Kalman Filter Introduction

Note: This section is currently under revision.

While the Kalman Filter is simple and elegant, it is not always obvious what it is doing conceptually or mathematically. Constructing your own with a guide, or using a code package with instructions is quite possible to do without understanding the filter. However, when choices must be made about the code, the hardware, or the values, or when general problems arise, a more thorough understanding becomes paramount.

The true algorithm for the Kalman filter is covered in the Kalman Filter section. This introduction will instead incrementally construct an equivalent algorithm starting from the concept of simple Linear Least Square Estimation, using only basic matrix operations and basic statistics.

Note: all images below have been created with simple Matlab Scripts. If seeing the code helps clarify what's going on, the .m files can all be found under internal location cs:localization:kalman.

Section 1 - Linear Least Squares Estimation

The Kalman Filter relies on a simple underlying concept – the linear least squares estimation. Given multiple noisy measurements of some state (speed, depth, acceleration, voltage, etc) the **LLSE** is an estimate that optimizes for the minimum of the sum of the squares of the errors.

In more formal terms, for some m measurements Y that are linear functions of a system with n unknown states X where $m>=n$. Such systems are said to be *over-determined,* whereby it is impossible to choose values of X that will satisfy every measurement perfectly, and thus a compromise of values of X is chosen that minimizes the total sum of the squares of the error between each measurement

$$ X_{est} = \text{arg}\,\min\limits_{\beta}\sum\|y-X\beta||^2 $$

Given the matrix format:

$$ \beta X = Y $$

$$ \begin{equation} \label{eq:control:expanded} \begin{bmatrix} \beta_{1,1} & \beta_{1,2} & \dots & \beta_{1,n} \\ \beta_{2,1} & \beta_{2,2} & \dots & \beta_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ \beta_{m,1} & \beta_{m,2} & \dots & \beta_{m,n} \end{bmatrix} \begin{bmatrix} X_{1} \\ X_{2} \\ \vdots \\ X_{n} \end{bmatrix} = \begin{bmatrix} Y_{1} \\ Y_{2} \\ \vdots \\ Y_{m} \\ \end{bmatrix} \end{equation} $$ This calculation could be performed iteratively, and the minimizing X_{est} discovered, but with the measurements enumerated in this format, matrix arithmetic offers us a simple way to solve for X_{est}. For those that recall their Geometry class in high school, to solve for X we need simply invert β and multiply that inverse by both sides.

$$ \beta^{-1} \beta X = \beta^{-1} Y \\ IX = \beta^{-1} Y \\ X = \beta^{-1} Y $$

However, you can only invert square matrices. For all $m \neq n$ this won't be the case. Here we

employ the Moore-Penrose LLSE calculation.

First both sides are multiplied by the transpose of β.

$$ \beta' \beta X = \beta' Y,\qquad \beta' = \begin{bmatrix} \beta_{1,1} & \beta_{2,1} & \dots & \beta_{m,1} \\ \beta_{1,2} & \beta_{2,2} & \dots & \beta_{n,1} \\ \vdots & \vdots & \ddots & \vdots \\ \beta_{1,n} & \beta_{2,n} & \dots & \beta_{n,m} \end{bmatrix} $$

$\beta' \beta$ will be a square matrix of size n. Assuming that at least one measurement of all states X have been included, and indicated in β this new square matrix will be invertable. We can then multiply both sides by that inverse to isolate the X state vector.

$$ (\beta' \beta)^{-1}\beta' \beta X = (\beta' \beta)^{-1}\beta' Y \\ X = (\beta' \beta)^{-1}\beta' Y $$

Remarkably, this equation will give us an X vector that satisfies the LLSE optimization.

Example:

If we want to fit a line to two points, both data points must satisfy the line equation $y = mx + b$.

$$y_1 = mx_1 + b, \\ y_2 = mx_2+b $$

We can pose the mathematical question in a matrix-format:

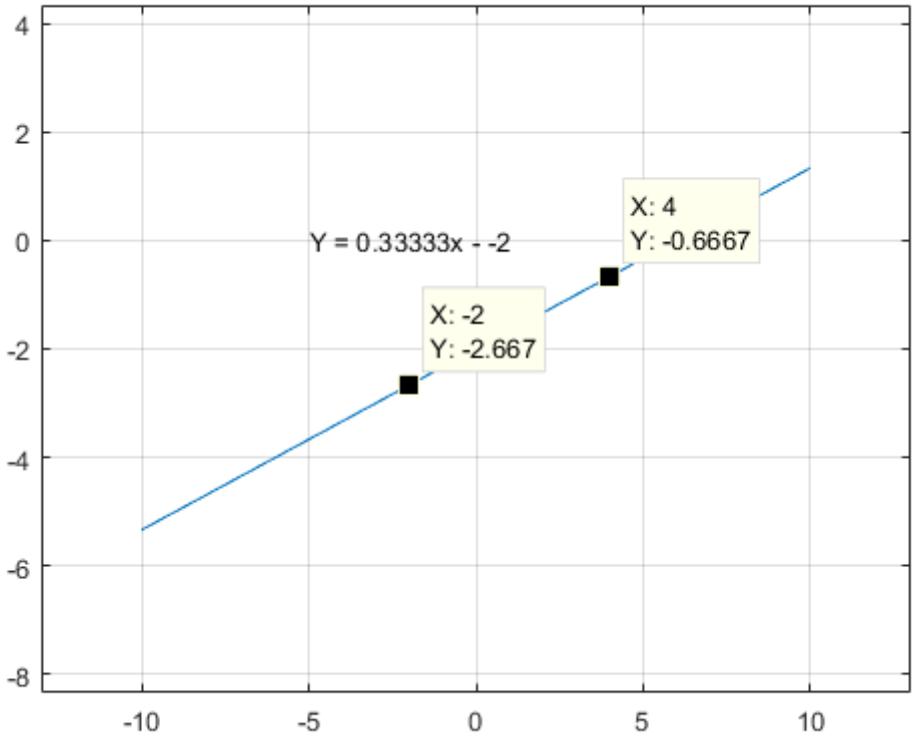
$$ \begin{bmatrix} x_1 & 1 \\ x_2 & 1 \end{bmatrix} \begin{bmatrix} m \\ b \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} $$

Here our unknown slope and y-intercept m and b form our unknown state vector X. Our dependent measurements y are used to construct our measurement vector Y, and the linear combination of independent variables and constants that relate that measurement to our state form β.

Given the two coordinate pairs $(-2, -\frac{8}{3})$ and $(4, -\frac{2}{3})$ we can find the line that is defined by them.

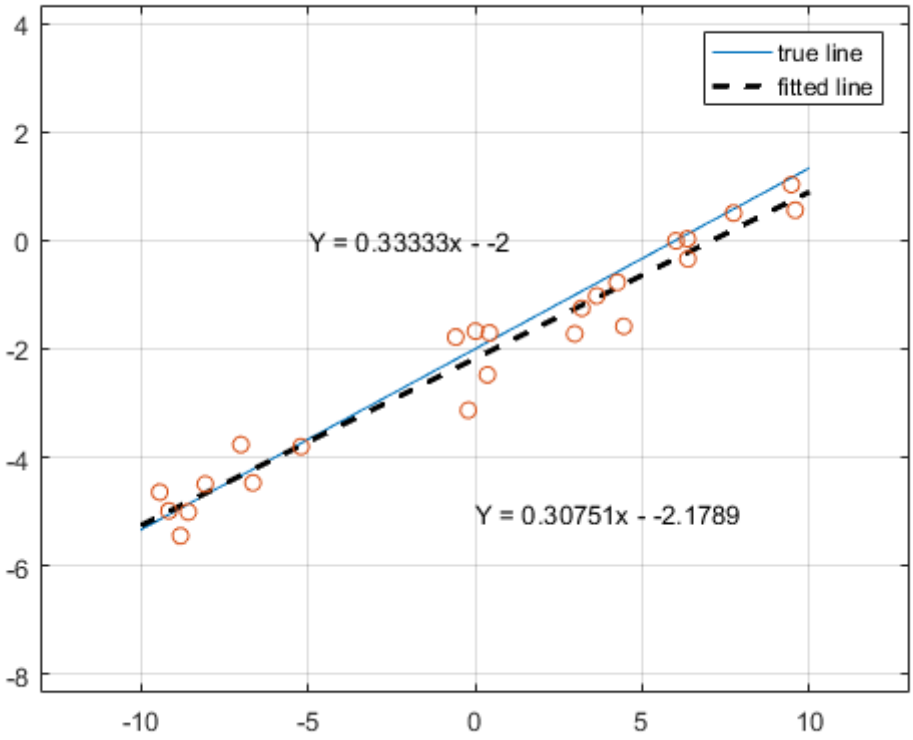
$$ \beta = \begin{bmatrix} -2 & 1 \\ 4 & 1 \end{bmatrix},\qquad X = \begin{bmatrix} m \\ b \end{bmatrix},\qquad Y = \begin{bmatrix} -\frac{8}{3} \\ -\frac{2}{3} \end{bmatrix} \\ \beta^{-1}Y = X \\ \begin{bmatrix} -2 & 1 \\ 4 & 1 \end{bmatrix}^{-1} \begin{bmatrix} -\frac{8}{3} \\ -\frac{2}{3} \end{bmatrix} = \begin{bmatrix} \frac{1}{3} \\ -2 \end{bmatrix} $$

Thus our slope $m = \frac{1}{3}$ and our linear offset $b = -2$.



Now let's try fitting a line to an over-determined set of points. Below 20 points have been randomly generated. The x components were are uniformly random samples across domain $[-10, 10]$. The corresponding y components were first calculated directly using the true line equation $y = -\frac{1}{3}x -2$, and subsequently adding samples from a Gaussian random distribution with standard deviation $\sigma = 0.5$.

$$ \beta = \begin{bmatrix} x_1 & 1 \\ x_2 & 1 \\ \vdots & \vdots \\ x_{20} & 1 \end{bmatrix}, \qquad X = \begin{bmatrix} m \\ b \end{bmatrix}, \qquad Y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_{20} \end{bmatrix} \\ X = (\beta' \beta)^{-1}\beta' Y $$



We've just *estimated* our *state* based off of *noisy* measurements in an optimal fashion. At it's core, line-fitting like this is all that the Kalman Filter is doing. These next sections we will continuously build upon this basic function until we have something resembling the Kalman Filter.

Section 2 - Performance of Multiple Noisy Sensors

We can guess intuitively that the noisier our sensors, the worse our estimation. Likewise, the more sensors (measurements) we obtain, the better our estimation. Bayesian statistics tells us that all information, no matter how noisy, is still *good* information. Indeed the entire point of this field of mathematics is to get very accurate estimations from a combination of far-less accurate measurements. But exactly *how much* better do our estimates get?

Let's consider 4 depth sensors. Each depth sensor makes a noisy measurement z_n of the depth of our submarine.

For a quick refresher, a Guassian Random Variable has a mean and a variance. The Variance **(var)**of the distribution is equal to σ^2. Standard Deviation **(std)** is simply σ. **Std** and **var** are both perfectly valid ways to describe the distribution, and their usage depends mostly on ease of understanding, or ease of mathematical operations. As a good way to conceptualize what the **std** translates to, 68% of values pulled from a Gaussian distribution will be within $\pm1\sigma$ of the mean. 95% of the values will fall within $\pm2\sigma$.

If we simply read of one depth sensor, the estimate of our depth will have an equal noise.

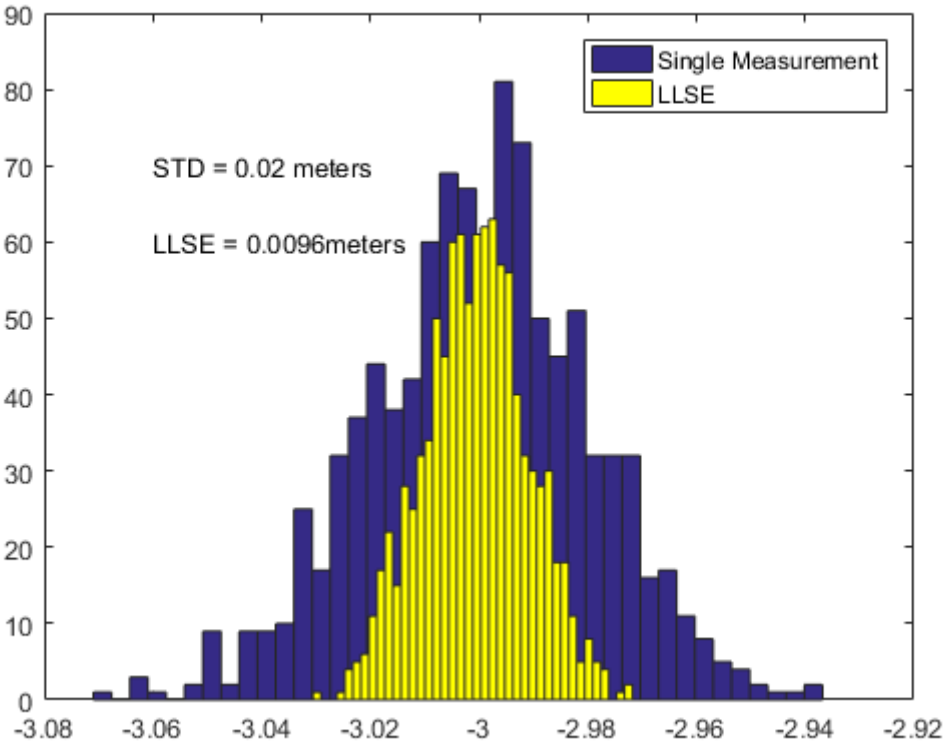
If we average the results of all four depth sensors, we will get a better estimate. Specifically, the Variance of an estimate is inversely proportional to the number of measurements n taken.

$$z_1 \sim\mathcal{N}(0,\sigma^2),\qquad \frac{z_1 + z_2 + z_3 + z_4}{4} \sim\mathcal{N}(0,\frac{\sigma^2}{4}),\qquad $$

and thus the estimate from n depth sensors of std σ will have an std of $\frac{\sigma}{\sqrt{n}}$. For this example, we'll assume the noise is Gaussian, with a mean of zero (no bias) and a standard deviation of $\sigma = 2cm = 0.02m$. The actual depth we're measuring is $3m$ Thus, we'd expect the distribution of our estimate to be $\sigma/2$ or $0.01m$. Let's see if that happens.

$$ \beta = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix},\qquad X = \begin{bmatrix}Z\end{bmatrix},\qquad Y = \begin{bmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \end{bmatrix} \\ X_{est} = (\beta'\beta)^{-1}\beta' Y $$

If we create a script that generates 4 measurements by taking our true depth and adding a sample from a Guassian distribution with std $\sigma = 0.02$, then we can use the above equation to estimate our depth. $y_{1,2,3,4} = 3 + \sim\mathcal{N}(0,.02^2)$. If we estimate our depth 1000 times, we should get a distribution of estimated depths that has a standard deviation of $\frac{\sigma}{\sqrt{4}} = \frac{0.02m}{2} = 0.01m$.



Now we know how confident we can be in our estimates given multiple, identical sensors.

Section 3 - Weighted Least Squares

The underlying assumption of the Linear Least Squares Estimation is that all measurements hold equal weight. That is to say, all are equally noisy and should be trusted equally. This can work well for fusing the results of duplicate sensors, but becomes a poor assumption when combining different

sensors.

Some of the cleverer readers might be thinking of a simple workaround – just add copies of the more accurate sensors' measurements into the matrix so the system listens to them more. And that'd certainly emulate what we're trying to do. But it's not mathematically perfect, nor is it elegant to code or computationally efficient to run.

If you have different sensors of different quality, we need to move to the *Weighted Linear Least Squares Estimation* **(WLLSE)**.

Before for **LLSE** we had:

$$ X = (\beta' \beta)^{-1}\beta' Y $$

Our new equation for *WLLSE* is:

$$ X = (\beta'W \beta)^{-1}\beta' WY,\qquad W = \begin{bmatrix} \frac{1}{\sigma_{1}^2} & 0 & \dots & 0 \\ 0 & \frac{1}{\sigma_{2}^2} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots & \\ 0 & 0 & \dots & \frac{1}{\sigma_{n}^2} \end{bmatrix} $$

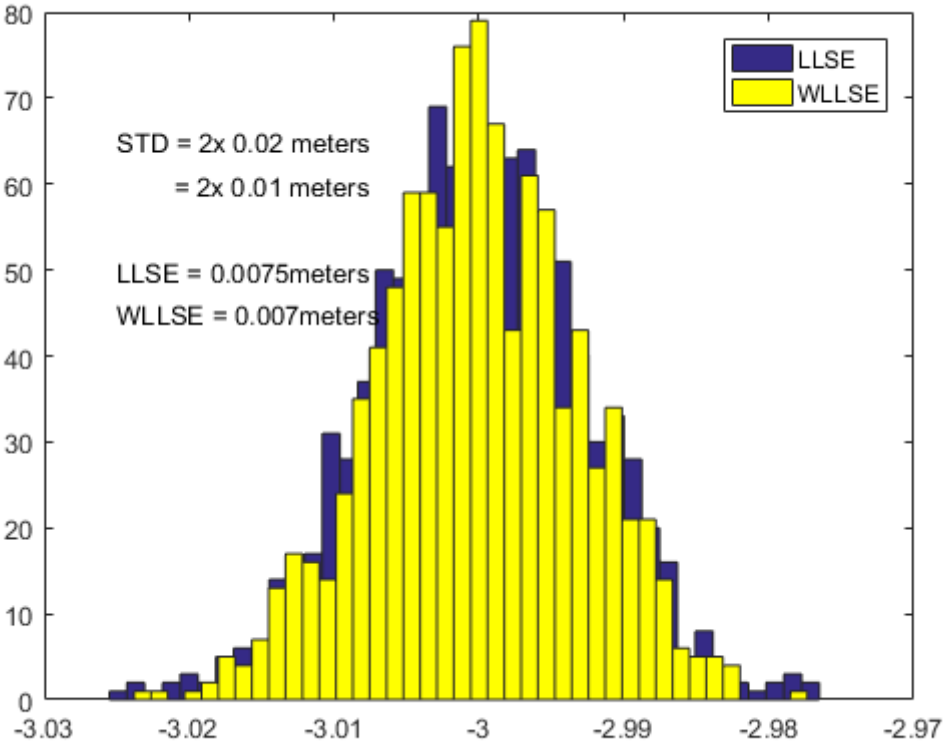
Working out the exact operations going on here is an exercise left to the student. However, intuitively we can see what's happening. When you multiply a matrix by an identity matrix I_n it remains unchanged. More generally, If you multiply a matrix by a diagonal matrix, it will scale each row or column (depending on order of operation) by it's corresponding value. Here we're inserting a scaling diagonal matrix in the middle of our **LLSE** solution. And we're scaling each value by the inverse of its variance.

Thus, measurements enumerated in the β matrix with low variances will be scaled by large numbers, and those with large variances will be similarly shrunk. In this way, the measurements are given more weight according to how well they can be trusted.

If we repeat our example from Section 1.2, but reduce the noise of two of the depth sensors, we should get an ever better estimation. For Depth sensors 1 and 2, $\sigma = 0.02m$. For Depth Sensors 3 and 4, $\sigma = 0.01$.

$$ \beta = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix},\qquad X = \begin{bmatrix}Z\end{bmatrix},\qquad Y = \begin{bmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \end{bmatrix},\qquad W = \begin{bmatrix} \frac{1}{0.02^2} & 0 & 0 & 0 \\ 0 & \frac{1}{0.02^2} & 0 & 0 \\ 0 & 0 & \frac{1}{0.01^2} & 0 \\ 0 & 0 & 0 & \frac{1}{0.01^2} \end{bmatrix} \\ X_{est} = (\beta'W\beta)^{-1}\beta WY $$

Recall that in **Section 1.2** we used four depth sensors of **std** $\sigma_z = 0.02$ and got an estimate with **std** $\sigma_{est} = 0.01m$.



By using a regular **LLSE**, where we assume all four depth sensors are equally trustworthy, we get a reduced std of $\sigma_{est} = 0.0075m$ simply by virtue of having more accuracy in some sensors. However, by using **WLLSE** we tighten the estimate down to $\sigma_{est} = 0.007$. Both estimates made here were made with identical information. The only question was how we processed that information. In this regard, appropriately trusting different sensors by different amounts gives us an overall better estimate.

However, the question now is, *exactly* how much should we trust our overall estimate? It's fine to make many estimates many times and find that distribution, but there should be *some* way to mathematically *know* how trustworthy our estimate is, based off of how trustworthy our various sensors were.

There is. In fact, we already calculated it. $$ R = inv(\beta' W \beta) = 5.413e^{-5},\qquad R = \begin{bmatrix} \sigma_{1}^2 & \sigma_{1}\sigma_{2} & \dots & \sigma_{1}\sigma_{n} \\ \sigma_{2}\sigma_{1} & \sigma_{2}^2 & \dots & \sigma_{2}\sigma_{n} \\ \vdots & \vdots & \ddots & \vdots & \\ \sigma_{n}\sigma_{1} & \sigma_{n}\sigma_{2} & \dots & \sigma_{n}^2 \end{bmatrix} \\ \sigma^2 = 5.413e^{-5},\qquad \sigma = 0.0756 $$

Where R is the co-variance matrix of the estimated state. The diagonals of R reflect the variance of each estimated state. The off-diagonal terms are co-variances. Put simply, two random variables can have normal distributions, but share some underlying property that makes them correlate. If one is above the mean, the other is likely above the mean. Or vice-versa. This is important when you start trying to calculate things from linear combinations of state elements. Mathematically, $$var(aX+bY) = a^2Var(X) + b^2Var(Y) + 2abCov(X,Y)$$. Normally separate states will be uncorrelated, and have $Cov(X,Y)=0$. However, when you're dealing with physical models and sensors, correlations start to pop up, like the position and the velocity of your vehicle.

If you had an overestimated position but an underestimated velocity, and you try to predict where you'll be in 1 second, your predicted position could be pretty accurate. But over time, if your velocity

is being under-estimated for too long, you're also likely underestimating your position. If you know that more often that not, your position and velocity are both over or under estimated, it's more likely the predicted position based on your estimates is *further* from the true position than if they were uncorrelated.

Thus, if you try to predict your location 1 second in the future, $x[k+1] = x[k] + \dot{x}[k] * 1sec$, the variance of that predicted location isn't going to simply be the noise of your current position plus the noise of your velocity, but additionally some extra noise because both states are inter-related and more liable to compound their noise rather than counteract it. That's where the $2cov(X,Y)$ term comes in.

And if you think that equation for variance looks remarkably like the Law of Cosines in Trigonometry - *Congratulations*. You get a cookie.

Section 4 - Modeling a System

Taking a break from probability and estimation, we need to consider how to model a system in time. Think of it like modeling the position of a shell fired from a cannon. Physics 101 stuff.

We're going to put our system into something called the State-Space format. It equates the derivative of the system as a linear function of the current state of the system, plus any current external forces.

$$\dot{x}(t) = Ax(t) + Bu(t)$$

where $x(t)$ is the current state and $u(t)$ are inputs to the state. The reasons for putting it in this form are good, but can be a little unclear in the continuous time domain. Since the Kalman Filter operates in discrete-time anyway, we'll move there. The equivalent form is:

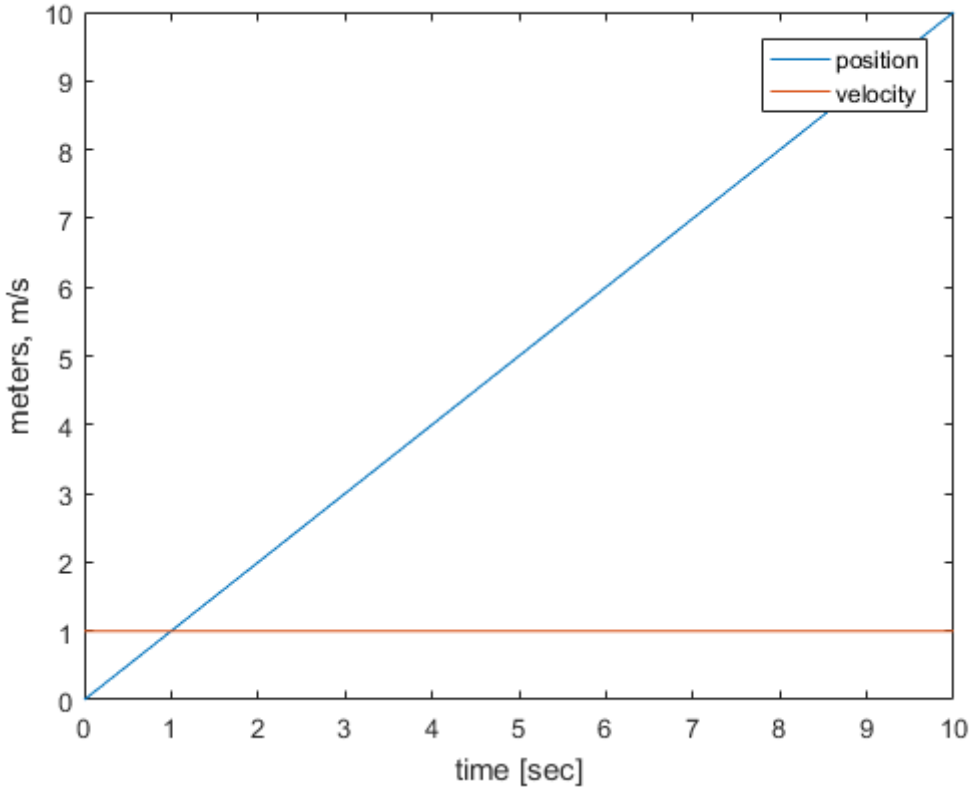
$$x[k+1] = Ax[k] + Bu[k]$$

Where we model the next state as a linear function of our current state and current inputs. As an example, let's say our state is a vector of two variables, a position vector x, and our velocity, $\dot{x}$, and we want to make a discrete-time model of our system with position, velocity, and acceleration. Our State Space representation would be:

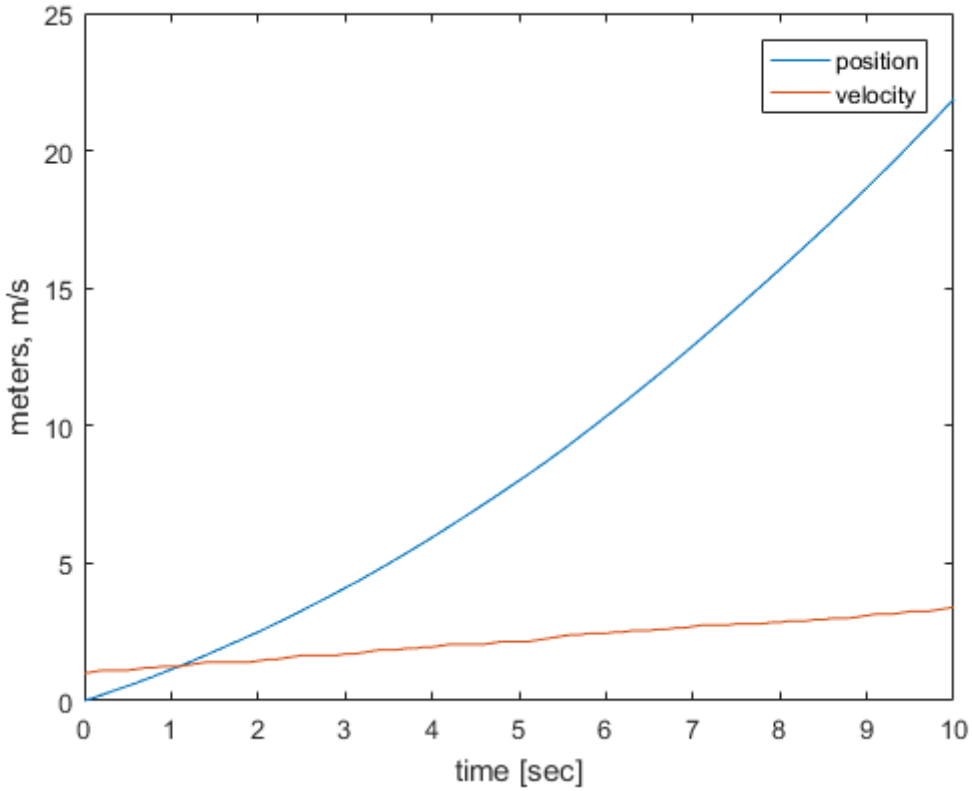
$$ x[k+1] = x[k] + \dot{x}[k]dt + \frac{1}{2}\ddot{x}[k]dt^2 \\ \dot{x}[k+1] = \dot{x}[k] + \ddot{x}[k]dt \\ X = \begin{bmatrix} x \\ \dot{x} \end{bmatrix},\qquad A = \begin{bmatrix} 1 & dt \\ 0 & 1 \end{bmatrix},\qquad B = \begin{bmatrix} \frac{dt^2}{2} \\ dt \end{bmatrix},\qquad u = \ddot{x} $$

For computer modeling, you can see why this is a very useful format. Matrix A called the Transition Matrix, multiplied by the current state X will calculate the state X at the next time-step, barring any inputs. External inputs influence the system through the B matrix, appropriately called the Input Matrix.

Simply placing this equation into a for loop will allow you to model the system iteratively. Here is a system with an initial position of $0m$ and an initial velocity of $1m/s$ with no external input.



That's rather boring, however. Here's a more interesting version, with random inputs.



It doesn't look like much, but this format is infinitely scalable in complexity. Linear systems with two, or ten, or two hundred states can all have their dynamics modeled in this simple manner. But we'll leave building our Submarine model for a different tutorial.

The second part of the State Space format is the measurement vector equation.

$$ Y[k] = CX[k] + Du[k] $$

This model doesn't iterate the future, but rather calculates measurements Y of sensors, as linear functions of the state X, specified by the Emission Matrix C. Matrix D specifies distortions to measurements that occur as functions of the input. While inputs distorting sensors is not uncommon, matrix D is seldom used. Often distortions from inputs are removed from the measurements separately, as a form of calibration, prior to solving for state estimations.

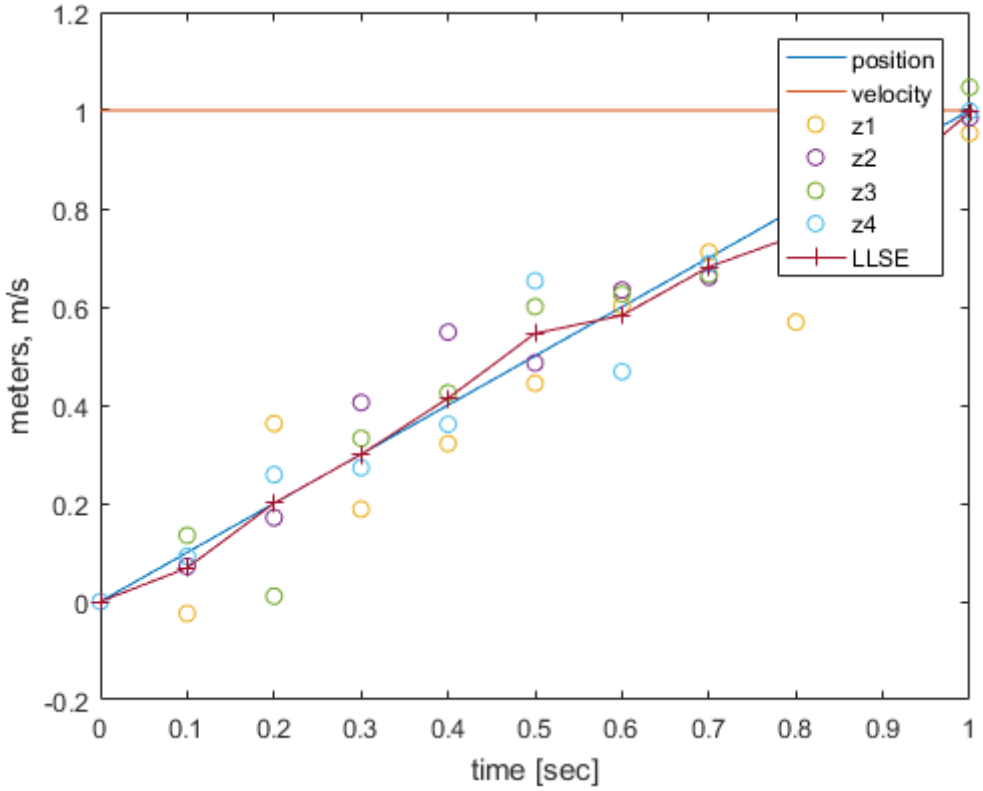
In our previous examples with depth sensors, C was a simple vector of $1s$ because our only state was depth, and it was being measured directly. As with the Transition matrix, the Emission matrix can rapidly become complex as the type of sensors, states, and measurements being considered expand. However, creating the Emission Matrix will be covered under a different Localization Topic. Here, all we care about is creating a systematic format for our system dynamics and our sensor measurements.

Example

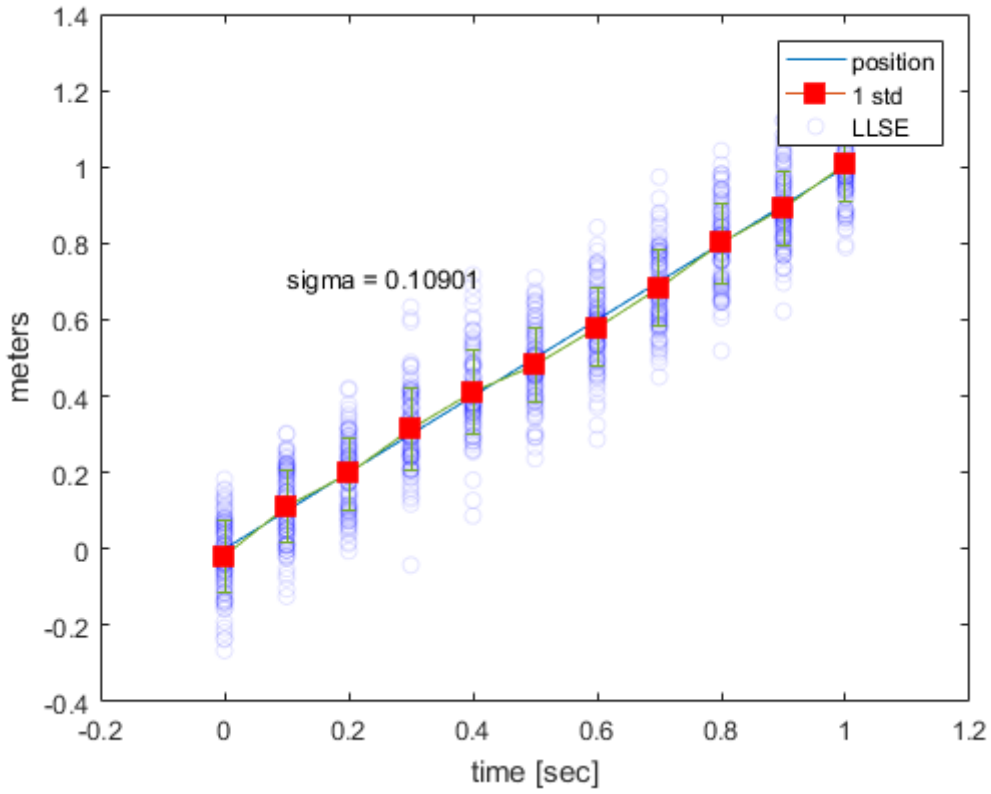
Let's use our State Space Model to make an example with our Depth Sensors.

$$ dt = 0.1sec,\quad X[0] = \begin{bmatrix} 0m \\ 1ms^{-1} \end{bmatrix},\quad u = 0,\quad A = \begin{bmatrix} 1 & 0.1 \\ 0 & 1 \end{bmatrix},\quad B = \begin{bmatrix} 0.005 \\ 0.1 \end{bmatrix},\quad C = \begin{bmatrix} 1 & 0\\ 1 & 0\\ 1 & 0\\ 1 & 0 \end{bmatrix},\quad $$ and we'll use our emission matrix to generate measurements $Y[k]$ from our true state $X[k]$, add noise of $\sigma = 0.02m$, and then calculate an estimate of our position $x_{est}[k]$ through our LLSE equation.

Modeling this from $t=[0,1]$ gives us:



That seems to track the true state pretty well. Let's see exactly how well, by following the state 1000 times.



We see that at each point in time, we take many measurements, and estimate a position that is close to our true position. If we did this 1000 times, we can see how accurate we are at tracking our system

at any one time. If you notice, our error in measurement at any point in time is roughly constant.

Next, we'll see how we can do *better*.

Section 5a Previous State as a Measurement - Intuition

Now that we know how to model our physical systems, and take measurements at individual points in time, we can start to get to the root of the Kalman Filter.

Consider this scenario: Our submarine is diving under water. Our depth sensors estimate we're at a depth of about 7.5 meters. Let's say I asked which 'true depth' is more likely: $7.3m$ or $7.5m$ or $7.7m$? As we have nothing else to go on, $7.5m$ is the obvious pick.

Now consider that the last three measurements over the last three seconds have been 4.1 meters, 5.1 meters, 6.0 meters, and now 7.5 meters. Also, assume that our thrusters have been running at a constant rate. We seemed to be descending at about 1 meter/second, and suddenly we jump to 1.5 meters/second, without any significant change in deliberate thrust. Something isn't adding up. We'd expect to be at around $7m-7.2m$. Being $0.5m$ deeper would indicate an acceleration of $1ms^{-2}$. For a $20kg$ submarine, ignoring drag, that'd require an extra $20N$ of force (4 pounds) coming from no-where. Or random currents in the water suddenly sped up by half a meter per second.

Both are possible… but extremely unlikely. The more likely explanation is that our depth sensors have all randomly measured high, and have significantly over-estimated our depth. If I asked "which 'true depth' was more likely, 7.3m or 7.5m or 7.7 meters?", you wouldn't be certain, but you'd probably now guess 7.3m is closer to the truth. We're now making a reasonable judgement we couldn't make before when we just had the individual measurement. That means that we are somehow getting extra information from knowing our previous position and velocity. Put another way, we're somehow using our prior state as some sort of extra measurement!

Section 5b Previous State as a Measurement - Math

Okay, so we can look at an individual scenario and make a gut-feeling intuitive modification. But how can we properly, justifiably, mathematically program the computer to do the same?

First, let's be explicit about our State-Space models. The models we've created so far have been assuming they are perfect models. This is never the case when doing anything real-world. The true form of our State Space system is as follows:

$$ X_{k+1} = AX_k + B(u_k + \sim\mathcal{N}(0,Q^2)) \\ Y_k = CX_k + \sim\mathcal{N}(0,\sigma^2) $$

When modeling the dynamics of a system, as in the first equation, there are always going to be unexpected perturbations. Both errors in your expected input, and random external forces, which serve to modify the overall state. Even if these modifications are slight, overtime they can accumulate and cause the real system to deviate from its modeled path.

While the noise $var(u)$ can be be empirically determined for some systems, predicting the unknown environment is a much more difficult task, and often simply estimated. As there is no discernible difference in an inaccurate input, and an external force on the system, the uncertainty for both is encapsulated in the Process Noise Q.

Likewise, the measurement noise σ can be treated as a random Gaussian variable added to what measurements the sensors *should* be getting given the actual state.

With that out of the way, we need to start considering how to transmit information about our previous state to our new state as a measurement.

We want to provide as much information to the system as we can, with no redundant equations, and in a way where we can accurately evaluate how noisy these 'measurements' should be. There are likely many valid equations to derive for this, but the one below is functional.

With our state-space form, we need to somehow provide numerical values $f(x_{k-1},\dot{x}_{k-1},u_{k-1})$ and equate them to a linear function $g(x_k,\dot{x}_k)$. Then they'll fit into our C Emission matrix and Y measurement vector.

Starting from here: $$ x[k+1] = x[k] + dt\dot{x}[k] + \frac{dt^2}{2}u[k] \\ \dot{x}[k+1] = \dot{x}[k] + dtu[k] $$

We can find $$ x[k] + dt^2/2u[k] = x[k+1] – dtx'[k] \\ dtx'[k] = dtx[k+1] – dt^2u[k] $$ so $$ x[k] + dt^2/2u[k] = x[k+1] – dtx'[k+1] + dt^2u[k]\\ x[k]-dt^2/2u[k] = x[k+1] – dt\dot{x}[k+1] $$ and from above $$ \dot{x}[k+1] = \dot{x}[k] + dtu[k] $$

So now we have our previous position, minus some amount of our previous input, equal to what our current position minus our current velocity *should be*. Similarly, our present velocity *should be* equal to our previous velocity plus some amount of our previous acceleration. Thus we can augment our C Emission matrix and Y measurement matrix: $$ C = \begin{bmatrix} 1 & 0\\ 1 & 0\\ 1 & 0\\ 1 & 0\\ 1 & -dt \\ 0 & 1 \end{bmatrix},\qquad Y = \begin{bmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \\ z_{k-1} + \frac{dt^2}{2}u_{k-1} \\ \dot{z}_{k-1} + dtu_{k-1} \end{bmatrix} $$

We are providing information about two unknown states with two equations that include all 3 pieces of information we possess, our previous depth, our previous vertical velocity, and our previous vertical acceleration. We can be confident we've transmitted all the information we possess to help inform the next state.

So we need to find how noisy these two new 'measurements' are:

$$ Var(z_{k-1} + \frac{dt^2}{2}u_{k-1}) = Var(z_{k-1}) + \frac{dt^4}{4}Var(u_{k-1}) + dt^2Cov(z_{k-1},u_{k-1}) \\ Var(\dot{z}_{k-1} + dtu_{k-1}) = Var(\dot{z}_{k-1}) + dt^2Var(u_{k-1}) + dtCov(z_{k-1},u_{k-1}) $$

We can assume that input noise isn't correlated with our depth or velocity, so we're left with the scaled sum of the variances for each 'measurement'. We'll designate these 'measurements' σ_a and σ_b for simplicity.

$$ \sigma_a^2 = Var(z_{k-1}) + \frac{dt^4}{4}Var(u_{k-1}) \\ \sigma_b^2 = Var(\dot{z}_{k-1}) + dt^2Var(u_{k-1}) $$

However, we can't calculate these yet. We'd need to know how accurate our the estimate of our previous position and velocity are. We don't even *have* a previous state yet!

Recall back to **Section 3** that we can determine the noise of the estimations of our WLLSE from the Covariance Matrix R as a function of our β and W matrices.

$$ R = inv(\beta' W \beta) = \begin{bmatrix} \sigma_{1}^2 & \sigma_{1}\sigma_{2} & \dots & \sigma_{1}\sigma_{n} \\ \sigma_{2}\sigma_{1} & \sigma_{2}^2 & \dots & \sigma_{2}\sigma_{n} \\ \vdots & \vdots & \ddots & \vdots & \\ \sigma_{n}\sigma_{1} & \sigma_{n}\sigma_{2} & \dots & \sigma_{n}^2 \end{bmatrix} $$

Which, for our example with depth sensors

$$ R = inv(C' W C) = \begin{bmatrix} \sigma_{z}^2 & \sigma_{z}\sigma_{\dot{z}} \\ \sigma_{\dot{z}}\sigma_{z} & \sigma_{\dot{z}}^2 \end{bmatrix} $$

Now we can describe the measurement errors σ_a and σ_b as functions of R and create our W matrix.

$$ \sigma_a^2 = R_{k-1}(1,1) + \frac{dt^4}{4}Q^2,\qquad \sigma_b^2 = R_{k-1}(2,2) + dt^2Q^2 \\ W = \begin{bmatrix} \frac{1}{\sigma_{z1}^2} & 0 & 0 & 0 & 0 & 0 \\ 0 & \frac{1}{\sigma_{z2}^2} & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{\sigma_{z3}^2} & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{\sigma_{z4}^2} & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{\sigma_{a}^2} & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{1}{\sigma_{b}^2} \end{bmatrix} $$

Now we have R_k as a function of W, and W as a function of R_{k-1}. This might seem circular, but in fact it's *iterative*. The accuracy of the estimation of our current state X_k is a function of the accuracy of the estimation of our previous state X_{k-1}.

Example

Putting everything together now, let's model a system and track it with our new iterative WLLSE algorithm.

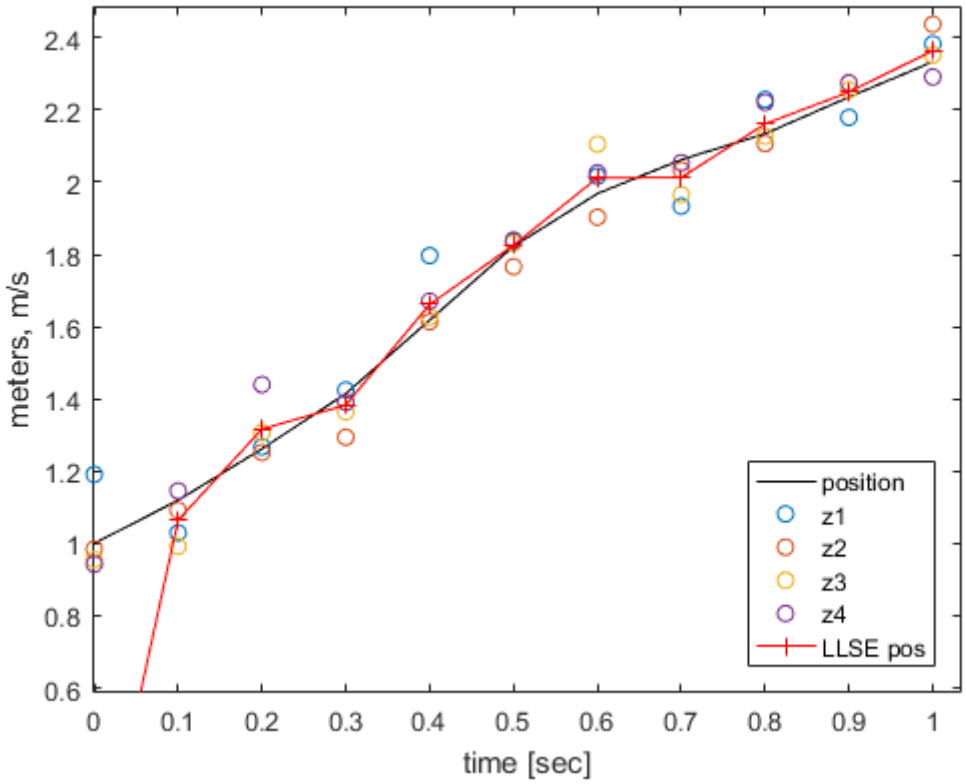
Starting Parameters: $$ X[0] = \begin{bmatrix} z \\ \dot{z} \end{bmatrix} = \begin{bmatrix} 0m \\ 1ms^{-1} \end{bmatrix},\qquad X_{est}[0] = \begin{bmatrix} 0m \\ 0ms^{-1} \end{bmatrix},\qquad R[0] = \begin{bmatrix} 9999m & 0 \\ 0 & 9999m \end{bmatrix},\qquad Q = 10ms^{-2},\qquad \sigma_z = 008m,\qquad u = 0ms^{-2} $$

Model: $$ dt = 0.1sec,\qquad A = \begin{bmatrix} 1 & dt \\ 0 & 1 \end{bmatrix},\qquad B = \begin{bmatrix} \frac{dt^2}{2} \\ dt \end{bmatrix},\qquad C = \begin{bmatrix} 1 & 0\\ 1 & 0\\ 1 & 0\\ 1 & -dt \\ 0 & 1 \end{bmatrix} \\ X_{k+1} = AX_k + B(u_k + \sim\mathcal{N}(0,Q^2)) $$

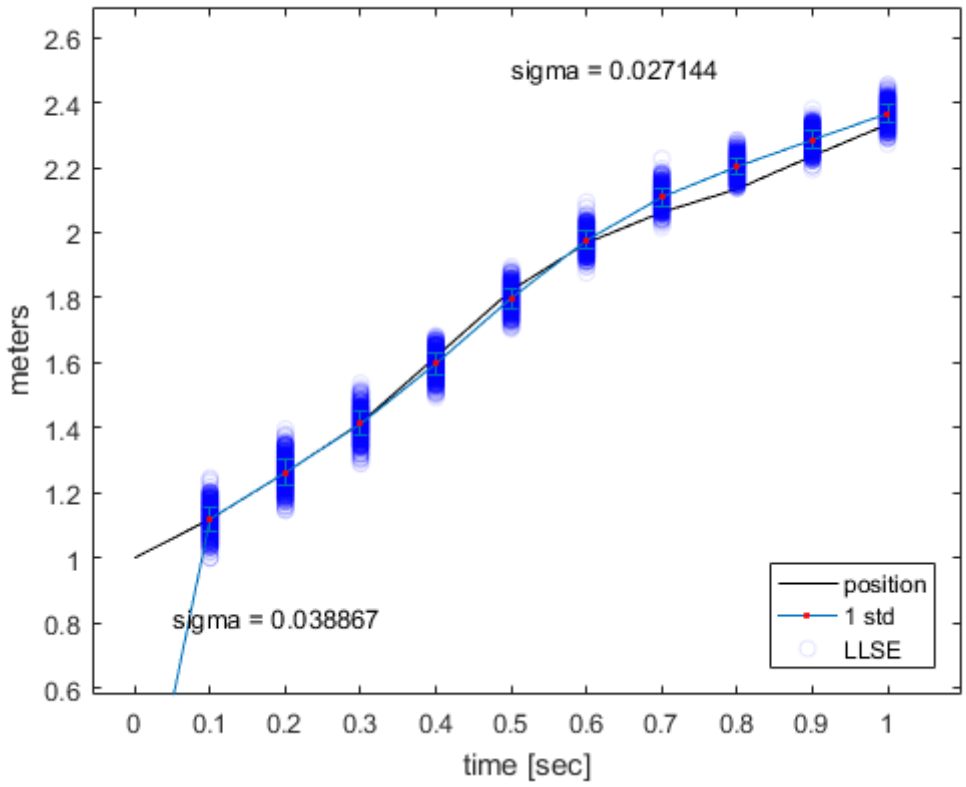
Estimation Process:

$$ \sigma_a^2 = R_{k-1}(1,1) + \frac{dt^4}{4}Q^2,\qquad \sigma_b^2 = R_{k-1}(2,2) + dt^2Q^2 \\ W_k = \begin{bmatrix} \frac{1}{\sigma_{z1}^2} & 0 & 0 & 0 & 0 & 0 \\ 0 & \frac{1}{\sigma_{z2}^2} & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{\sigma_{z3}^2} & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{\sigma_{z4}^2} & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{\sigma_{a}^2} & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{1}{\sigma_{b}^2} \end{bmatrix},\qquad Y = \begin{bmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \\ z_{k-1} + \frac{dt^2}{2}u_{k-1} \\ \dot{z}_{k-1} + dtu_{k-1} \end{bmatrix} \\ X_{est}[k] = (C'W_kC)^{-1}C'W_kY \\ R_k = (C'W_kC)^{-1} $$

After every update of the model, the R covariance matrix is used to recalculate W, and the X_est state vector estimate is used with fresh samples to create a new Y measurement vector, which are then each used to form a new estimate of R and X_est for this cycle, and to be used in the next.



It looks like this estimation is following the line pretty closely. Let's see how well 1000 estimations follow this path caused by random inputs.



Notice how originally, the std of the estimation is close to $0.04m$? That's exactly what we'd expect from just using 4 depth sensors with $\sigma = 0.08m$. However as time goes on and more estimates are made, the noise of the position estimate drops to around $0.027m$.

Now we can start to see the full power of the Kalman filter taking shape. The better our estimates get, the better our predictions get, and thus, the better our future estimates get. As time goes on we get a more and more accurate state estimation.

It doesn't converge to an error of 0, of course. The Process Noise Q guarantees that we can't perfectly predict the future, which sets a lower bound on how well we can guess forward in time. But using nothing more than the same 4 depth sensors, we've significantly increased the accuracy of our information.

End

That's all there is to it. The true algorithm for the Kalman Filter upon first glance, will seem a bit different that what we've done here. It utilizes separate prediction and measurement steps, and then compromises between them based off of a factor K called the Kalman Gain. However, looking more closely, you'll notice that both our algorithm here, and the true Kalman Filter algorithm require the same input data, and have very similar mathematical patterns.

In fact, mathematically the two are identical. The Kalman Filter is simply a more elegant way of performing the same calculations - it doesn't require inventing linear relationships from previous to current states, and doesn't require a modification of the measurement Variance matrix W each iteration.

Fundamentally, both systems are doing the same thing. They are applying a Linear Least Square Estimate, with measurements weighted based on each sensor's noise, combined with information from the previous state, to propagate forward an increasingly accurate estimate.