

The Extended Kalman Filter: An Interactive Tutorial for Non-Experts

In working with autopilot systems like [OpenPilot](#) and [Pixhawk](#) I have frequently come across references to something called an *Extended Kalman Filter* (EKF). Googling this term led me to several different web pages and reference papers, most of which I found too difficult to follow. ^[1] So I decided to create my own tutorial for teaching and learning about the EKF from first principles. This tutorial assumes only high-school-level math and introduces concepts from more advanced areas like linear algebra as needed, rather than assuming you already know them. Starting with some simple examples and the standard (linear) Kalman filter, we work toward an understanding of actual EKF implementations at end of the tutorial.

Part 1: A Simple Example

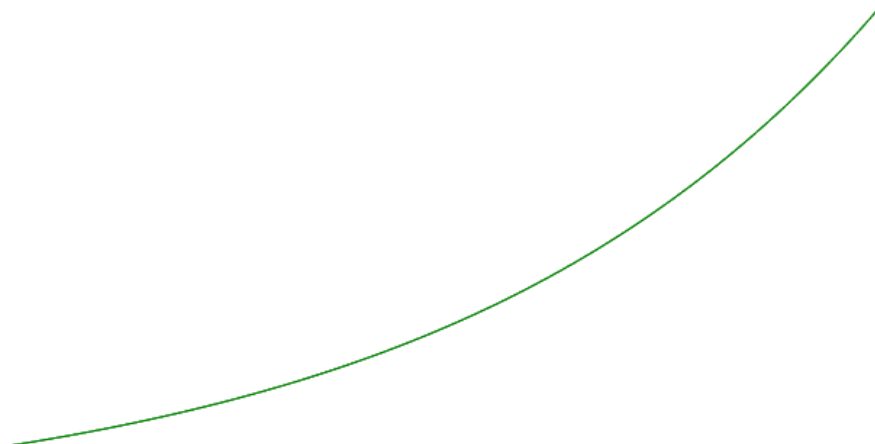
Imagine a airplane coming in for a landing. Though there are many things we might worry about, like airspeed, fuel, etc., the most obvious thing to focus on is the plane's altitude (height above sea level). As a very simple approximation, we can think of the current altitude as a fraction of the previous altitude. For example, if the plane loses 2% of its altitude each time we observe it, then its altitude at the current time is 98% of its altitude at the previous time:

$$\text{altitude}_{\text{current_time}} = 0.98 * \text{altitude}_{\text{previous_time}}$$

Engineers use the term *recursive* to refer to a formula like this where a quantity is defined in terms of its previous value: to compute the current value, we must “recur” back to the previous. Eventually we recur back to some initial “base case”, like a known starting altitude.



Try moving around the slider above to see how the plane's altitude changes for different percentages.



Next: [Dealing with Noise](#)

^[1] Two notable exceptions are [Kalman Filtering for Dummies](#) and the [the Wikipedia page](#), from which I have borrowed here.

This tutorial has also benefited from helpful comments by members of the [OpenPilot](#) and [DIY Drones](#) communities (esp. Tim Wilkins, who corrected a number of inaccuracies in my explanations).

The Extended Kalman Filter: An Interactive Tutorial for Non-Experts

Part 2: Dealing with Noise

Of course, real-world measurements like altitude are obtained from a sensor like a GPS or barometer. Such sensors offer varying degrees of accuracy. ^[2] If the sensor is off by a constant amount, we can simply add or subtract that amount to determine our altitude. Typically, though, sensor accuracy varies unpredictably from moment to moment, making the observed sensor reading a “noisy” version of the true altitude:

$$\text{observed_altitude}_{\text{current_time}} = \text{altitude}_{\text{current_time}} + \text{noise}_{\text{current_time}}$$



Try moving around the slider above to see the effect of noise on the observed altitude. The noise is represented as percentage of the range of observable altitudes.

Previous: [A Simple Example](#) Next: [Putting it Together](#)

[2] For example, [Garmin](#) publishes the accuracy of its barometric altimeter readout as “10 feet with proper calibration” So, for example, if the altimeter reads 1000 feet, our actual altitude could be anywhere between 990 and 1010 feet.

The Extended Kalman Filter: An Interactive Tutorial for Non-Experts

Part 3: Putting it Together

So now we have two equations describing the state of our airplane:

$$\text{altitude}_{\text{current_time}} = 0.98 * \text{altitude}_{\text{previous_time}}$$

$$\text{observed_altitude}_{\text{current_time}} = \text{altitude}_{\text{current_time}} + \text{noise}_{\text{current_time}}$$

These equations are pretty easy to understand, but they aren't general enough to deal with systems other than our airplane-altitude example. To make the equations more general, engineers adopt the familiar mathematical convention of using names like x , y , and z for variables and a and b for constants, and the subscript k to represent time.^[3] So our equations become:

$$x_k = ax_{k-1}$$

$$z_k = x_k + v_k$$

where x is the current state of our system, x_{k-1} is its previous state, a is some constant (0.98 in our example), z_k is our current observation of the system, and v_k is the current noise measurement. One reason the Kalman filter is so popular is that it allows us to get a very good estimation of the actual current state x_k given the observation z_k , the constant a , and the overall amount of measurement noise v .

To complete the picture, we should also consider that that actual altitude of the airplane may not describe a perfectly smooth path. As anyone who has ever flown can tell you, airplanes typically experience a certain amount of turbulence as they descend for a landing. This turbulence is by definition noisy, and so can be treated as another noise signal:

$$\text{altitude}_{\text{current_time}} = 0.98 * \text{altitude}_{\text{previous_time}} + \text{turbulence}_{\text{current_time}}$$

More generally:

$$x_k = ax_{k-1} + w_k$$

where w_k is called the *process noise*, because, like turbulence, it is an inherent part of the process, and not an artifact of observation or measurement. We will ignore process noise for a while in order to focus on other topics, but we'll return to it in the section on Sensor Fusion.^[4]

Previous: [Dealing with Noise](#) Next: [State Estimation](#)

[3] Why not t for time? Probably because time is being treated as a sequence of discrete steps, for which an index variable like k is conventional.

[4] I have also ignored the control-signal component of the state equation, because it is tangential to most of the Kalman Filter equations and can be easily added when needed.

The Extended Kalman Filter: An Interactive Tutorial for Non-Experts

Part 4: State Estimation

Here again (ignoring process noise) are our two equations describing the state of a system we are observing:

$$x_k = ax_{k-1} + w_k$$

$$z_k = x_k + v_k$$

Since our goal is to obtain the states x from the observations z , we could rewrite the second equation as:

$$x_k = z_k - v_k$$

The problem of course is that we don't know the current noise v_k : it is by definition unpredictable. Fortunately, [Kalman](#) had the insight that we can *estimate* the state by taking into account both the current observation and the previous estimated state. Engineers use a little caret or “hat” ^ over a variable to show that it is estimated: so \hat{x}_k is the estimate of the current state. Then we can express the estimate as a tradeoff between the previous estimate and the current observation:

$$\hat{x}_k = \hat{x}_{k-1} + g_k(z_k - \hat{x}_{k-1})$$

where g is a “gain” term expressing the tradeoff.^[5] I've highlighted this equation in red because it is one that we will use directly in implementing our Kalman filter.

Now, this all looks rather complicated, but think of what happens for two extreme values of the gain g_k . For $g_k = 0$, we get

$$\hat{x}_k = \hat{x}_{k-1} + 0(z_k - \hat{x}_{k-1}) = \hat{x}_{k-1}$$

In other words, when the gain is zero, observation has no effect, and we get the original equation relating the current state to the previous. For $g_k = 1$ we get

$$\hat{x}_k = \hat{x}_{k-1} + 1(z_k - \hat{x}_{k-1}) = \hat{x}_{k-1} + z_k - \hat{x}_{k-1} = z_k$$

In other words, when the gain is one, the previous state doesn't matter, and we get the current state estimation entirely from the current observation.

Of course, the actual gain value will likely fall somewhere between these two extremes. Try moving the slider below to see the effect of gain on current state estimation:

$$x_{k-1} = 110 \quad z_k = 105 \quad g_k = 0.97 \quad \hat{x}_k = 105.15$$


Previous: [Putting it Together](#)

Next: [Computing the Gain](#)

^[5] The variable k is usually used for gain, because it is known as the *Kalman gain*. With due respect to Rudolf Kalman, I find it confusing to use the same letter for a variable and a subscript, so I have opted for the letter g instead.

The Extended Kalman Filter: An Interactive Tutorial for Non-Experts

Part 5: Computing the Gain

So now we have a formula we can actually use for computing the current state estimate \hat{x}_k based on the previous estimate \hat{x}_{k-1} , the current observation z_k , and the current gain g_k :

$$\hat{x}_k = \hat{x}_{k-1} + g_k(z_k - \hat{x}_{k-1})$$

So how do we compute the gain? The answer is: *indirectly, from the noise*. Recall that each observation is associated with a particular noise value:

$$z_k = x_k + v_k$$

We don't know the individual noise value for an observation, but we typically do know the average noise: for example, the published accuracy of a sensor tells us approximately how noisy its output is. Call this value r ; there is no subscript on it because it does not depend on time, but is instead a property of the sensor. Then we can compute the current gain g_k in terms of r :

$$g_k = p_{k-1} / (p_{k-1} + r)$$

where p_k is a *prediction error* that is computed recursively:^[6]

$$p_k = (1 - g_k)p_{k-1}$$

As with the state-estimation formula, let's think about what these two formulas mean before we continue.

Let's say that the error p_{k-1} on our previous prediction was zero. Then our current gain g_k will be $0/(0 + r) = 0$, and our next state estimate will be no different from our current state estimate. Which makes sense, because we shouldn't be adjusting our state estimate if our prediction was accurate. At the other extreme, say the prediction error is one. Then the gain will be $1/(1 + r)$. If r is zero — i.e., if there is very little noise in our system — then the gain will be one, and our new state estimate \hat{x}_k will be strongly influenced by our observation z_k . But as r grows large, the gain can become arbitrarily small. In other words, when the system is noisy enough, a bad prediction will have to be ignored. Noise overcomes our ability to correct bad predictions.

What about the third formula, calculating the prediction error p_k recursively from its previous value p_{k-1} and the current gain g_k ? Again, it helps to think of what happens for extreme values of the gain: when $g_k = 0$, we have $p_k = p_{k-1}$. So, just as with the state estimation, a zero gain means no update to the prediction error. When on the other hand $g_k = 1$, we have $p_k = 0$. Hence the maximum gain corresponds to zero prediction errors, with the current observation alone used to update the current state.

Previous: [State Estimation](#) **Next:** [Prediction and Update](#)

[6] Technically r is really the *variance* of the noise signal; i.e., the spread, or squared average distance of individual noise values from their mean. The Kalman Filter will work equally well if this noise variance is allowed to change over time, but in most applications it can be assumed constant. In a likewise manner, p_k is technically the *covariance of the estimation process* at step k ; it is the average of the squared error of our predictions. Indeed, as Tim Wilkin has pointed out to me, *the state is a stochastic [random-like] variable/vector (an instantaneous value of a stochastic process) and it doesn't have a "true" value at all!* The estimate is merely the most likely value for the process model describing the state.

The Extended Kalman Filter: An Interactive Tutorial for Non-Experts

Part 6: Prediction and Update

We're almost ready to run our Kalman Filter and see some results. First, though, you may be wondering what happened to the constant a in our original state equation:

$$x_k = ax_{k-1}$$

which seems to have vanished in our equation for the state estimate:

$$\hat{x}_k = \hat{x}_{k-1} + g_k(z_k - \hat{x}_{k-1})$$

The answer is, we need *both* of these equations to estimate the state. Indeed, both equations represent an estimate of the state, based on different kinds of information. Our original equation represents a *prediction* about what the state *should* be, and our second equation represents an *update* to this prediction, based on an observation. [7] So we rewrite our original equation with a little hat on the x to indicate an estimate:

$$\hat{x}_k = a\hat{x}_{k-1}$$

Finally, we use the constant a in a prediction of the error as well: [8]

$$p_k = ap_{k-1}a$$

Together these two formulas in red represent the *prediction phase* of our Kalman Filter. The idea is that the cycle predict / update, predict / update, ... is repeated for as many time steps as we like.

Previous: [Computing the Gain](#) **Next:** [Running the Filter](#)

[7] Technically, the first estimate is called a *prior*, and the second a *posterior*, and most treatments introduce some additional superscript or subscript to show the distinction. Because I am trying to keep things simple (and easy to code up in your favorite programming language!), I avoid complicating the notation any further.

[8] As Zichao Zhang has kindly pointed out to me, we multiply twice by a because the prediction error p_k is itself a squared error; hence, it is scaled by the square of the coefficient associated with the state value x_k . The reason for representing the error prediction as $ap_{k-1}a$ instead of a^2p_{k-1} will become clear in [Part 12](#).

The Extended Kalman Filter: An Interactive Tutorial for Non-Experts

Part 7: Running the Filter

So now we have everything we need to run our Kalman Filter:

Predict:

$$\hat{x}_k = a\hat{x}_{k-1}$$

$$p_k = ap_{k-1}a$$

Update:

$$g_k = p_k / (p_k + r)$$

$$\hat{x}_k = \hat{x}_{k-1} + g_k(z_k - \hat{x}_{k-1})$$

$$p_k = (1 - g_k)p_k$$

To try out our filter, we'll need:

- A sequence of observations z_k
- An initial value (base case) \hat{x}_0 for the state estimates. This can just be our first observation z_0 .
- An initial value p_0 for the prediction error. It can't be 0, otherwise p_k would stay 0 forever by multiplication. So we arbitrarily set it to 1.

For our observations, rather than trying to observe an actual system (like a plane coming in for a landing), we'll fake up some observations based on adding random noise [\[10\]](#) v_k in the interval [-200,+200] to the idealized values $x_k = 0.75x_{k-1}$, starting with $x_0 = 1000$:

k	0	1	2	3	4	5	6	7	8	9
x_k										
z_k										
p_k										
\hat{x}_k										
g_k										

$x_0 =$ $r =$ $a =$

Once you're ready to run the filter, hit the Run button to see how the Kalman filter produces a smooth version (green) of the noisy signal (red) that is often remarkably close to the original clean signal (blue). You can also try different values for x_0 , r , and a .

Previous: [Prediction and Update](#) Next: [A More Realistic Model](#)

[\[9\]](#) Indeed, if you look at the source code for this page, you will see that the JavaScript for the prediction and update is even simpler than the formulas:

```
// Predict
xhat = a * xhat;
p     = a * p * a;

// Update
g     = p / (p + r);
xhat = xhat + g * (z - xhat);
p     = (1 - g) * p;
```

I thank Marco Camurri for pointing out an inconsistency in my use of the equations, in an earlier version of this tutorial.

[\[10\]](#) I have added noise from a [uniform distribution](#) rather than the [Gaussian \(normal\)](#) distribution assumed by the Kalman Filter, but it doesn't make much difference from the perspective of this demo.

The Extended Kalman Filter: An Interactive Tutorial for Non-Experts

Part 8: A More Realistic Model

Recall the two equations describing our system:

$$x_k = ax_{k-1}$$
$$z_k = x_k + v_k$$

where x_k is the current state of our system, x_{k-1} is its previous state, a is some constant, z_k is our current observation of the system, and v_k is the current noise (inaccuracy) associated with the observation.

Although these two equations apply well to many kinds of systems, they are sometimes not the whole story. For one thing, we have not accounted for the time-varying *control* that the pilot exercises over the airplane, by (for example) moving the [control column](#) forward and back. To account for the control we introduce another subscripted variable u_k , representing the current value of the *control signal* that the pilot is sending to the airplane. Just as the previous state x_{k-1} was scaled by a constant amount a , this control signal can be scaled by a constant amount if we like; call it b . So our complete equation for the state becomes

$$x_k = ax_k + bu_k$$

with the new component highlighted in blue.

In general, any signal other than noise can be scaled by some constant, so our equation for the observation z_k can be rewritten thus: [\[11\]](#)

$$z_k = c x_k + v_k$$

Previous: [Running the Filter](#) Next: [Modifying the Estimates](#)

[11] For the sake of consistency with the original example I have chosen to use variables a , b , and c here for the constants, instead of the more common f , b , and h .

The Extended Kalman Filter: An Interactive Tutorial for Non-Experts

Part 9: Modifying the Estimates

Here again our more realistic / more general equations for the state and observation variables of our system:

$$x_k = ax_{k-1} + bu_k$$

$$z_k = cx_k + v_k$$

As we might expect, the introduction of these new components into our model requires a corresponding modification to the prediction and update equations:

Predict:

$$\hat{x}_k = a\hat{x}_{k-1} + bu_k$$

$$p_k = ap_{k-1}a$$

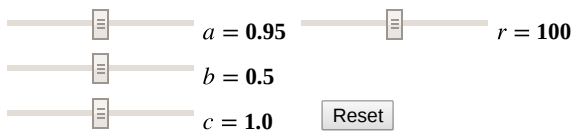
Update:

$$g_k = p_k c / (c p_k c + r)$$

$$\hat{x}_k = \hat{x}_k + g_k(z_k - c \hat{x}_k)$$

$$p_k = (1 - g_k c) p_k$$

Here is an extension of our airplane demo, showing a longer duration of time and adding in a control signal representing the pilot steadily pulling back on the control column to raise the altitude of the plane. Try moving around the sliders to adjust the values of the different constants. As in the previous demo, the original signal is shown blue, the observed signal in red, and the Kalman-filtered signal in green.



Previous: [A More Realistic Model](#) Next: [Adding velocity to the system](#)

The Extended Kalman Filter: An Interactive Tutorial for Non-Experts

Part 10: Adding Velocity to the System

Recall our original equation for the altitude of an airplane:

$$altitude_{current} = 0.98 * altitude_{previous}$$

with the more general form:

$$x_k = ax_{k-1}$$

Thinking back to the math and physics you may have learned in high school, this kind of formula seems a bit odd. Altitude, after all, is a kind of distance (above sea level, or above ground level), for which we learned the formula

$$distance = velocity * time$$

Can we reconcile these two different ways of thinking about distance? The answer is yes, but it will require us to take two steps.

First, we need to introduce the concepts *current time* and *previous time* into our high-school formula, and think about distance in discrete time steps rather than overall distance:

$$distance_{current} = distance_{previous} + velocity_{previous} * (time_{current} - time_{previous})$$

In other words, where we are now is where we were a moment ago, plus the distance we just traveled. If we perform this computation at regular time intervals, or timesteps (one second, 100 nanoseconds, six months, etc.), then we can simplify this to:

$$distance_{current} = distance_{previous} + velocity_{previous} * timestep$$

This equation moves us closer to our general form

$$x_k = ax_{k-1}$$

but we still seem to have two very different systems: one involves a simple product, and the other involves a product and a sum. Coming up with a general equation for both brings us the second step: *linear algebra*.

Previous: [Modifying the Estimates](#) **Next:** [Linear Algebra](#)

The Extended Kalman Filter: An Interactive Tutorial for Non-Experts

Part 11: Linear Algebra

So we have an equation expressing distance in terms of velocity and time:

$$distance_{current} = distance_{previous} + velocity_{previous} * timestep$$

which we are trying to reconcile with a more general equation

$$x_k = ax_{k-1}$$

Fortunately for us, mathematicians long ago devised “one weird trick” for representing both kinds of equations in the same way. The trick is to think of a situation (like the state of a system) not as a single number, but rather as a list of numbers called a *vector*, which is like a column in an Excel spreadsheet. The size of the vector (number of elements) corresponds to the number of things we want to encode about the state. For distance and velocity, we have two items in our vector:

$$x_k \equiv \begin{bmatrix} distance_k \\ velocity_k \end{bmatrix}$$

Here I've used the “triple equal \equiv ” to indicate that this is a definition: *the current state is defined as a vector containing the current distance and current velocity*.

So how does this help us? Well, another thing we get from linear algebra is a *matrix*. If a vector is like a column of values in a spreadsheet, then a matrix is like a whole spreadsheet, or table of values. When we multiply a matrix by a vector we get another vector of the same size:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} ax + by \\ cx + dy \end{bmatrix}$$

For example:

$$\begin{bmatrix} 8 & 3 \\ 4 & 7 \end{bmatrix} \begin{bmatrix} 2 \\ 5 \end{bmatrix} = \begin{bmatrix} 31 \\ 43 \end{bmatrix}$$

The vectors and matrices can be of any size, as long as they match:

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} ax + by + cz \\ dx + ey + fz \\ gx + hy + iz \end{bmatrix}$$

We can also multiply two matrices together to get another matrix:

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} r & s & t \\ u & v & w \\ x & y & z \end{bmatrix} = \begin{bmatrix} ar + bu + cx & as + bv + cy & at + bw + cz \\ dr + eu + fx & ds + ev + fy & dt + ew + fz \\ gr + hu + ix & gs + hv + iy & gt + hw + iz \end{bmatrix}$$

Adding two matrices is simpler; we just add each pair of elements:

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} + \begin{bmatrix} r & s & t \\ u & v & w \\ x & y & z \end{bmatrix} = \begin{bmatrix} a+r & b+s & c+t \\ d+u & e+v & f+w \\ g+x & h+y & i+z \end{bmatrix}$$

Returning to the task at hand, we define a matrix

$$A = \begin{bmatrix} 1 & timestep \\ 0 & 1 \end{bmatrix}$$

following the convention of using an uppercase letter to represent a matrix. Then our general equation is pretty much the same:

$$x_k = Ax_{k-1}$$

which works out as we want:

$$\begin{aligned} \begin{bmatrix} distance_k \\ velocity_k \end{bmatrix} &= \begin{bmatrix} 1 & timestep \\ 0 & 1 \end{bmatrix} \begin{bmatrix} distance_{k-1} \\ velocity_{k-1} \end{bmatrix} \\ &= \begin{bmatrix} 1 * distance_{k-1} + timestep * velocity_{k-1} \\ 0 * distance_{k-1} + 1 * velocity_{k-1} \end{bmatrix} \\ &= \begin{bmatrix} distance_{k-1} + timestep * velocity_{k-1} \\ velocity_{k-1} \end{bmatrix} \end{aligned}$$

In other words, the current distance is the previous distance plus the previous velocity times the timestep, and the current velocity is the same as the previous velocity. If we want to model a system in which the velocity changes over time, we can easily modify our vector and matrix to include acceleration: [\[12\]](#)

$$\begin{bmatrix} distance_k \\ velocity_k \\ acceleration_k \end{bmatrix} = \begin{bmatrix} 1 & timestep & 0 \\ 0 & 1 & timestep \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} distance_{k-1} \\ velocity_{k-1} \\ acceleration_{k-1} \end{bmatrix}$$

Previous: [Adding Velocity to the System](#) **Next:** [Prediction and Update Revisited](#)

[\[12\]](#) Several people have emailed me to ask why the upper-right value in this matrix is 0, rather than the $0.5t^2$ value that we learned in high-school physics. The answer is that this system does not relate distance directly to acceleration; instead, it uses acceleration to update velocity, and velocity to update distance. If you're still not convinced, you can run this little Python [program](#), which produces the expected parabolic distance [curve](#) under constant acceleration.

The Extended Kalman Filter: An Interactive Tutorial for Non-Experts

Part 12: Prediction and Update Revisited

Here again is our modified formula for system state:

$$x_k = Ax_{k-1}$$

where x is a vector and A is a matrix. As you may recall, the original form of this equation was

$$x_k = ax_{k-1} + bu_k$$

where u_k is a control signal, and b is a coefficient to scale it. We also had the equation

$$z_k = cx_k + v_k$$

where z_k is a measurement (observation, sensor) signal, and v_k is some noise added to that signal resulting from the sensor's inaccuracy. So how do we modify these original forms to work with our new vector/matrix approach? As you might suspect, linear algebra makes this quite simple: we write the coefficients b and c in capital letters, making them matrices rather than scalar (single) values:

$$x_k = Ax_{k-1} + Bu_k$$

$$z_k = Cx_k + v_k$$

Then all the variables (state, observation, noise, control) are considered vectors, and we have a set of matrix * vector multiplications. [\[13\]](#)

So what about our prediction and update equations? Recall that they are:

Predict:

$$\hat{x}_k = a\hat{x}_{k-1} + bu_k$$

$$p_k = ap_{k-1}a$$

Update:

$$g_k = p_k c / (c p_k c + r)$$

$$\hat{x}_k = \hat{x}_k + g_k (z_k - c\hat{x}_k)$$

$$p_k = (1 - g_k c) p_k$$

We would like to just capitalize all the constants a , b , c , and r so that they become matrices A , B , C , and R , and be done with it. But things are a little trickier for linear algebra! You may remember when I promised to explain why I didn't simplify $ap_{k-1}a$ to $a^2 p_{k-1}$. The answer is that linear algebra multiplication is not as straightforward as ordinary multiplication. To get the answer to come out right, we often have to perform the multiplication in a certain order; for example, AxB is not necessarily equal to BxA . We may also need to *transpose* the matrix, indicated by putting a little superscript T next to the matrix. Transposing a matrix is done by turning each row into a column and each column into a row. Here are some examples:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}^T = \begin{bmatrix} a & c \\ b & d \end{bmatrix}$$

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}^T = \begin{bmatrix} a & d & g \\ b & e & h \\ c & f & i \end{bmatrix}$$

With this knowledge in mind, we can rewrite our prediction equations as follows:

$$\hat{x}_k = A\hat{x}_{k-1} + Bu_k$$

$$P_k = AP_{k-1}A^T$$

Note that we have rewritten both A and P_k in capital letters, indicating that they are matrices. We already know why A is a matrix; we will defer for a moment understanding why P_k is also a matrix.

What about our update equations? The second update equation, for updating the state estimation \hat{x}_k , is straightforward:

$$\hat{x}_k = \hat{x}_k + g_k (z_k - C\hat{x}_k)$$

but the first equation involves a division:

$$g_k = p_k c / (c p_k c + r)$$

Since multiplication and division are so closely related, we run into a similar issue with matrix division as we did with multiplication. To see how we deal with this issue, let's first rewrite our second update equation using the familiar superscript $^{-1}$ to indicate division as inversion ($a^{-1} = 1/a$):

$$g_k = p_k c (c p_k c + r)^{-1}$$

Now we turn c , r , and g and the constant 1 into matrices, transposing C as needed – and deferring for a moment, once more, our understanding of why they need to be matrices:

$$G_k = P_k C^T (C P_k C^T + R)^{-1}$$

$$P_k = (I - G_k C) P_k$$

So how do we compute $(C P_k C^T + R)^{-1}$, the inverse of the matrix $(C P_k C + R)$? Just as with ordinary inversion, where $a * a^{-1} = 1$, we need matrix inversion to work in a way that multiplying a matrix by its own inverse produces an *identity* matrix, which when multiplied by any other matrix leaves that matrix unchanged:

$$AA^{-1} = I$$

where (for a 3x3 matrix)

$$I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Computing a matrix inverse isn't as simple as inverting each element of the matrix, which would yield the wrong answer in most cases. Although the details of matrix inversion are beyond the scope of this tutorial, there are excellent resources like [MathWorld](#) for learning about it. Even better, you usually don't have to code it yourself; it is built into languages like [Matlab](#), and is available via a package in [Python](#) and other popular languages.

Previous: [Linear Algebra](#) **Next:** [Sensor Fusion Intro](#)

[13] Indeed, we can think of our original scalar form of these equations as a special case of the linear-algebra form, with zeros in all but one position in each matrix.

The Extended Kalman Filter: An Interactive Tutorial for Non-Experts

Part 13: Sensor Fusion Intro

So now we have a complete set of equations for our Kalman Filter in linear algebra (vector, matrix) form:

Model:

$$x_k = Ax_{k-1} + Bu_k$$

$$z_k = Cx_k + v_k$$

Predict:

$$\hat{x}_k = A\hat{x}_{k-1} + Bu_k$$

$$P_k = AP_{k-1}A^T$$

Update:

$$G_k = P_k C^T (C P_k C^T + R)^{-1}$$

$$\hat{x}_k = \hat{x}_k + G_k (z_k - C\hat{x}_k)$$

$$P_k = (I - G_k C) P_k$$

This seems like an awful lot of work just to be able to add a few extra items to our state variable! In fact, using linear algebra supports an extremely valuable capability of the Kalman Filter, called *sensor fusion*.

Returning to our airplane example, we note that pilots have access to much more information (observations) than just altitude: they also have gauges showing the plane's airspeed, groundspeed, heading, latitude and longitude, outside temperature, etc. Imagine a plane with just three sensors, each of which corresponds to a given part of the state: a barometer for altitude, a compass for heading, and a [Pitot tube](#) for airspeed. Assume for the time being that these sensors are perfectly accurate (no noise). Then our observation equation

$$z_k = Cx_{k-1} + v_k$$

becomes

$$\begin{bmatrix} \text{barometer}_k \\ \text{compass}_k \\ \text{pitot}_k \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \text{altitude}_{k-1} \\ \text{heading}_{k-1} \\ \text{airspeed}_{k-1} \end{bmatrix}$$

Now imagine we have another sensor for altitude, say a GPS. Both the barometer and the GPS will be affected by altitude. So our equation becomes:

$$\begin{bmatrix} \text{barometer}_k \\ \text{compass}_k \\ \text{pitot}_k \\ \text{gps}_k \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} \text{altitude}_{k-1} \\ \text{heading}_{k-1} \\ \text{airspeed}_{k-1} \end{bmatrix}$$

Here we have our first example of a system without a simple one-to-one correspondence between sensors and state values. [14] Any such system affords us with the opportunity for *sensor fusion*; that is, the ability to combine readings from more than one sensor (barometer, GPS) to infer something about a component (altitude) of the state.

As when we seek a second doctor's opinion on a medical condition, our intuition tells us that it is better to have more than one source of information about something important. In the next section, we will see how the Kalman Filter uses sensor fusion to give us a better state estimate than we can get with a one sensor alone.

Previous: [Prediction and Update Revisited](#) **Next:** [Sensor Fusion Example](#)

[14] Our C matrix is somewhat unrealistic, in that the values would likely be something other than 1. The point is that a nonzero value in the matrix corresponds to a relationship between a sensor and a component of the state vector.

The Extended Kalman Filter: An Interactive Tutorial for Non-Experts

Part 14: Sensor Fusion Example

To get a feel for how sensor fusion works, let's restrict ourselves again to a system with just one state.^[15] To simplify things even further, we'll assume we have no knowledge of the state-transition model (A matrix) and so have to rely only on the sensor values. Perhaps we are measuring the temperature outside with two different thermometers. So we'll just set our state-transition matrix to 1:

$$\hat{x}_k = A\hat{x}_{k-1} = 1 * \hat{x}_{k-1} = \hat{x}_{k-1}$$

Lacking a state-transition model for our thermometer, we just assume that the current state is the same as the previous state.

For sensor fusion we will of course need more than one sensor value in our observation vector z_k , which for this example we can treat as the current readings of our two thermometers. We'll assume that both sensors contribute equally to our temperature estimation, so our C matrix is just a pair of 1's:

$$z_k = Cx_k + v_k = \begin{bmatrix} 1 \\ 1 \end{bmatrix} x_k + v_k$$

We now have two matrices (A , C) of the three (A , C , R) that we need for the prediction and update equations. So how do we obtain R ?

Recall that for our single-sensor example, we defined r as the *variance* of the observation noise signal v_k ; that is, how much it varies around its mean (average) value. For a system with more than two sensors, R is a matrix containing the *covariance* between each pair of sensors. The elements on the diagonal of this matrix will be the r value for each sensor, i.e., that sensor's variance with itself. Elements off the diagonal represent how much one sensor's noise varies with another's. For this example, and many real-world applications, we assume that such values are zero. Let's say that we've observed both our thermometers under climate-controlled conditions of steady temperature, and observed that their values fluctuate by an average of 0.8 degrees; i.e., the standard deviation of their readings is 0.8, making the variance $0.8 * 0.8 = 0.64$. This gives us the R matrix:

$$R = \begin{bmatrix} 0.64 & 0 \\ 0 & 0.64 \end{bmatrix}$$

Now we can also see why P_k and G_k must be matrices: as mentioned in a footnote earlier, P_k is the *covariance of the estimation process* at step k ; so, like the sensor covariance matrix R , P_k is also a matrix. And, since G_k is the gain associated with these matrices at each step, G_k must be likewise be a matrix, containing a gain value for each covariance value in these matrices. The sizes of these matrices P_k and G_k are of course determined by what they represent. In our present example, the size of P_k is 1×1 (i.e., a single value), because it represents the covariance of the single value estimate value \hat{x}_k with itself. And the gain G_k is a 1×2 (one row, two column) matrix, because it relates the single state estimate \hat{x}_k two the two sensor observations in z_k .

Putting it all together, we get the following equations for prediction and update in our airplane example (using covariance noise values between 0 and 200 feet, as before):

Predict:

$$\begin{aligned}\hat{x}_k &= A\hat{x}_{k-1} = 1 * \hat{x}_{k-1} = \hat{x}_{k-1} \\ P_k &= AP_{k-1}A^T = 1 * P_{k-1} * 1 = P_{k-1}\end{aligned}$$

Update:

$$\begin{aligned}G_k &= P_k C^T (C P_k C^T + R)^{-1} = P_k \begin{bmatrix} 1 & 1 \end{bmatrix} \left(\begin{bmatrix} 1 \\ 1 \end{bmatrix} P_k \begin{bmatrix} 1 & 1 \end{bmatrix} + \begin{bmatrix} 200 & 0 \\ 0 & 180 \end{bmatrix} \right)^{-1} \\ \hat{x}_k &= \hat{x}_k + G_k(z_k - C\hat{x}_k) = \hat{x}_k + G_k(z_k - \begin{bmatrix} 1 \\ 1 \end{bmatrix} \hat{x}_k) \\ P_k &= (I - G_k C)P_k = (1 - G_k \begin{bmatrix} 1 \\ 1 \end{bmatrix})P_k\end{aligned}$$

As it turns out, our impoverished state-transition model can get us into trouble if we don't reintroduce something we mentioned earlier: *process noise*. Recall that our complete equation for the state transition in a single-variable system was

$$x_k = ax_{k-1} + w_k$$

where w_k is the process noise at a given time. With our linear algebra knowledge we would now of course write this equation as

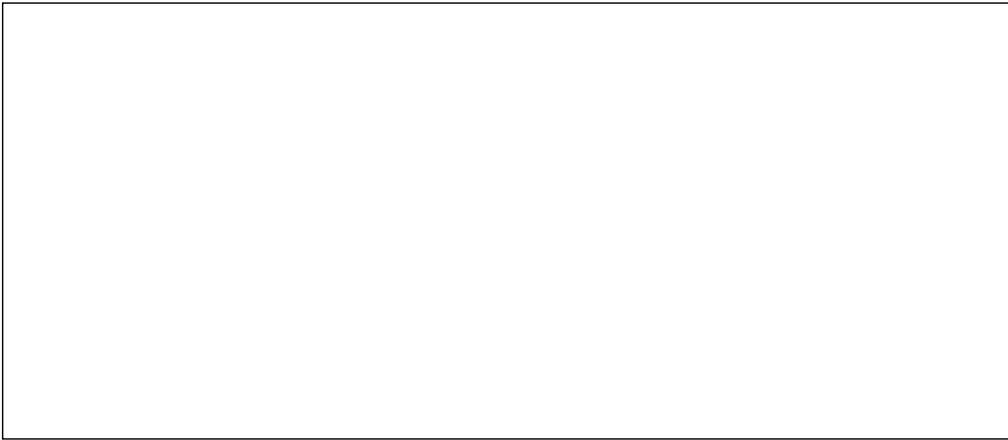
$$x_k = Ax_{k-1} + w_k$$

but the fact remains that we still have not accounted for the process noise in our prediction / update model. Doing this turns out to be pretty easy. Just as we used R to represent the covariance of the measurement noise v_k , we use Q to represent the covariance of the process noise w_k . Then we make a slight modification to our P_k prediction, simply adding in this covariance:

$$P_k = AP_{k-1}A^T + Q$$

The interesting thing is, even very small values for the nonzero elements of this Q matrix turn out to be very helpful in keeping our estimated state values on track.

So here is at last is a little sensor-fusion demo, allowing you to experiment with the values in R and Q , and also to change the amount of *bias* (constant inaccuracy; or mean value of the noise) in each of the two sensors. As you can see, when sensors are biased in different directions, sensor fusion can provide a closer approximation to the "true" state of the system than you can get with a single sensor alone.



Plot: ☐ Actual x_k ☐ Sensor 1 z_{1k} ☐ Sensor 2 z_{2k} ☐ Estimated \hat{x}_k



Sensor 1 Bias = -1 Sensor 2 Bias = 1



R1 = 0.64

R2 = 0.64

Q = .05

Previous: [Sensor Fusion Intro](#) Next: [Nonlinearity Intro](#)

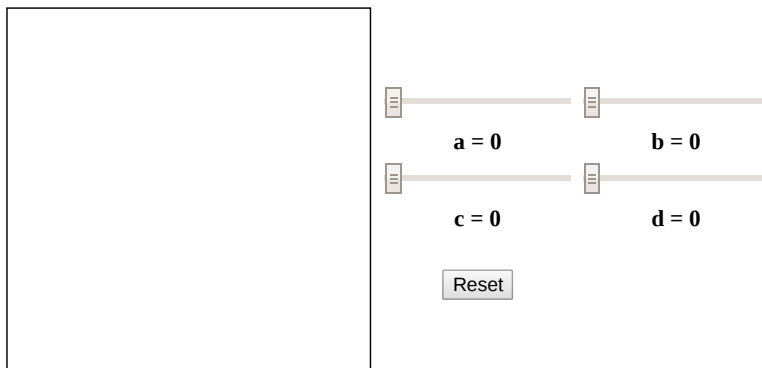
[15] I adapted this material from the example in Antonio Moran's excellent [slides](#) on Kalman filtering for sensor fusion. Matlab / Octave users may want to try out the [version](#) I've posted on Github, which includes a more general implementation of the Kalman filter.

Part 15: Nonlinearity

By now it should be obvious that linear algebra is pretty awesome, letting us express a sophisticated algorithm like the Kalman Filter in a very compact form. Linear algebra is however not the whole story. As its name suggests, linear algebra is limited to representing relationships that are linear, i.e., characterized by a straight line. To see this, consider again our simple example of matrix * vector multiplication:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} ax + by \\ cx + dy \end{bmatrix}$$

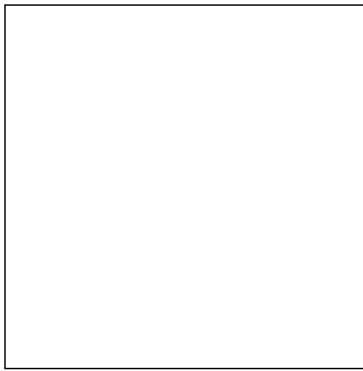
The demo below allows you to adjust the values of the constants a , b , c , and d to see how the resulting vector changes, for an arbitrary line segment containing points (x, y) . As you can see, all settings result in another straight line segment:



(x, y) $(ax + by, cx + dy)$

On the other hand, go outside and take a walk, and you'll appreciate that very little in nature is linear. The things you'll see with straight lines – buildings, roads, telephone poles, and the like – are mostly man-made, and most of everything else (trees, birds, clouds) is curved or [fractal](#). And because sensors and motors and other artifacts are made from physical materials, their behavior likewise tends toward nonlinearity, outside some limited range.

Pretty much all the functions you learned about in high-school math, like $f(x) = ax^2 + bx + c$, $f(x) = \sin(x)$, $\log(x)$, etc., are nonlinear, so there's a wide variety to choose from. The demo below allows you to pass the line segment from the previous demo through a few different nonlinear functions that I chose for their interesting appearance:



(x, y) $f(x, y)$

☐ $f\left(\begin{bmatrix} x \\ y \end{bmatrix}\right) = \begin{bmatrix} \sin(2\pi x + \pi y) \\ \cos(\pi x + \pi y) \end{bmatrix}$
☐ $f\left(\begin{bmatrix} x \\ y \end{bmatrix}\right) = \begin{bmatrix} (1 + e^{-4y})^{-1} \\ x \end{bmatrix}$


☐ $f\left(\begin{bmatrix} x \\ y \end{bmatrix}\right) = \begin{bmatrix} xy^2 \\ x + y^2 \end{bmatrix}$
☐ $f\left(\begin{bmatrix} x \\ y \end{bmatrix}\right) = \begin{bmatrix} y \\ \text{abs}(x + y) \end{bmatrix}$

Previous: [Sensor Fusion Example](#) **Next:** [Dealing with Nonlinearity](#)

Part 16: Dealing with Nonlinearity

Although nonlinearity introduces a whole new world of possible variations into any system, hope is not lost. As you may have noticed in the previous demo, the first three nonlinear examples have a useful property in common: they are all smoothly curved, as opposed to the final example, which has sharp turnaround. Mathematicians refer to such smoothly curving functions as [differentiable](#). As anyone who has studied calculus can attest, differentiable functions (smooth curves) can be modeled to an arbitrary degree of precision by a sequence of successively smaller line segments. Even without calculus, you can see this in the demo below, which allows you to approximate the function $f(x) = x^2$ by adjusting the size of the line segment Δx :




 $\Delta x = 0$

So how does this line-segment trick help handle nonlinear relationships in our Kalman Filter? Consider a very simple filter with the following linear equation for its sensor:

$$z_k = 3x_{k-1} + 2$$

This equation says that the sensor reading is always three times the corresponding state value, plus two: *no matter what the state value, the sensor reading is always three times that value plus two*. Now consider a nonlinear sensor equation:

$$z_k = \log_2(x_{k-1})$$

This equation says that the sensor reading is the logarithm (to the base 2) of the state value: a typical relationship, for example, in our sensation of [pitch](#) as a function of frequency. Even if you've never heard of a logarithm before, a quick look at the following table of approximate values shows that the relationship between the state x_k and the sensor reading z_k is not as straightforward as the previous one:

x_k	1.0	2.0	3.0	4.0	5.0	6.0	7.0	8.0	9.0	10	11	12	13	14	15	16	17	18	19	20
z_k	0.0	1.0	1.6	2.0	2.3	2.6	2.8	3.0	3.2	3.3	3.5	3.6	3.7	3.8	3.9	4.0	4.1	4.2	4.2	4.3

Here, you can see that there are no constants a and b such that $z_k = ax_{k-1} + b$. We can, however, use our line-segment insight from above to derive a different set of values a_k and b_k , one for each timestep, that approximates such a relationship. If you studied calculus, you may remember that we can compute the *first derivative* of many nonlinear functions (\log , \sin , \cos , etc.) directly. If you didn't study calculus, don't feel bad: the first derivative of a function is really just the best linear (line-segment) approximation to that function at each given point. A little Googling reveals that the first derivative of $\log_2(x)$ is approximately $1/0.693x$, which makes sense: as x increases, the value of $\log_2(x)$ goes up more and more gradually.

Previous: [Nonlinearity](#) **Next:** [A Nonlinear Kalman Filter](#)



Linear approximation $c = 0$

Previous: [Nonlinearity](#) Next: [Computing the Derivative](#)

[16] Why not call this function f ? We will see why in a couple more pages.

Part 18: Computing the Derivative

If you've made it this far, you are in a very good position to understand the Extended Kalman Filter. There are just two more things to consider:

1. How to compute the first derivative from an actual signal, without knowing its underlying function.
2. How to generalize our single-valued nonlinear state/observation model to the multi-valued systems we've been considering.

To answer the first question, we note that the first derivative of a function is defined as the limit of the difference between successive values of that function, divided by the timestep, as the timestep approaches zero:

$$f'(x) = \lim_{\Delta x \rightarrow 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}$$

If you don't understand that equation, don't worry: just think about subtracting successive differences of a signal y to approximate its first derivative:

$$\frac{(y_{k+1} - y_k)}{\text{timestep}}$$

Indeed, as the demo below shows, this *finite difference* formula is often a very good approximation to the first derivative. The demo allows you to choose among the same three functions as on the previous page (shown in the interval $[0,1]$), but this time you can select between the derivative and finite difference:



☐ $f(x) = e^{2x}$ ☒ $f(x) = 10 \log_{10}(x)$ ☐ $f(x) = 10e^{-x/10}$

☐ Derivative ☒ Finite difference

If one signal z_k (like sensor value z_k) is a function of another signal (like state x_k), we can divide successive differences of the first signal by successive differences of the second signal:

$$\frac{z_{k+1} - z_k}{x_{k+1} - x_k}$$

Previous: [A Nonlinear Kalman Filter](#) Next: [The Jacobian](#)

The Extended Kalman Filter: An Interactive Tutorial for Non-Experts

Part 19: The Jacobian

To answer our second question – how to generalize our single-valued nonlinear state/observation model to a multi-valued systems – it will be helpful to recall the equation for the sensor component of our linear model:

$$z_k = Cx_k$$

For a system with two state values and three sensors, we can rewrite this as:

$$z_k = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \\ c_{31} & c_{32} \end{bmatrix} \begin{bmatrix} x_{k1} \\ x_{k2} \end{bmatrix} = \begin{bmatrix} c_{11}x_{k1} + c_{12}x_{k2} \\ c_{21}x_{k1} + c_{22}x_{k2} \\ c_{31}x_{k1} + c_{32}x_{k2} \end{bmatrix}$$

Although you may not have seen notation like this before, it is pretty straightforward: the double numerical index on the elements of the C matrix indicates the row and column position of each element, but more importantly, the relationship being expressed. For example, c_{12} is the coefficient (multiplier) relating the current value z_{k1} of the first sensor to the second component x_{k2} of the current state.

For a nonlinear model, there will likewise be a matrix whose number of rows equals the number of sensors and number of columns equals the number of states; however, this matrix will contain *the current value of the first derivative of the sensor value with respect to that state value*. Mathematicians call such a derivative a [partial derivative](#), and the matrix of such derivatives they call the [Jacobian](#). Computing the Jacobian is beyond the scope of the present tutorial [\[17\]](#), but this Matlab-based [EKF tutorial](#) and this Matlab-based [implementation with GPS examples](#) show that it involves relatively little code.

If these concepts seems confusing, think about a survey in which a group of people is asked to rate a couple of different products on a scale (say, 1 to 5). The overall score given to each product will be the average of all the people's ratings on that product. To see how one person influenced the overall rating for a single product, we would look at that person's rating on that product. Each such person/product rating is like a partial derivative, and the table of such person/product ratings is like the Jacobian. Replace people with sensors and issues with states, and you understand the sensor model of the Extended Kalman Filter.

All that remains at this point is to generalize our nonlinear sensor/state model to the state-transition model. In other words, our linear model

$$x_k = Ax_{k-1} + w_k$$

becomes

$$x_k = f(x_{k-1}) + w_k$$

where A is replaced by the Jacobian of the state-transition function f . In fact, the convention is to use F_k for this Jacobian (since it corresponds to the function f and changes over time), and to use H_k for the Jacobian of the sensor function h . Incorporating the control signal u_k into the state-transition function, we got the “full Monty” for the Extended Kalman Filter that you are likely to encounter in the literature:

Model:

$$x_k = f(x_{k-1}, u_k) + w_k$$

Predict:

$$z_k = h(x_k) + v_k$$

$$\hat{x}_k = f(\hat{x}_{k-1}, u_k)$$

Update:

$$P_k = F_{k-1} P_{k-1} F_{k-1}^T + Q_{k-1}$$

$$G_k = P_k H_k^T (H_k P_k H_k^T + R)^{-1}$$

$$\hat{x}_k = \hat{x}_k + G_k (z_k - h(\hat{x}_k))$$

$$P_k = (I - G_k H_k) P_k$$

Previous: [Computing the Derivative](#) **Next:** [TinyEKF](#)

[\[17\]](#) In most EKF examples I've seen, the state transition function is simply the identity function $f(x) = x$. So its Jacobian is just the identity matrix described in [Section 12](#). Likewise for the observation function h and its Jacobian H .

The Extended Kalman Filter: An Interactive Tutorial for Non-Experts

Part 20: TinyEKF

If you've come this far, you're ready to start experimenting with an actual EKF implementation.

[TinyEKF](#) is a simple C/C++ implementation that I wrote primarily for running on a microcontroller like [Arduino](#), [Teensy](#), and the [STM32](#) line used in popular flight controllers like [Pixhawk](#), [Multiwii32](#), and [OpenPilot](#). Having looked over the EKF code in some of these flight controllers, I found the code difficult to relate to the understanding expressed in this tutorial. So I decided to write a simple EKF implementation that would be practical to use on an actual microcontroller, taking up a “tiny” amount of memory, while still being flexible enough to use on different projects. I also wrote a Python implementation, so you can prototype your EKF before running it on an actual microcontroller.

TinyEKF requires you to write only a single `model` function, filling in the values of the state-transition function $f(x)$, its Jacobian matrix $F(x)$, the sensor function $h(x)$, and its Jacobian $H(x)$. The prediction and update then handled automatically by passing the observation vector z to the `step` function.

Three examples are provided:

1. a pure C example using GPS satellite data stored in a file, based on You Chong's [Matlab implementation](#)
2. a C++ sensor-fusion example that runs on Arduino
3. a Python mouse-tracking example (requires OpenCV)

Previous: [The Jacobian](#)