Using and Designing Kalman Filters

A review of how Kalman filters work, and how to use them, with some discussion of design considerations

Abstract

Without re-deriving the theory of how Kalman filters came to be, we review what they are and how to go about using them. Walking through three very basic theoretical examples, we show how state estimates are generally calculated from predictions and measurements, opening up a straightforward way of thinking about things that can make future designs easier to reason about. Following through a very basic example, we end with additional discussion of design considerations for other things like time-varying filters and extended filters for nonlinear systems.

Linear algebra gets a bad rap. Not to brag or anything, but when people on the internet argue over the epistemological efficacy of linear operator interpretations of linear algebra as inherently more intuitive, I'm the guy in the back of the class that asks, "isn't it just solving lots of algebra equations?" Of course there's tons of fantastic stuff as you go deep into the abyss of abstractions, but I personally find it way more interesting to come up for air every now and again to solve a practical problem.

Kalman filters are one such problem. The way people talk about them as a crazy mix of the worst parts of linear algebra *and* probability theory make them seem confusing, when in reality they're quite straightforward. Once you figure out how they're straightforward, you can accept the proofs of their operation or optimality as they are—within reason—to do some truly wacky things with them; which can get you even more interesting results.

For the sake of posterity, here's what we'll try to do: we'll try to figure out how to build and use Kalman filters in real life—accepting the proofs as they are, refusing to re-derive anything—but showing in reality how everything works. As such, there will be no mention of probability theory (that one MIMO course I took in grad school made me afraid of it) except for a brief discussion of why we need to keep track of variances and covariances.

Here is my attempt at a pedantic description of a Kalman filter:

> A Kalman filter is an optimal, linear, recursive filter that lets you create an estimate of the state of a system in real time based on previous estimates of the system's state, a physical model of that system used to predict the current system state, and current measurements of the system that can be related to the predictions of the system's state.
>
> At each point in time, the filter accepts the immediately previous value of the estimates of the system's state, predicts what they might be right now, and then compares them to the current measurements of the system. The estimated value of the system's state at this current point in time is then determined by a weighted average of the predicted values of the system's state and the measurements of the system. The weights used in this average are determined using the relative uncertainty of the predictions when compared to the uncertainty of the measurements.

By the way, there are of course multiple decent references for this stuff, Wikipedia being an ok place to start[1], kalmanfilter.net is really quite good[2], and of course, I personally like to use really old textbooks[3]. Oh, and then recently I found a reference that someone is building on GitHub[4].

[1] https://en.wikipedia.org/wiki/Kalman_filter
[2] https://www.kalmanfilter.net/default.aspx
[3] Stengel, Robert, "Optimal Control and Estimation." Dover Publications Inc., New York, 1994.
[4] https://github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python

One relatively straightforward example of Kalman filters is if you want to determine the position of a vehicle as it moves across a mostly flat, two-dimensional plane (like the earth's surface), but all you have are measurements from a GPS or GNSS system that gives you a scalar speed value and a heading value relative to the magnetic north pole.

Using Newton's equations of motion and a little trigonometry, assuming you know the starting position with some certainty (or at least set it to (0, 0) to, "normalize," it), it's pretty straightforward to build a physical model for this, assuming you know what the speed or acceleration of the vehicle is at any point in time. But what if you don't? What if all you have access to are assumptions about how the vehicle travels through space, and then periodic, noisy measurements of qualities that are related to what you want to know: the vehicle's position? Well, that's where the Kalman filter comes in.

First, let's get the discussion of probability theory out of the way. The nice thing about Kalman filters is that the fundamental assumption that makes this whole thing work is that while each piece that feeds the filter at every point in time (previous estimate, prediction, and measurement) has some uncertainty associated with, we assume that each uncertainty possesses the same characteristic: all noise is zero-mean, additive, white, Gaussian noise (AWGN).

The nice thing about zero-mean AWGN is that in one dimension it can be completely characterized by a single number, it's variance. With more than one dimension, multi-variate AWGN can be perfectly characterized by the variances of each individual variable, as well as the covariances characterizing how each variable relates to one another, arranged in a covariance matrix. What this means is that all we have to do to compensate for noisy measurements is keep track of some simple noise characteristics (either making them up, using reasonable guesses, or looking at an instrument's datasheet), adding them, multiplying them, and dividing them along the way as they're transformed by the physical model of our state, or in order to compare them to measurements. That's it; it's not complicated, it's just usually discussed in a confusing way. Examples further along will make things clearer.

But, before some examples, let's quickly review how the Kalman filter works at a higher level. Because a Kalman filter is recursive, it means we take series of data points, stepping through them in order, where updates at any given time are dependent on the output of the step immediately beforehand. In that way it's very similar to an infinite impulse response (IIR) filter. Now *this* means that at every time step as we march through our data, we have access to the output from the previous time step, as well measurements that we collect at this point of time. At each step, our Kalman filter then works through the following three steps:

1. **Predict:** Using the estimates determined in the previous time step, and the physical model that represents how these estimates propagate through time, predict what the state will be in this current time step. Also, using the estimated uncertainty of the state from the previous timestep, use the same physical model (and model of the model's uncertainty) to determine what the uncertainty of the predictions will be.

2. **Weigh:** Combine the predicted uncertainty of the predicted state with the uncertainty of the measurements (and how the state relates to the measurements) to determine the relative uncertainty of measurements compared to your predicted state. Relative uncertainty is literally a number between zero and one for each measurement or state prediction that can be used as weights to combine predictions and measurements.

3. **Estimate:** Using the relative uncertainties as weights, the state estimates for this time step are determined by combining—as a weighted average—the predictions determined in the first step above, and the measurements for this time step.

Ok that's a bit of a mouthful, but it's accurate, while also proof of how once you understand what's going on, the math can be a tad more succinct. Case in point, here are the above three steps in math form:

1. **Predict:**

$$\mathbf{x}_p = \mathbf{F}\,\mathbf{x}'$$
$$\mathbf{P}_p = \mathbf{F}\,\mathbf{P}'\,\mathbf{F}^{\mathrm{T}} + \mathbf{Q}$$

Where **x'** is the estimate from the previous time step, **F** is a matrix representing the model of a physical process for how our system propagates through time, $\mathbf{x}_p$ is the prediction of our state for this time step based on our process model, **P'** is the covariance matrix representing the uncertainties in the state estimates, $\mathbf{P}_p$ is the predicted covariance matrix representing the uncertainties in the state predictions, **Q** is a covariance matrix representing the uncertainty injected into the system by the assumptions of the state prediction model, and an upper-case T indicates the matrix transpose.

I should note that there are other ways of predicting the current state if you also have control inputs in your system (e.g. you can change the throttle position of your car), or there are other perturbations. This is the simplest way to predict your state, and in my work I use it because I only have measurements and state estimates, with no other controls, etc.

2. **Weigh:**

$$\mathbf{K} = \mathbf{P}_p\,\mathbf{H}^{\mathrm{T}}\left(\mathbf{H}\,\mathbf{P}_p\,\mathbf{H}^{\mathrm{T}} + \mathbf{R}\right)^{-1}$$

Where **K** is the Kalman gain, a matrix representing the relative uncertainties of measurement and prediction, **H** is a matrix representing your model for transforming your state predictions into predictions of measurements, and **R** is a covariance matrix representing the uncertainty of the measurements themselves.

3. **Estimate:**

$$\begin{aligned}\mathbf{x}_e &= \mathbf{x}_p + \mathbf{K}(\mathbf{z} - \mathbf{H}\,\mathbf{x}_p)\\ &= (\mathbf{I} - \mathbf{K}\,\mathbf{H})\mathbf{x}_p + \mathbf{K}\,\mathbf{z}\\ \mathbf{P}_e &= (\mathbf{I} - \mathbf{K}\,\mathbf{H})\,\mathbf{P}_p\,(\mathbf{I} - \mathbf{K}\,\mathbf{H})^{\mathrm{T}} + \mathbf{K}\,\mathbf{R}\,\mathbf{K}^{\mathrm{T}}\end{aligned}$$

Where $\mathbf{x}_e$ is the updated estimate of your state for this current time step (to be fed in to the filter as **x'** in your next time step), **z** is the measurements that you collected for this time step, and $\mathbf{P}_e$ is the estimated covariance matrix representing the uncertainty in your current state estimate.

The One-Dimensional Kalman Filter

For a fairly straightforward example, we'll start with a simple one-dimensional filter. While not particularly useful in practice, working through it helped me wrap my head around a few things at first. This is mostly adapted from the example found over at kalmanfilter.net. So let's say that all we want to do is estimate the speed, *v*, of something using a straightforward process and some noisy measurements. This means that our state estimate is very straightforward, at every time step we start off with a single estimate of speed that we bring over from our previous time step, i.e.:

$$\mathbf{x}' = v$$

Now, in order to develop a process for this, we have to look at the values that we're estimating and figure out a way to propagate them forward in time one step. Because we only have one estimate, our model can be only one thing: we assume that the speed doesn't change, meaning that the matrix **F**, representing how we turn previous estimates into new predictions is a simple constant:

$$\mathbf{F} = 1$$

If we had access to say, an estimate of acceleration, this would be slightly more complex, but we'll cross that bridge when we get there. Now, in order to figure out the process noise, we have to first realize that if all we ever have is a constant speed, then we're not doing anything interesting. But since our process model is such that speed never changes, how can we account for any changes in speed due to our process? Why, by adding in a little bit of uncertainty, that's how. I mean, if we have a constant speed, but with error bars of +/- 10%, then there's plenty of uncertainty to account for a reasonable change in speed within each time step. So let's do that, let's assume that there's AWGN on top of our speed process, that takes the form of a single variance such that:

$$\mathbf{Q}_a = \sigma_v^2$$

Where the subscript *v* indicates that this uncertainty (variance) is associated with our speed estimate. Now in order to see how this is affects the noise throughout our process model, including other parameters, we can project it forward in time to make a prediction of the process noise using the arguments from the kalmanfilter.net[5] website for, "Projection Using the State Transition Matrix," such that we find the uncertainty injected into our prediction by the process:

$$\mathbf{Q} = \mathbf{F}\,\mathbf{Q}_a\,\mathbf{F}^{\mathrm{T}} = \sigma_v^2$$

Where since our process to predict how the state evolves in time is just a constant number, the number one, our process uncertainty is still just the uncertainty in the speed itself. This is also discussed from a different perspective in that one book on that GitHub page[6] I found. There are of course a number of ways to figure out what your process noise is, e.g. using uncertainties into the inputs you use to control your system, or modelling noise in parameters that you don't also want to estimate. I prefer it to do this way, as in my work I'll never have any control into the system itself, and I also want to find estimates of the things that happen to be major sources of process noise.

Dealing with the measurement side of the modelling is easy. First, since we only want to estimate speed from noisy speed measurements, our measurement vector, **z**, is also a scalar such that:

$$\mathbf{z} = v_z$$

With the subscript *z* indicating a measurement. And then of course the uncertainty in our measurements, given by the measurement covariance matrix is also a scalar value:

$$\mathbf{R} = \sigma_z^2$$

Where the subscript *z*, is there again to denote measurement. Now, while it can seem a bit confusing, another way that we can describe the operation of a Kalman filter is by comparing measurements with predictions of measurements. This is usually written down as the measurement residual, something like:

$$\mathbf{y} = \mathbf{z} - \mathbf{H}\,\mathbf{x}_p$$

The residual is important because it allows us to ignore probability theory some more. Without getting too into the weeds, **z** and $\mathbf{H}\,\mathbf{x}_p$ are technically identical, except **z** includes some extra AWGN. So, because we

[5] https://www.kalmanfilter.net/covextrap.html

[6] R. Labbe, "Kalman and Bayesian Filters in Python." Chapter 7, Section 3, Page 243, "Design of the Process Noise Matrix." https://github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python/blob/master/07-Kalman-Filter-Math.ipynb

cannot predict what the value of the noise will be in any given moment, we can only seek to minimize the effect of the noise through our process that translates state predictions into state measurements, and the resulting estimates by way of the Kalman gain. The way this is done is by minimizing the square of the measurement residual, **y**, as it's the only thing we can know in any meaningful way—which is what the Kalman filter does. Which is why **H** is so important, without a way to map state predictions to measurements, we're not able to build an optimal estimator. For a full discussion, please see the textbook[7].

From here, the estimates are found by multiplying the residual by the Kalman gain, and adding it to the state predictions so that the estimates are:

$$\mathbf{x}_e = \mathbf{x}_p + \mathbf{K}\,\mathbf{y}$$

Where the Kalman gain also takes the measurement residual, and maps it back into the domain of your state predictions. But, if you add these twists and turns into the above third step in our three step Kalman filter process, you can see that they're equivalent. Ergo, it can be said that in order to compare measurements and estimates via something like a residual, **y**, you first have to make a prediction of your measurements using the predictions of your estimate. That's where the matrix **H** comes in, it's a straight linear model that maps your state predictions into the same space as your measurements. For our simple model where we're estimating speed using speed measurements, it's a very straightforward model: they're both the same! This means that **H** is also quite simple:

$$\mathbf{H} = 1$$

Now we're ready to walk through the Kalman filter equations for one single time step to see how it combines measurements, previous estimates, predictions, and your physical model of the system. Let's start with some initial conditions. Say at time zero we know with absolute certainty that we're travelling at a specific speed:

$$\mathbf{x}_e\big|_{t=0} = \mathbf{x}' = v_0$$

Where absolute certainty means that our estimate's covariance is set to zero, so:

$$\mathbf{P}_e\big|_{t=0} = \mathbf{P}' = 0$$

Now, we just work through our three steps. First, we have to use our physical model to create a prediction of what our speed should be:

$$\mathbf{x}_p = \mathbf{F}\,\mathbf{x}' = 1 \cdot v_0 = v_0 = v_p$$

Which again, because our model is very simple—assuming constant speed—isn't that exciting. However, things start to get interesting when we move to predict the uncertainty of this prediction:

$$\mathbf{P}_p = \mathbf{F}\mathbf{P}'\mathbf{F}^{\mathrm{T}} + \mathbf{Q} = 1 \cdot 0 \cdot 1 + \sigma_v^2 = \sigma_v^2$$

While it's true that our estimate was known with absolute certainty, our process adds some uncertainty, which affects what we eventually trust more, predictions or measurements. Now, to figure out how to weigh one versus the other, we have to determine our Kalman gain:

[7] Stengel, Robert, "Optimal Control and Estimation." Dover Publications Inc., New York, 1994. Ch. 4 *Optimal State Estimation*, pp. 299.

$$\begin{aligned}\mathbf{K} &= \mathbf{P}_p \mathbf{H}^{\mathrm{T}}(\mathbf{H}\mathbf{P}_p \mathbf{H}^{\mathrm{T}} + \mathbf{R})^{-1} \\ &= \sigma_v^2 \cdot 1(1 \cdot \sigma_v^2 \cdot 1 + \sigma_z^2)^{-1} \\ &= \frac{\sigma_v^2}{\sigma_v^2 + \sigma_z^2}\end{aligned}$$

Which is the hardest part of the linear algebra, but as you can see it actually ends up being pretty straightforward for this simple example. To see what this actually means though, we substitute it into our estimate step, getting an estimate of our current speed:

$$\begin{aligned}\mathbf{x}_e = v_e &= \left(1 - \frac{\sigma_v^2}{\sigma_v^2 + \sigma_z^2}\right) v_p + \frac{\sigma_v^2}{\sigma_v^2 + \sigma_z^2} v_z \\ &= \frac{\sigma_z^2}{\sigma_v^2 + \sigma_z^2} v_p + \frac{\sigma_v^2}{\sigma_v^2 + \sigma_z^2} v_z\end{aligned}$$

Which if you stare at it long enough is just... a weighted average of prediction and measurement. In the extreme, if the uncertainty in measurement is large, only the prediction matters. Conversely, if the uncertainty in the process is large, only the measurement matters. If the uncertainty in both is equal, then both prediction and measurement count for 50% of the estimate. Similarly, the uncertainty of the estimate ends up being something quite similar:

$$\mathbf{P}_e = \left(\frac{\sigma_z^2}{\sigma_v^2 + \sigma_z^2}\right)^2 \sigma_v^2 + \left(\frac{\sigma_v^2}{\sigma_v^2 + \sigma_z^2}\right)^2 \sigma_z^2$$

Which is really the exact same weighted sum if you think of everything as standard deviations, then squared to get variances. Remember all of this is Gaussian, so that explanation more or less works. Now, what if there was some initial uncertainty in the estimate, like:

$$\mathbf{P}_e|_{t=0} = \mathbf{P}' = \sigma_e^2$$

Then the estimate of speed for this time speed is still a weighted sum of prediction and measurement:

$$\mathbf{x}_e = v_e = \frac{\sigma_z^2}{\sigma_e^2 + \sigma_v^2 + \sigma_z^2} v_p + \frac{\sigma_e^2 + \sigma_v^2}{\sigma_e^2 + \sigma_v^2 + \sigma_z^2} v_z$$

Which also makes sense, because uncertainty in the previous estimate means more uncertainty in the prediction, which would of course add to the weighting of the measurement. And again, if the uncertainty in the previous estimate is large, then your estimate for this time step is going to be dominated by your measurement.

Sensor Fusion: The 1.5-Dimensional Kalman Filter

This example isn't quite at all useful for anything but demonstration of a single point: it's possible to use measurements from multiple sensors to improve your estimates. That's why this is 1.5-dimensional: it estimates only one quantity, but uses two measurements to do so. Following a similar scheme as the one-dimensional case, we will run through a single iteration of the filter in order to obtain a single estimate of speed. That means that like above, we'll start out with a previous estimate of speed that is:

$$\mathbf{x}' = v$$

Where we'll also assume we know this speed with zero uncertainty, so that its variance going into this iteration of the filter is zero:

$$\mathbf{P}' = 0$$

Which also means that in order to predict how our estimate will evolve in time, the model of our state has to be a constant, as before:

$$\mathbf{F} = 1$$

Meaning of course that our predictions are the same as estimates from the previous moment in time:

$$\mathbf{x}_p = v = v_p$$

But we can allow our estimate to change by introducing uncertainty into the state time evolution process such that:

$$\mathbf{Q}_a = \mathbf{Q} = \sigma_v^2$$

Which means that the variance on our prediction, given that we knew the speed with absolute certainty at the previous time step, only depends on the uncertainty in our process, so our prediction process variance is again:

$$\mathbf{P}_p = \sigma_v^2$$

Now, for the first time, our bolded letters start to make sense as we need to start talking about measurements. In this example we are going to have two, speed and acceleration, and we keep track of them in the single **z** vector so that we can do linear algebra with them:

$$\mathbf{z} = \begin{bmatrix} v_z \\ a_z \end{bmatrix}$$

Where the subscript *z* is there to denote measurement, distinguishing them from speed estimates, predictions, or previous values. Now, the whole point of the Kalman filter is that it minimizes the measurement residual, **y**, given by:

$$\mathbf{y} = \mathbf{z} - \mathbf{H}\,\mathbf{x}_p$$

Where the matrix **H** translates our state prediction, $\mathbf{x}_p$, into the same domain as measurements so that they can be compared—effectively our state predictions become measurement predictions via the matrix **H**. Now, the only difference between actual measurements, **z**, and their predictions, **H x**, is the noise that's present, and of course it's that noise, that uncertainty, that we're minimizing. Because we assume from the beginning that all of our sources of noise are zero mean AWGN, we can perfectly represent the noise in the covariance matrix, **R**. But, how do we figure out how to translate from predictions to measurements? Simple: we assume that it's possible, just for a moment, for the residual to be zero such that we can see:

$$\mathbf{z} = \begin{bmatrix} v_z \\ a_z \end{bmatrix} = \mathbf{H} v_p$$

Which now requires us to ask the question: how do we turn a scalar speed into a vector of speed and acceleration? At this point in time, the only information we have access to is our measurements and our speed prediction. So if the residual is somehow zero, then obviously there is also no difference between speed measurement and speed prediction, so the relationship between them is singularly constant. Now, if we had access to previous values of speed estimates, perhaps we could figure out a way to translate a speed prediction to an acceleration measurement, but for now we'll just assume that they differ by only a regular constant, alpha, giving us:

$$\mathbf{H} = \begin{bmatrix} 1 \\ \alpha \end{bmatrix}$$

We'll discuss what alpha could be later. Now, all we need is to figure out what the uncertainty in our measurements are. For that, we'll need a covariance matrix, not just a single variance. Something that holds on to the variances of each measurement in and of itself, but also the correlation between them. Now, in our case we don't have to worry about covariances, just regular variances (the diagonal elements in a covariance matrix) because for this example obviously we're using two different sensors: a GPS speed sensor and an accelerometer in a completely separate device. So, it's not possible for the noise in these sensors to be correlated (even though the measurements of course will be) because they're completely different devices. This means that the uncertainty in our measurements given by the covariance matrix, **R**, is simply:

$$\mathbf{R} = \begin{bmatrix} \sigma_{zv}^2 & 0 \\ 0 & \sigma_{za}^2 \end{bmatrix}$$

Where the subscript *zv* denotes speed measurement and the subscript *za* denotes an acceleration measurement. The rest of this calculation is just moving symbols around, so to get the Kalman gain we have:

$$\begin{aligned} \mathbf{K} &= \mathbf{P}_p \mathbf{H}^{\mathrm{T}} \left(\mathbf{H} \mathbf{P}_p \mathbf{H}^{\mathrm{T}} + R \right)^{-1} \\ &= \mathbf{P}_p \mathbf{H}^{\mathrm{T}} \left(\begin{bmatrix} 1 & \alpha \\ \alpha & \alpha^2 \end{bmatrix} \sigma_v^2 + \begin{bmatrix} \sigma_{zv}^2 & 0 \\ 0 & \sigma_{za}^2 \end{bmatrix} \right)^{-1} \end{aligned}$$

Which is a pain, but we can do it because the inverse of a two-dimensional matrix is relatively simple to do:

$$\begin{bmatrix} \sigma_v^2 + \sigma_{zv}^2 & \alpha\sigma_v^2 \\ \alpha\sigma_v^2 & \alpha^2\sigma_v^2 + \sigma_{za}^2 \end{bmatrix}^{-1}$$

$$= \frac{1}{(\sigma_v^2 + \sigma_{zv}^2)(\alpha^2\sigma_v^2 + \sigma_{za}^2) - \alpha^2\sigma_v^4} \begin{bmatrix} \alpha^2\sigma_v^2 + \sigma_{za}^2 & -\alpha\sigma_v^2 \\ -\alpha\sigma_v^2 & \sigma_v^2 + \sigma_{zv}^2 \end{bmatrix}$$

Which means that our Kalman gain then becomes something slightly simpler:

$$\mathbf{K} = \frac{\sigma_v^2}{\sigma_v^2\sigma_{za}^2 + \alpha^2\sigma_{zv}^2\sigma_v^2 + \sigma_{zv}^2\sigma_{za}^2}\begin{bmatrix}\sigma_{za}^2 & \alpha\sigma_{zv}^2\end{bmatrix}$$

In order to actually make sense of the use of this, it's handy to go back to our previous relation between estimates, predictions, measurements, and the Kalman gain:

$$\begin{aligned}\mathbf{x}_e &= \mathbf{x}_p + \mathbf{K}(\mathbf{z} - \mathbf{H}\,\mathbf{x}_p) \\ &= (\mathbf{I} - \mathbf{K}\,\mathbf{H})\mathbf{x}_p + \mathbf{K}\,\mathbf{z}\end{aligned}$$

Because our gain is a 1 x 2 vector, our measurements are a 2 x 1 vector, and our estimate is a scalar, it's handy that our multiplying state predictions by the measurement prediction matrix **H** gives us a 2 x 1 vector, then multiplying again by a 1 x 2 Kalman gain vector gives us a scalar. Similarly for actual measurements multiplied by the gain. Putting it all together, our state estimate, the optimal calculation of our speed at this moment in time, becomes:

$$\mathbf{x}_e = \frac{\sigma_{zv}^2\sigma_{za}^2\,v_p + \sigma_v^2\sigma_{za}^2\,v_z + \alpha\sigma_v^2\sigma_{zv}^2\,a_z}{\sigma_v^2\sigma_{za}^2 + \alpha^2\sigma_{zv}^2\sigma_v^2 + \sigma_{zv}^2\sigma_{za}^2}$$

Can you tell where the sensor fusion is? In the one-dimensional filter, our estimates were a weighted sum simply trading off uncertainties between the two. But now we have a third quantity, an acceleration measurement, that's added to the mix in equal measure. It's hard to tell, but squinting at this equation hard enough, you can see that if the uncertainty in acceleration measurement is very high, then the trade-off becomes exactly the same as before, a weighted sum between speed prediction and speed measurement. Similarly for speed prediction and measurement. Also, if it turns out that we can't figure out how to translate a speed scalar into an acceleration scalar, thus setting alpha to be zero, then we again have the exact same thing as in the one-dimensional case: a weighted sum between speed predictions and measurements. However, if all of the uncertainties are the same, then we use all three in equal measure which is why we call this sensor fusion: we can use two related measurements to give us a better estimate.

It's not obvious if sensors are being fused here because this is a useless example. Why? Because it's not possible to make a reasonable linear estimate of acceleration from past estimates. How? We made our state transition model of speed to literally be constant, so by definition any prediction of speed is constant, meaning zero acceleration, meaning alpha has to be, by definition, zero. And when alpha is zero? Our estimate is identical to the one-dimensional case.

Also, if you were to flip this around, using speed and acceleration measurements to estimate acceleration, you'd have a similar problem: you'd just end up comparing acceleration predictions to measurements, much for the same reason. In order to use a speed measurement to improve your acceleration estimate, you'd need a way to relate a scalar acceleration to a scalar speed, which is not possible. You can certainly shoe-horn something in here to make it work, but at the risk of introducing instability into your filter. This is a great example to work through though, so we shall leave it as an exercise for the reader.

Let's move on to a better example.

Real Sensor Fusion: The 2-Dimensional Kalman Filter

The two-dimensional Kalman filter is called that because for the first time so far, the matrix that uses state predictions to get measurement predictions, **H**, needs to be two-dimensional. That is, this example estimates

two quantities, speed and acceleration, while also having access to measurements of two quantities, also speed and acceleration. This means that the estimates from our previous time step will be:

$$\mathbf{x}' = \begin{bmatrix} v' \\ a' \end{bmatrix}$$

And like before, we assume to make the math easier, that these estimates are known absolutely so have no uncertainty meaning that the covariance matrix, **P**, is zero:

$$\mathbf{P}' = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

Now to move our estimates forward in time, we're just going to use Newton's equations of motion, but we still have to assume that our acceleration is constant. If we collect everything in terms of linear algebra this gives us:

$$\mathbf{x}_p = \begin{bmatrix} v_p \\ a_p \end{bmatrix} = \begin{bmatrix} v' + a'\Delta t \\ a' \end{bmatrix} = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix} \begin{bmatrix} v' \\ a' \end{bmatrix}$$

Now because Newton's equations of motion are linear in terms of speed and acceleration, it's very easy to separate it out into a simple linear algebra matrix multiplied by a vector. For more complex systems where the predictions are not straight linear functions of the estimates from the previous time step, it's also possible to turn it into matrix-vector multiplication by way of Jacobians, but that's the domain of the Extended Kalman Filter. So, for us, this means that the matrix that progresses our estimates through time to get state predictions is:

$$\mathbf{F} = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix}$$

Now, what to do about the state process noise? Because our state process has one variable that remains constant from step to step, acceleration, it definitely needs to have some uncertainty otherwise there's no way we can get the measurements to allow it to change according to measurement. What about speed; should there be uncertainty in the speed process model? There is no good reference on this, but after looking at how everyone does it, I'm going to go ahead and say one thing: no. Why? Because in order for our process to be meaningful—and to encourage us to build good processes—we must assume the uncertainty in predictions that are dependent on other estimates (e.g. speed predictions depend on previous speed estimates *and* acceleration estimates) is zero.

A good heuristic to use is that wherever in your model you have a quantity that remains constant from step to step, it requires some uncertainty in order to change based on measurements, but any prediction dependent on those constants will only have uncertainty informed by the process model. So in this case where we assume constant acceleration, acceleration needs an uncertainty, but not speed since speed gets its prediction, and thus uncertainty, from a process that requires acceleration. This is similar to the previous example where we needed an uncertainty in our speed prediction because our underlying model assumed a constant speed along time steps.

This means that:

$$\mathbf{Q}_a = \begin{bmatrix} 0 & 0 \\ 0 & \sigma_a^2 \end{bmatrix}$$

Where we remember that $\mathbf{Q}_a$ is a covariance matrix representing uncertainty such that the diagonals are the uncertainties in the process predictions, and the off-diagonal terms are the relevant covariances. Again, this process only has uncertainty in the acceleration. We can now use our state model matrix, **F**, to inform the uncertainty in the dependent variables (e.g. speed that's dependent on acceleration). This means that the total process noise after it's been marched ahead in time becomes:

$$\mathbf{Q} = \mathbf{F}\mathbf{Q}_a\mathbf{F}^{\mathrm{T}} = \sigma_a^2 \begin{bmatrix} (\Delta t)^2 & \Delta t \\ \Delta t & 1 \end{bmatrix}$$

Interestingly, this implies, since the uncertainty in our previous estimates is zero, that the uncertainty in our speed prediction is simply:

$$\sigma_v^2 \Rightarrow (\Delta t)^2 \sigma_a^2$$

This is important because our speed uncertainty, if the time step is less than one, will be significantly smaller than the acceleration uncertainty. This is why Kalman filters are so powerful: we assume our processes are infinitely certain, except for their inputs that we have to assume are constant from moment to moment, but with some uncertainty in that process. That is, we only need to assume there's uncertainty in the constants that feed the model of our state. This means that usually estimates that come out of a good state process model will be less uncertain than the measurements, which can really help us straighten some things out. That's also why it's so important to get the model right.

Again, because the uncertainty in the previous estimates, **P'**, is set to be zero, this means that the uncertainty in all of the predictions is the same as the state process uncertainty:

$$\mathbf{P}_p = \sigma_a^2 \begin{bmatrix} (\Delta t)^2 & \Delta t \\ \Delta t & 1 \end{bmatrix}$$

Since this is a basic two-dimensional filter, remember that we're estimating the same two things as we're measuring, speed and acceleration, such that our measurement vector is:

$$\mathbf{z} = \begin{bmatrix} v_z \\ a_z \end{bmatrix}$$

It's worth repeating that the goal of the Kalman filter is to minimized the squared residual, or difference between noisy measurements and estimate predictions. That is, we're trying to minimize:

$$\mathbf{y} = \mathbf{z} - \mathbf{H}\,\mathbf{x}_p$$

Thankfully, this helps us gain some insight into how we can construct our matrix, **H**, so that it can translate predictions from their domain to the domain of measurements so that it makes sense to compare measurements and predictions. To do this, of course we first assume that our measurements have zero noise, meaning that the residual is zero, so that:

$$\mathbf{z} = \mathbf{H} \begin{bmatrix} v_p \\ a_p \end{bmatrix}$$

Now, the predictions for speed already take into account the acceleration from the previous time step, so the simplest way to predict measurements from estimates, given that we're estimating both quantities that we're measuring, is easily:

$$\mathbf{H} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Next is the uncertainty in the measurements represented by, **R**, which is also a covariance matrix. Using the same arguments as before (the sensors are uncorrelated, with uncertainty represented only by variances), we're left with a measurement uncertainty of:

$$\mathbf{R} = \begin{bmatrix} \sigma_{zv}^2 & 0 \\ 0 & \sigma_{za}^2 \end{bmatrix}$$

Finally, all we have to do is follow through on the math to find the gain, and then the estimates. So the gain is just a matter of working out:

$$\begin{aligned} \mathbf{K} &= \mathbf{P}_p \mathbf{H}^{\mathrm{T}} \left(\mathbf{H} \mathbf{P}_p \mathbf{H}^{\mathrm{T}} + R \right)^{-1} \\ &= \mathbf{P}_p \mathbf{H}^{\mathrm{T}} \left(\begin{bmatrix} (\Delta t)^2 & \Delta t \\ \Delta t & 1 \end{bmatrix} \sigma_a^2 + \begin{bmatrix} \sigma_{zv}^2 & 0 \\ 0 & \sigma_{za}^2 \end{bmatrix} \right)^{-1} \end{aligned}$$

Which is a total pain, but eventually gets us:

$$\mathbf{K} = \frac{\sigma_a^2}{(\Delta t)^2 \sigma_a^2 \sigma_{za}^2 + \sigma_a^2 \sigma_{zv}^2 + \sigma_{zv}^2 \sigma_{za}^2} \begin{bmatrix} (\Delta t)^2 \sigma_{za}^2 & \Delta t\, \sigma_{zv}^2 \\ \Delta t\, \sigma_{za}^2 & \sigma_{zv} \end{bmatrix}$$

We then put this straight into our equation to update our estimates for this time step. Now, because we're estimating two quantities, our estimate vector, $\mathbf{x}_e$, will be a 2-by-1 vector. However, because we're now doing real linear algebra, things are complicated, so I'm only going to write down the estimate for speed, which is:

$$\begin{aligned} v_e =& \frac{\left(\sigma_{zv}^2 \sigma_a^2 + \sigma_{zv}^2 \sigma_{za}^2\right) v_p + \Delta t \sigma_a^2 \sigma_{za}^2\, a_p}{(\Delta t)^2 \sigma_a^2 \sigma_{za}^2 + \sigma_{zv}^2 \sigma_a^2 + \sigma_{zv}^2 \sigma_{za}^2} \\ &+ \frac{(\Delta t)^2 \sigma_a^2 \sigma_{za}^2\, v_z + \Delta t \sigma_a^2 \sigma_{zv}^2\, a_z}{(\Delta t)^2 \sigma_a^2 \sigma_{za}^2 + \sigma_{zv}^2 \sigma_a^2 + \sigma_{zv}^2 \sigma_{za}^2} \end{aligned}$$

I'm pretty sure that's right. It's hard to tell, but this is still a straight up weighted sum. For example, the speed predictions are weighted such that if the uncertainty in the speed measurement *and* acceleration prediction are very high, the speed prediction will be favoured. Also, if the speed measurement *and* acceleration

measurement are very high, the speed prediction will be favoured. Then, if the uncertainty in the measured acceleration is high *and* the uncertainty in the speed process noise, i.e. the uncertainty in the acceleration prediction multiplied by the squared time step, is very high then the acceleration prediction will be favoured. Similarly for both measurements. What's nice to know here is that (if I did the math right) everything being summed to create the new speed estimate has the appropriate units of *m / s*. That's the beautiful thing about sensor fusion: if you set up your state process model and prediction model and measurement prediction model correctly, all of the units work out and everything is just a weighted sum.

My favourite part is that this is *almost* as complicated as any weighted sum usually gets. Think about it: the only things that will ever get summed together in any given estimate are ones that can be related either via a state process or measurement prediction. So here we have four terms, one for each measurement and estimate. If we wanted to get total distance travelled as an estimate on top of the 2-dimensional Kalman filter, its updated estimate would have five total terms: distance prediction, speed prediction, acceleration prediction, speed measurement and acceleration measurement. All of the other estimates would only have four terms because while there is a way to relate speeds and accelerations to distances, there is no neat and tidy linear way to relate a distance prediction to a speed or acceleration measurement. Now, if we were able to measure distance as well, then it's easy to see that every estimate would have at least six terms, three each for predictions and measurements.

Example Two-Dimensional Filter

For a very simple example of how this works in practice we'll compare implementations for the one-dimensional Kalman filter above with a two-dimensional filter. The one-dimensional filter will estimate speed using noisy speed measurements and a constant process model. The two-dimensional filter will estimate the speed and acceleration using noisy speed and acceleration measurements, but with a process model that uses constant acceleration and then accumulates acceleration estimates to predict speed.

For both filters we'll set things up so that we receive a measurement and update our estimate every one hundredth of a second, or at 100 Hz. Well also set up speed measurement uncertainty, that is its variance, to be 4 m/s. This means that in our theoretical work through above, we'll have:

$$\sigma_{zv}^2 = 4 \text{ m/s}$$

This is arbitrary and it seems quite high compared to say, GPS or other GNSS systems, but very noisy measurements help us prove our point about the usefulness of these filters so we'll keep it as is. Now, for the process model uncertainty, what should we do?

Assume for a moment you're driving a car on the highway and you take your foot off the gas for literally one second. How much does the speed change? Maybe less than 5 km / h? Less? What if you're not driving so fast and there's less wind resistance, how much would your speed change if you were driving at 40 km / h and took your foot off the gas pedal for literally one second? Definitely less than 5 km / h. So let's use our engineering intuition and say that in any given second, there is a 1 m / s (or 3.6 km / h) uncertainty in our speed process. This means that in any given time interval, the uncertainty in the model of our physical process will be equal to that time interval. Relating this back to the one-dimensional theory above we have:

$$\sigma_v^2 = \Delta t \text{ m/s}$$

Now since the physical process model and the measurement prediction model of the one-dimensional filter are both constant, all we need is to initialize our estimates and create some data before we're reading to see how the one-dimensional filter works. But before that let's discuss the two-dimensional filter.

Really quickly, let's make up a number for uncertainty for the measurement of acceleration:

$$\sigma_{za}^2 = 1 \text{ m/s}^2$$

Why this number? Because this is a contrived example so we're allowed to make up numbers for measurements. Also, we assume there is no correlation between speed and acceleration measurement *noise* because accelerometers are separate devices from GPS systems measuring speed. But how are the uncertainties in the physical process model affected? Remember from the two-dimensional derivation above that the predicted uncertainty injected into our estimates by the physical process model (assuming initial estimate uncertainty of zero) is given by:

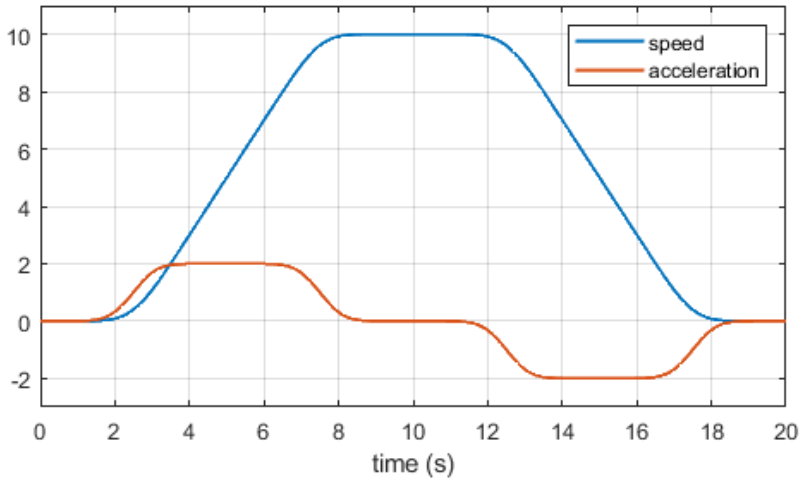
$$\mathbf{P}_p = \sigma_a^2 \begin{bmatrix} (\Delta t)^2 & \Delta t \\ \Delta t & 1 \end{bmatrix}$$

So to make a fair comparison, we need the uncertainty in the constant acceleration process, σ^2_a, to be such that the uncertainty in the speed process is the same, namely equal to the time step, Δt. That means that the acceleration process uncertainty must be:

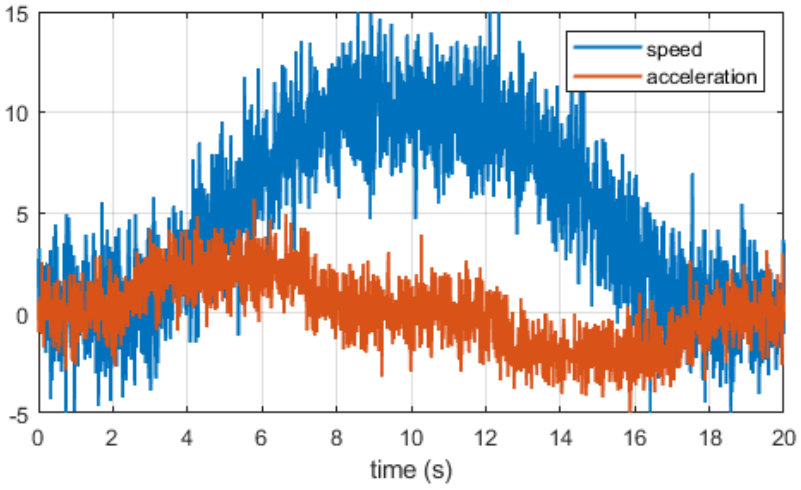
$$\sigma_a^2 = \frac{1}{\Delta t} \text{ m/s}^2$$

Which at 100 Hz is a truly gargantuan uncertainty. This effectively turns our two-dimensional filter into a 1.5-dimensional filter—as the acceleration prediction is so uncertain it's ignored—with both speed and acceleration measurements used in the speed estimate. I still think it ends up being a good example though.

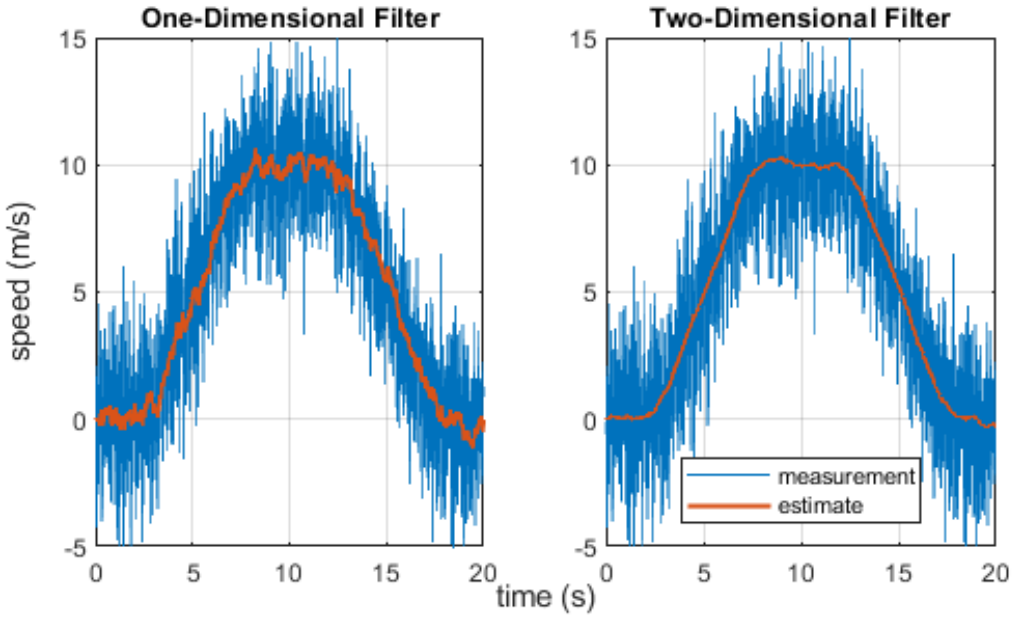
In Appendix A, both of these filters are implemented with some simple MATLAB code. First, we create an acceleration and speed profile that's mostly realistic, such as:



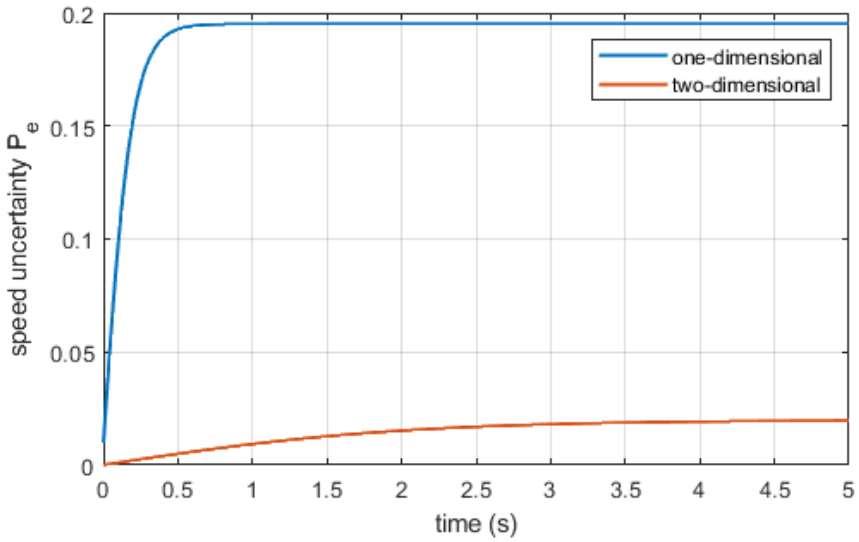
While this is a contrived example, it's not a terrible representation of what you might experience as you accelerate from a stop sign and then decelerate a few seconds later as you approach the next one. Now, what we do is add some Gaussian noise to it, using the variances as discussed above, to get us:



Now all we have to do is set up initial conditions, define the models, and loop through this data, once for each filter. Since all of our models are pretty straightforward, there's not much left to do here but take a look at the results:



Which are pretty good! Even the one-dimensional case gets us excellent results for a simple problem like this, where all of the models and processes are well defined. However, by simply adding in acceleration measurements we can get noticeably better results. But how much better? The easiest way to determine that is to look at how the uncertainty in the estimates evolves over time. That is, since we're focusing on speed estimates here, how does that element of $\mathbf{P}_e$ change as we work through filtering all of the measurements? That's easy enough as one of the outputs of the Kalman filter at each time step is the estimate uncertainty, so plotting the speed estimate uncertainty over time gets us:



While not obvious this shows some pretty interesting results.

By simply adding in acceleration measurements—remembering that the acceleration process is too uncertain to be useful based on our setup—we can decrease the uncertainty in our speed estimate by about ten times! This is because in this contrived example, our process model perfectly matches the physics of the actual acceleration and speed. If our model were off, and say there was another part of the process that was left unaccounted for, the speed uncertainty would definitely be higher; e.g., if there were say a constant bias in the acceleration or something, or a constant bias in the speed. This would mean that everything would be slightly off, and in order for estimates and processes to account for that, the uncertainties would end up being higher.

Finally, the estimate uncertainties start out at zero (because we initialized them to be zero) and then converge to their final value after some time. For simple models like ours, initial conditions—as long as they're in the right ballpark—don't matter that much to the final result. However, as the model for how your state advances through time becomes more complex, estimate uncertainty takes longer to converge. This means that for more complex models, initial conditions can play a large role in the overall accuracy of your filter if they're way off base. This can be partly mitigated by initializing things to effectively match the first measurements that you collect, or at least initializing with a larger uncertainty so that measurements are trusted more than estimates at first.

Some Notes on Process, Measurement, Prediction & Noise

While physical process models are for the most part based on real-life physics, e.g. kinematics equations, there is a lot more flexibility when figuring out what the noise is going to be. So while the physical process you choose is very important, as related quantities will end up being fused, and the measurement prediction process is very important, as related quantities will end up being fused, they're mostly set in stone by physical reality. Even though you should probably be guided by the actual noise characteristics of, e.g., your sensors, you can go bonkers with noise models to do some interesting things. How's this? Because:

1) Kalman filters are weighted sums of quantities, meaning that uncertainties are always *relative*, so don't worry about absolute uncertainty so much, only focus on the relative uncertainty between process and measurement. That is to say, 1/100 is the same as 10/1000; and,
2) When things are in the computer, you can do what you want super easily to experiment so don't be afraid to do weird things; however,
3) Large deviations from the standard linearized Kalman filter models (e.g. for state or measurement prediction), can result in filter instability.

This means that given a certain physical process, once you make reasonable assumptions about the uncertainties in the process, you can use reasonable assumptions about measurement noise as long as the measurement uncertainties are in the same ballpark as those of the process uncertainty. This allows us to focus our design efforts, perhaps counter-intuitively, on figuring out under what conditions we want to favour the physical process model, and under what conditions we want to favour the measurements we collect. For example, if we have conditions where we want to favour prediction over measurement, we just make the measurement uncertainty incredibly high relative to process uncertainty while those conditions are met.

Coming back to the car situation as a quick thought experiment: if your accelerometer (not including gravity) shows an extreme acceleration event, does that mean that your car has crashed, or that someone has kicked your accelerometer? Can we use other sensors to help us decide when to trust process over measurements or vice versa? What if your sensors showed an extreme acceleration event, but the GPS said your velocity didn't change at all? I think that probably means that someone kicked something they shouldn't have; or, perhaps the car crashed and the GPS was thrown from the car.

So what information do we need, at a bare minimum, in order to figure all of this out and make these decisions? What if we want to estimate something that we don't have noisy measurements for?

Estimate Fusion: The Other 1.5-Dimensional Kalman Filter

Seeing how filters work when you want to estimate something that you don't have a measurement for will help us get some insight in order to better reason about designing processes and noise models. The simplest example of this is if we wanted to estimate both speed and acceleration, but only had access to speed measurements. This means that our estimates would take the form:

$$\mathbf{x}' = \begin{bmatrix} v' \\ a' \end{bmatrix}$$

Which of course means that our measurements are just:

$$\mathbf{z} = v_z$$

Now using a simple physical process to advance these estimates through time as in the previous examples (assume a constant acceleration), our process model is:

$$\mathbf{F} = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix}$$

But since we only have access to speed measurements, we have to discard acceleration predictions when calculating the residuals, such that our measurement prediction model looks like:

$$\mathbf{H} = \begin{bmatrix} 1 & 0 \end{bmatrix}$$

Where we also assume our process uncertainty due to the constant acceleration model is the same as in previous examples, as is the measurement uncertainty for speed. We also assume zero uncertainty in the estimates from previous time steps to make the math simpler. Now skipping ahead a few steps, calculating the prediction uncertainty, the Kalman gain, and then the final estimates for this step in our process, we find our speed estimate to be:

$$v_e = \frac{\sigma_{zv}^2 \, v_p + (\Delta t)^2 \sigma_a^2 \, v_z}{(\Delta t)^2 \sigma_a^2 + \sigma_{zv}^2}$$

Which is a weighted sum of speed prediction and speed measurement, and then following the same steps for our acceleration estimate we get:

$$a_e = a_p - \frac{\Delta t \sigma_a^2 \, (v_p - v_z)}{(\Delta t)^2 \sigma_a^2 + \sigma_{zv}^2}$$

Where just staring at this it's easy to see that as the uncertainty in your measurements gets incredibly large, then both speed and acceleration will be based solely on their predictions. But now, the acceleration prediction isn't a weighted sum as before, it's a combination of the prediction itself that is corrected using the speed prediction and speed measurement. The meaning of this can be seen in the limit where process uncertainty gets arbitrarily large:

$$\begin{aligned} \lim_{\sigma_a^2 \to \infty} a_e &= a_p - \frac{1}{\Delta t}(v_p - v_z) \\ &= a' - \frac{1}{\Delta t}(v' + \Delta t \, a' - v_z) \\ &= \frac{1}{\Delta t}(v_z - v') \end{aligned}$$

Remember how in the last 1.5-dimensional Kalman filter we said that it's not possible to turn a scalar speed into a scalar acceleration in order to compare two measurements with one prediction? Well, it turns out that's ok if you're in the opposite situation, with two estimates and one measurement, because the Kalman filter will use your process model to determine how the one measurement can be used for both estimates. For example, with acceleration you end up with a finite difference (i.e. discretized instantaneous derivative) of the current measured speed, and the speed estimate from the previous time step.

Now, flipping things around, what if we want to use just acceleration measurements instead of speed? That would mean that our measurement vector becomes:

$$\mathbf{Z} = a_z$$

And then the measurement prediction matrix is:

$$\mathbf{H} = \begin{bmatrix} 0 & 1 \end{bmatrix}$$

Where if we leave everything the same as above then follow through with the math to find the speed estimate we end up with:

$$v_e = v_p - \frac{\Delta t \sigma_a^2 (a_p - a_z)}{\sigma_a^2 + \sigma_{za}^2}$$

Which in the limit of infinite process uncertainty becomes:

$$\begin{aligned}\lim_{\sigma_a^2 \to \infty} v_e &= v_p - \Delta t(a_p - a_z) \\ &= v' + \Delta t\, a' - \Delta t(a_p - a_z) \\ &= v' + \Delta t\, a_z\end{aligned}$$

Which is of course just updating the speed prediction from your previous time step; not using the prediction from your physical process model, but using the measurements *informed* by your physical process model. It's a pretty neat trick.

Not having a model to map predictions to measurements can also be thought of as saying that those measurements, should they exist, have infinite uncertainty. However, since there is a physical process model linking those predictions with their measurements, we can use both the measurements and predictions to improve estimates. Of course, if you do not assume zero uncertainty in previous estimates, there will usually be a term in here that increases the uncertainty in the speed prediction, making the speed measurement more valuable. Similarly for the acceleration estimate above.

Another example is if you want to measure speed from acceleration and distance measurements. If you only include a speed estimate in your process model, you're out of luck, as there's no way to relate your measurements to your estimates. However, if you also include estimates for distance and acceleration, you can build a process model that relates all three, with a straightforward relationship between predictions and measurements. The final result is that your estimate will be based on a speed prediction corrected with acceleration measurements *and* predictions as well as with distance measurements *and* predictions.

One final thing to note before we move on is: when you build a process model for predictions and you don't have direct measurements for their estimates, what you're predicting (and estimating) has to be directly connected to what you're measuring. That is, your physical process model has to provide a direct connection between a quantity that you're estimating *and* measuring to the one that you're estimating but have no measurement for.

For example, if you have acceleration measurements, and you want to estimate total distance travelled using a constant acceleration process model, you'll need at least three estimates: acceleration, speed, and distance. This is because speed is directly related to acceleration, but total distance travelled requires both speed and acceleration estimates.

Let's see how this works through a broken process model. Let's see if we can find an estimate for distance, using a constant acceleration process model, where speed is directly related to acceleration and distance in the usual way, but distance is *only* related to speed, and not acceleration. That would mean that in order to propagate the estimates from our previous time step to get predictions for this time step, we would have:

$$\begin{bmatrix} d_p \\ v_p \\ a_p \end{bmatrix} = \begin{bmatrix} 1 & \Delta t & 0 \\ 0 & 1 & \Delta t \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} d' \\ v' \\ a' \end{bmatrix}$$

Meaning that our physical process matrix, **F**, is simply:

$$\mathbf{F} = \begin{bmatrix} 1 & \Delta t & 0 \\ 0 & 1 & \Delta t \\ 0 & 0 & 1 \end{bmatrix}$$

But since we only have acceleration measurements, our measurement prediction matrix, **H**, is:

$$\mathbf{H} = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}$$

And because we're using a constant acceleration model, our process uncertainty is:

$$\mathbf{Q}_a = \sigma_a^2 \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Where, remember, the uncertainty in our predictions is given by:

$$\mathbf{P}_p = \mathbf{F}\mathbf{Q}_a\mathbf{F}^{\mathrm{T}} + \mathbf{F}\mathbf{P}'\mathbf{F}^{\mathrm{T}}$$

But to keep things simple we assume the uncertainty in previous estimates is zero, such that the uncertainty in our predictions becomes:

$$\mathbf{P}_p = \sigma_a^2 \begin{bmatrix} 0 & 0 & 0 \\ 0 & (\Delta t)^2 & \Delta t \\ 0 & \Delta t & 1 \end{bmatrix}$$

Ok, now we have to pay attention. See how the first row is all zeros as is the first column? If you ever see a row of all zeros, it means that the estimate for that row (in this case, for distance) will *only* be based on the prediction from your physical process model. Similarly, if you ever see a column of all zeros, it means that the prediction that relates to that column (in this case, also distance), will not be used to correct for any other predictions or measurements. The easiest way to see this is by following through a bit more with the math such that the Kalman gain is:

$$\mathbf{K} = \frac{\sigma_a^2}{\sigma_a^2 + \sigma_{za}^2} \begin{bmatrix} 0 \\ \Delta t \\ 1 \end{bmatrix}$$

Because the estimate for this time step uses your measurements multiplied by this gain, you can see straight away that the first row (i.e. distance estimate) will not use our scalar acceleration measurement. If for simplicity, we make the following substitution:

$$\alpha = \frac{\sigma_a^2}{\sigma_a^2 + \sigma_{za}^2}$$

We can then see how the final estimate uses our predictions, since our predictions are multiplied by:

$$\mathbf{I} - \mathbf{K}\mathbf{H} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & -\alpha\Delta t \\ 0 & 0 & 1 - \alpha \end{bmatrix}$$

And since the top row is where our distance estimate will be, we can see that when we multiply this quantity by our vector of predictions (with distance on the top), the only thing used in the distance estimate will be that prediction. But because we deliberately used the wrong physical process model, we know that this distance prediction will be wrong, and it will also have a huge uncertainty associated with it, making it effectively useless.

Putting this altogether, what this means is that if you have multiple measurements that you want to use to create your estimates, then you need at least that many estimates; however, as long as you have a direct link through a physical process model, you can use a single measurement to improve multiple estimates.

Oh, and here's one good heuristic to use when you want to quickly check if your physical process model is working correctly. Since matrices are hard to invert by hand, if you want to see what measurements are going to eventually get fused with what predictions in order to get you your final estimates, you just need to look at the prediction uncertainty matrix, $\mathbf{P}_p$. If there is a row that's all zeros, only the prediction associated with the estimate for that row will be used to determine the final estimate, there will be no fusion of other estimates or any measurements. If there's a column that's all zeros, the prediction associated with that column will not be used in any estimates. Everything else will be linked up via the process model and measurement prediction model.

Playing With Noise: Time-Varying Kalman Filters

When you read through the literature on Kalman filters, the one thing that always struck me as odd was that the only thing anyone ever says about designing noise models (either process or measurement) is that they need to be determined through, "engineering considerations." But what does *that* mean? I think the reasonable interpretation is, "get an engineer to look at the noise in the sensors, think about the process, and then just make up some numbers that make sense." And there's nothing wrong with that!

However: through our work so far, we've seen that uncertainties (due to noise) are used to determine estimates as *relative* weightings. We've also seen how process models and measurement prediction models determine how these weightings get shuffled around to be used. So, when thinking about say, the 2D filter above—estimating and measuring both speed and acceleration—we can start asking questions like, "when do I want to trust acceleration measurements the most in order to estimate speed?" Or, "when do I want to trust speed measurements the most in order to estimate acceleration?" That is, instead of looking at some uncertainty parameters in our sensors or making educated guesses about uncertainty given our models—treating them as constants—we can take a step back and look at the entire problem that we're trying to solve.

But, in order to do that, we need to be able to adjust our models as the underlying conditions change. That is, any uncertainties that we use (via variances and covariances) need to be able to change within every step of our Kalman filter. Again, one of the many benefits of doing this in the computer is that we can do whatever the hell we want, so this is not a problem.

One of the easiest ways to demonstrate this is by noticing that the horizontal accuracy of your GPS is loosely related to how fast you're going. Thankfully, most GPS devices report not only coordinates but horizontal accuracy as well. Accuracy can be all over the place, and maybe it's even better at high speed because highways tend to have fewer tall buildings surrounding them, but it generally gets better the faster you go. The easiest way to model this is with a simple clamping function, that can take the form based on what I see coming out of my terrible GPS device:

$$\sigma^2 = \max\left(3,\ \frac{9}{1+0.25v}\right)$$

Where sigma is the uncertainty in position readings (long/lat coords), and *v* is the current speed at this step of the filter. At zero speed, the uncertainty will be nine, but as speed increases it will slowly decrease until a minimum of 3, at 8 m/s, whereafter that's the lowest it can go.

Now, this engineering design process is easy to understand: stare at sensor, make up noise model; but, is there anything we can learn from it? After all, if these values are still incredibly large compared to what we assume the uncertainty in our process model is, these measurements won't be used in our estimates. There's not a lot of good discussion out there that I can find, so please feel free to play around and see what makes sense for you in whatever situation you find yourself stuck within.

Extended Kalman Filters

The reason we've been able to work through the examples so far with such ease isn't just because they're silly contrived examples, but because for the most part they're straightforward, i.e. linear. That is, when you want to make a prediction using Newton's equations of motion, they don't require quadratic or cubic versions of the inputs, i.e. the estimates from the previous time steps in the filter. But what if the process model you use to make your predictions (either of estimates or measurements) is nonlinear? For example, if you have a rotation, or you want to find the magnitude of something. Well, then you need an Extended Kalman Filter.

Extended Kalman filters are not strictly optimal in the sense of minimizing the residual, **y**, but they work well enough in most situations—and they're so easy to work with or fast to compute—that they're widely used in industry. An extended Kalman filter pretty much takes your nonlinear process or measurement prediction model and linearizes it by way of a Jacobian matrix (or Hessian and beyond for higher order filters).

What does that mean? Let's take a look. What if to predict the current state of your estimate using your process model instead of coming up with a matrix to multiply a vector by, you just had a whole bunch of functions? Like, you have one function that takes in all your previous estimates and calculates distance, and another completely different one for speed? Well, we can write that down like this:

$$\mathbf{x}_p = f(\mathbf{x}') = \begin{bmatrix} f_1(\mathbf{x}') \\ \vdots \\ f_n(\mathbf{x}') \end{bmatrix}$$

Where each row of the function vector, *f*, uses estimates from the previous time step to get a prediction of the estimate for this time step. That's all well and good, but the filtering process uses the matrix, **F**, to figure out how to mix uncertainties to ultimately find estimates for this time step. So how do we find **F**? We use the vector of functions above, and calculate the Jacobian matrix. The Jacobian matrix is just a bunch of derivatives all organized in such a way that each row is the derivative of the function for that prediction, and each column is the derivative with respect to the associated estimate. That is a terrible explanation, but the math version looks like:

$$\mathbf{F} = \frac{\mathrm{d}f}{\mathrm{d}\mathbf{x}'} = \begin{bmatrix} \frac{\mathrm{d}f_1}{\mathrm{d}x'_1} & \cdots & \frac{\mathrm{d}f_1}{\mathrm{d}x'_n} \\ \vdots & \ddots & \vdots \\ \frac{\mathrm{d}f_n}{\mathrm{d}x'_1} & \cdots & \frac{\mathrm{d}f_n}{\mathrm{d}x'_n} \end{bmatrix}$$

Just be aware that my notation is a bit different from the literature (and Wikipedia). As you can see, the first row is the first function in our function vector, *f*, but each column of that row is the derivative with respect to a different input estimate. You then repeat that process for each function in your function vector, getting you a nice square matrix. The easiest way to prove to you that this works is to keep using our contrived example from above. So let's see what this looks like for a filter where we want to estimate both speed and acceleration. We have two estimates going into our function, so our prediction vector looks something like:

$$\mathbf{x}_p = f(\mathbf{x}') = \begin{bmatrix} f_1(\mathbf{x}') \\ f_2(\mathbf{x}') \end{bmatrix} = \begin{bmatrix} v' + \Delta t a' \\ a' \end{bmatrix}$$

Which is not surprising. Previously we just stared at this for a while to figure out what **F** might be, but now that we know extended Kalman filters exist, we can use their process, which is:

$$\mathbf{F} = \frac{\mathrm{d}f}{\mathrm{d}\mathbf{x}'} = \begin{bmatrix} \frac{\mathrm{d}f_1}{\mathrm{d}x'_1} & \frac{\mathrm{d}f_1}{\mathrm{d}x'_2} \\ \frac{\mathrm{d}f_2}{\mathrm{d}x'_1} & \frac{\mathrm{d}f_2}{\mathrm{d}x'_2} \end{bmatrix} = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix}$$

This is exactly as we expect given the work that we've done so far. Things can get quite complicated though, especially when you put in magnitudes of vectors or rotations or other nonlinear functions. Luckily for me, my work is simple enough that we can use good ol' high school physics as a base. Now, we also have one more process to deal with, that of measurement predictions. Remember how the whole point of these filters is to minimize the residual, but we have to compare our measurements with our predictions using a measurement prediction matrix, **H**? Well with extended filters, we don't have a matrix, but a vector of functions, such that our residual, **y**, becomes:

$$\mathbf{y} = \mathbf{z} - h(\mathbf{x}_p) = \mathbf{z} - \begin{bmatrix} h_1(\mathbf{x}_p) \\ \vdots \\ h_n(\mathbf{x}_p) \end{bmatrix}$$

So like before, we use our nonlinear functions directly to calculate the residuals, but how do we make sure that we're comparing measurements with predictions correctly in order to improve our estimates? For that of course we still need our matrix, **H**, which we again find by using the Jacobian of the function vector *h*:

$$\mathbf{H} = \frac{\mathrm{d}h}{\mathrm{d}\mathbf{x}_p} = \begin{bmatrix} \frac{\mathrm{d}h_1}{\mathrm{d}x_{p1}} & \cdots & \frac{\mathrm{d}h_1}{\mathrm{d}h_{pn}} \\ \vdots & \ddots & \vdots \\ \frac{\mathrm{d}h_n}{\mathrm{d}x_{p1}} & \cdots & \frac{\mathrm{d}h_n}{\mathrm{d}h_{pn}} \end{bmatrix}$$

This is exactly the same as we did for the prediction process above. To demonstrate this; however, we'll use an actual nonlinear measurement prediction. Say we want to estimate two components of velocity, an x-component and a y-component; but, we only have access to measurements for speed, i.e. the magnitude of our velocity vector. How do we compare the two? Simple: we take our two predictions, and find the magnitude of that velocity vector. This means that our residual will look like:

$$\mathbf{y} = v_z - h(\mathbf{x}_p) = v_z - (v_{p_1}^2 + v_{p_2}^2)^{\frac{1}{2}}$$

Then to finally get better estimates by fusing everything, we find our measurement prediction matrix by finding the Jacobian of the function vector, *h*, and find ourselves with:

$$\mathbf{H} = \begin{bmatrix} \frac{v_{p_1}}{(v_{p_1}^2 + v_{p_2}^2)^{\frac{1}{2}}} & \frac{v_{p_2}}{(v_{p_1}^2 + v_{p_2}^2)^{\frac{1}{2}}} \end{bmatrix}$$

And that's it! That's everything you need to know about extended Kalman filters, because the rest of the process is exactly the same as with regular Kalman filters.

Implementation Considerations

Theoretically it's possible to control how often you receive data from smartphone sensors, e.g. ten times a second; however, in practice that's rarely true. Furthermore, different sensors have different upper limits to how fast they can send readings to the operating system of your mobile phone. For example, while some smartphone accelerometers can send accelerometer readings to the operating system one hundred times a second, GPS is strictly limited to updating data no more than once per second. Also, because different sensors have their own clocks, with slight errors compared to one another or the system clock, it can be hard to determine exactly *when* a measurement is sent to the operating system.

Reading through the literature, most filtering for telematics seems to assume that measurements all come in at the same time, so that every time you update your state estimates, you don't have to think about the above complications. However, since this isn't the case in real life, we investigated a few possible ways to address things:

Fixed Rate Filtering: filtering at regular time intervals using the most recent measurements,
Loose Filtering: accumulating high frequency measurements while waiting for your slowest measurement to arrive,
Sequential Filtering: updating your state estimate every time a measurement arrives.

None of these are really discussed in the literature, though fixed rate (with simultaneous measurements) seems to be the approach that's typically assumed. I have come across loose and sequential filtering in the wild via open-source implementations that are public on GitHub, but I don't really consider sequential filtering to be sensor fusion, as it really just implements a separate filter for each measurement, meaning it's just a least squares estimator.

Before we discuss each in kind, please note that there are no hard rules on any of this, feel free to mix and match based on your requirements. Personally, loose filtering at a fixed rate makes a lot of sense, but when using slow GPS measurements with strict data transfer constraints, loose filtering with a GPS measurement triggering a state update, is the way to go.

Fixed Rate Filtering

Fixed rate filtering is probably the default way that people think about using sensor data. That is, once every so often, at a fixed interval, data from multiple sensors arrives all at the same time, and can more or less be processed instantly (or near enough). Complications arise, however, when you realize that this is not quite so. For most smartphones, it's not about polling sensors for their current reading, it's about registering events with the operating system, to be fired at a specified rate, or after a certain amount of time has passed. It gets complicated, because these things aren't exact, and when you have more than one sensor,

they generally do not align in time. Moreover, depending on the system you're working with (phone, OS, sensors, etc.), you might not have any control over sensor event / interrupt rates, so you have to work with what you get.

How then, can we simulate fixed rate filtering in real-life?

Simple: maintain a series of relatively up-to-date readings that can be used at a fixed rate. That is, every time a sensor event occurs, you update whatever variable you're using to keep track of that sensor value. If you get events from the sensors fast enough, and filter at a rate slightly slower than that, that's almost as good as if all of your sensor events were to arrive at the same time.

Pseudocode for this approach could look something like this:

```
New Sensor Event:
update time between events using the system clock

    Accelerometer:
        low-pass and gravity filter accelerometer measurements
        discard old acceleration measurement
        update acceleration measurement

    Gyroscope:
        bias filter angular velocity measurement
        discard old angular velocity measurement
        update angular velocity measurement

    GPS:
        discard old speed measurement
        update speed measurement

    Timer:
        new state estimate = kalman_filter (old state, measurements)
```

For the most part, this type of approach is what I've used, but when using slow sensors like GPS, errors can accumulate in between measurements (assuming you're filtering faster than GPS readings come in) such that it might be worthwhile looking into other ways to go about things.

Loose Filtering

Loose filtering is something I stumbled upon when auditing open-source projects, and it might help fit filtering implementations into a stricter set of constraints. The gist of it is that if you are filtering at a slow fixed rate, or at the rate of your slowest sensor (e.g. GPS), you might lose critical information if you just discard measurements that come in from faster sensors. So, in between filter events you accumulate and average values from your faster sensors, using the result when you do eventually get around to filtering. This also allows you to perform some real-time filtering on the data that otherwise would not be possible at a lower rate.

For example, if you're measuring speed 25 times a second, but distance only once a second and then filtering only once a second, you would accumulate the 25 speed measurements, averaging them out over the course of a second, and then say, "that's the total speed measurement for the previous second," that you can then hand off to your filter.

This is useful if you need real sensor data for post-processing, but can't afford to transmit sensor data 25 times a second. You can still get useful information from your sensors at a slow rate, and still get decent state estimates from your filters, but you won't use up as much space or need to transfer as much data.

Pseudocode for this approach could look like:

```
New Sensor Event:
update time between events using the system clock

    Accelerometer:
      low-pass and gravity filter accelerometer measurements
      accumulate acceleration measurements

    Gyroscope:
      bias filter angular velocity measurement
      accumulate angular velocity measurements

    GPS:
      update speed measurement
      new state estimate = kalman_filter (old state, measurements)
      reset accumulators
```

This is a great approach and you can get good estimates while operating with other data / storage constraints, though it will ultimately result in lower resolution estimates. Also, if your filter rate is too slow, your averages can come to something completely wacky, like zero, when in reality your state has done something interesting like turn around.

Sequential Filtering

Don't use sequential filtering. I also came across this while auditing open-source projects, primarily in the autonomous vehicle space. Basically, you don't do sensor fusion: you just use a Kalman filter as a least squares filter to kinda smooth out your data as it comes in. That is, for every discrete sensor event, you build a separate filter that only takes that event as input, estimating the same thing. I don't get it, but I see it all over the place.

The pseudocode is very boring:

```
New Sensor Event:
update time between events using the system clock

    Accelerometer:
      low-pass and gravity filter accelerometer measurements
      update acceleration measurement
      new state estimate = kalman_filter_ACCEL (old state, acceleration measurement)

    Gyroscope:
      bias filter angular velocity measurement
      update angular velocity measurement
      new state estimate = kalman_filter_GYRO (old state, gyroscope measurement)

    GPS:
      update speed measurement
      new state estimate = kalman_filter_GPS (old state, speed measurement)
```

Appendix A: Example 1D & 2D Kalman Filter Code

Simple example function that compares the results of estimating speed using a Kalman filter with noisy speed measurements, versus estimating speed using both noisy speed and acceleration measurements.

```
function fig = test_kalman_filters_1D_2D

%   First, a one-dimensional filter:
%
%   Create Noisy Data:
    t = linspace(0, 10, 1000);
    dt = t(2) - t(1);

    za = 2 * exp(-(t - 5).^2 / 2 * 4) / sqrt(2*pi/4);
    za = conv(za, ones(1, length(t)/2), 'same') * dt;
    za = [za, -za];
    zv = cumtrapz(za) * dt;

    t = linspace(0, t(end) * 2, length(t) * 2);

    za = za + randn(size(za)) * 1;
    zv = zv + randn(size(zv)) * 2;

%   Initialize Filter:
    xe = zeros(1, 1);
    Pe = zeros(1, 1);

    F = 1;
    Qa = dt;
    Q = F * Qa * F.';

    H = 1;
    R = 2^2;

    x = zeros(1, length(t));
    z = zv;

%   Filter Data:
    for i = 1:length(t)

        [xe, Pe] = kalman_filter(xe, Pe, z(:,i));

        x(:,i) = xe;
    end

    fig = figure;
    subplot(1, 2, 1);
    plot(t, zv); grid on; hold on;
    plot(t, x, 'LineWidth', 1.5);
    title('One-Dimensional Filter');
    axis([0 20 -5 15]);

%   Second, a two-dimensional filter:
%
%   Initialize Filter:
    xe = zeros(2, 1);
    Pe = zeros(2, 2);

    F = [1, dt; 0, 1];
    Qa = [0, 0; 0, 1/dt];
    Q = F * Qa * F.';

    H = eye(2);
    R = [2^2, 0; 0, 1];

    x = zeros(2, length(t));
    z = [zv; za];
```

```
%   Filter Data:
    for i = 1:length(t)

        [xe, Pe] = kalman_filter(xe, Pe, z(:,i));

        x(:,i) = xe;
    end

    figure(fig);
    subplot(1, 2, 2);
    plot(t, zv); grid on; hold on;
    plot(t, x(1,:), 'LineWidth', 1.5);
    title('Two-Dimensional Filter');
    axis([0 20 -5 15]);
    legend('measurement', 'estimate', 'Location', 'South');

    ax = axes(fig, 'visible', 'off');
    ax.XLabel.Visible = 'on';
    ax.YLabel.Visible = 'on';
    ylabel(ax, 'speed (m/s)');
    xlabel(ax, 'time (s)');

        % KALMAN_FILTER  Standard single iteration of a Kalman filter.
        %
        %    [xe, Pe] = KALMAN_FILTER(xe, Pe, z) takes in previous state
        %    estimates, xe, their covariances, Pe, and current measurements
        %    to return updated current estimates and covariances.
        %
        %    Other parameters F, Q, H, and R are assumed to be global
        %    variables for this example implementation.
        function [xe, Pe] = kalman_filter(xe, Pe, z)

        %   1) Predict:
            xp = F * xe;
            Pp = F * Pe * F.' + Q;

        %   2) Weigh:
            K = Pp * H.' * ((H * Pp * H.' + R) \ eye(size(R)));

        %   3) Estimate:
            xe = xp + K * (z - H * xp);
            Pe = (eye(size(Pp)) - K * H) * Pp * (eye(size(Pp)) - K * H).' + K * R * K.';
        end
end
```