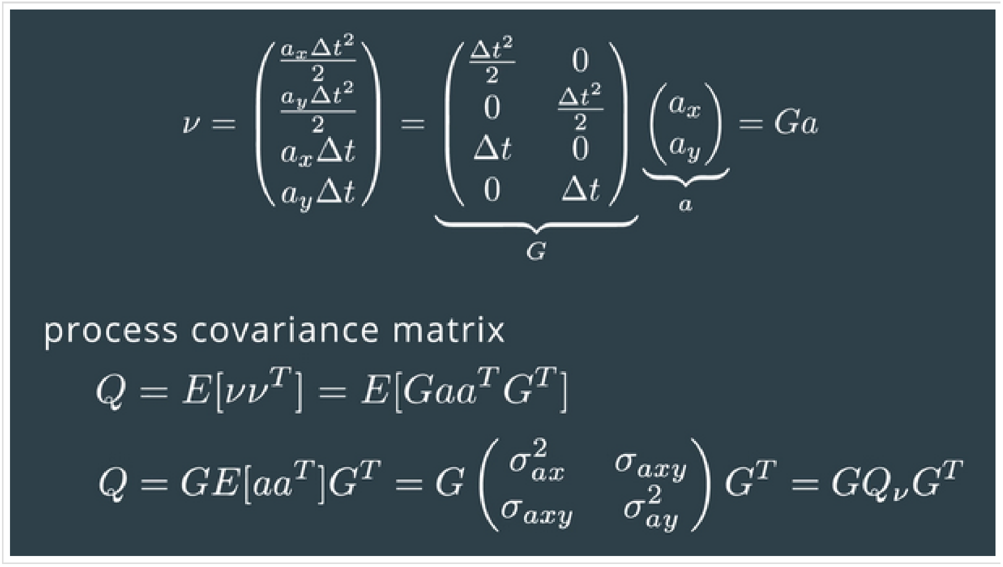
$$\left\{\begin{aligned} \boxed{\begin{aligned} p'_x &= p_x + v_x\Delta t \\ p'_y &= p_y + v_y\Delta t \\ v'_x &= v_x \\ v'_y &= v_y \end{aligned}} \boxed{\begin{aligned} &+ \frac{a_x\Delta t^2}{2} \\ &+ \frac{a_y\Delta t^2}{2} \\ &+ a_x\Delta t \\ &+ a_y\Delta t \end{aligned}} \end{aligned}\right.$$

deterministic part

stochastic part

noise vector:

$$\nu = \begin{pmatrix} \nu_{px} \\ \nu_{py} \\ \nu_{vx} \\ \nu_{vy} \end{pmatrix} = \begin{pmatrix} \frac{a_x\Delta t^2}{2} \\ \frac{a_y\Delta t^2}{2} \\ a_x\Delta t \\ a_y\Delta t \end{pmatrix}$$

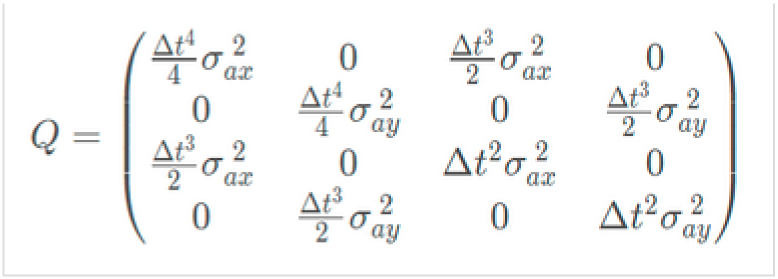


$$\nu = \begin{pmatrix} \frac{a_x\Delta t^2}{2} \\ \frac{a_y\Delta t^2}{2} \\ a_x\Delta t \\ a_y\Delta t \end{pmatrix} = \underbrace{\begin{pmatrix} \frac{\Delta t^2}{2} & 0 \\ 0 & \frac{\Delta t^2}{2} \\ \Delta t & 0 \\ 0 & \Delta t \end{pmatrix}}_{G} \underbrace{\begin{pmatrix} a_x \\ a_y \end{pmatrix}}_{a} = Ga$$

process covariance matrix

$$Q = E[\nu\nu^T] = E[Gaa^TG^T]$$

$$Q = GE[aa^T]G^T = G\begin{pmatrix} \sigma_{ax}^2 & \sigma_{axy} \\ \sigma_{axy} & \sigma_{ay}^2 \end{pmatrix} G^T = GQ_\nu G^T$$



$$Q = \begin{pmatrix} \frac{\Delta t^4}{4}\sigma_{ax}^2 & 0 & \frac{\Delta t^3}{2}\sigma_{ax}^2 & 0 \\ 0 & \frac{\Delta t^4}{4}\sigma_{ay}^2 & 0 & \frac{\Delta t^3}{2}\sigma_{ay}^2 \\ \frac{\Delta t^3}{2}\sigma_{ax}^2 & 0 & \Delta t^2\sigma_{ax}^2 & 0 \\ 0 & \frac{\Delta t^3}{2}\sigma_{ay}^2 & 0 & \Delta t^2\sigma_{ay}^2 \end{pmatrix}$$

Reference

http://www.bzarg.com/p/how-a-kalman-filter-works-in-pictures/