FRANKLIN

Kalman Filter Made Easy

Learn the Kalman Filter with Real Life Examples and Python

William Franklin

Kalman Filter Made Easy

By William Franklin

Table of Contents

Tribute to the Kalman Filter

This book is an introduction to the Kalman Filter and its closely related algorithms, Extended Kalman Filter and Unscented Kalman Filter. Many books and papers already cover the deeply technical aspects of these but, this book is meant to talk you through applications of these algorithms so you can see how they work in real life, rather than just theoretical.

The Kalman Filter state estimation technique is widely used for a variety of applications. It has made its way into race cars, missiles, radars, rocket ships, lunar landers and many other places. I can't stress enough how important this algorithm is to the modern world and how it continues to be used in emerging technologies.

So as you continue to build and create in this ever changing world, make sure to have the Kalman Filter in your toolbelt. More times than not, you will find a practical way for it to improve your system. Once you have your "Aha!" moment while reading this book, you will never look back. Each application of the Kalman Filter is different, this book will prepare you to solve your problem when the opportunity arises!

Book Overview

The Kalman Filter is an algorithm that was developed over 60 years ago and is well understood by many academics. Unfortunately, there does not seem to be a good introductory book on the market that could be used by beginners or more experienced professionals looking to gain an understanding of how to implement and use a Kalman Filter.

More specifically, it seems as though every book believes the reader should have a full understanding of how the Kalman Filter works before implementing it. By analogy, what if every person who decided to get a driver's license needed to fully understand how a combustion engine works in synchrony with the drive shaft to allow the driver to drive? This seems silly right? Drivers can benefit from that information if something goes wrong but they really just need to know how to turn the key, press the gas, use the brake, and use mirrors to do 95% of the driving. This book is going to treat the Kalman Filter like a vehicle and show the reader how to drive or how to use it. For those interested, it will also include some information about the derivations of the Kalman Filter.

The secret to understanding the Kalman Filter is that an in-depth mathematical understanding is not needed to use it. The Kalman Filter has been proven to work over the past 50 years. This book will not spend much time convincing the reader that this approach is mathematically sound, there is a 60 year history to prove that.

With all of that said, this book will use examples, plots, and images to show the reader to implement and tune a Kalman Filter. After reading through the examples and going over the levers and knobs used to make adjustments, the reader will be well equipped to apply the same techniques to their problems.

Enjoy!

Chapter 1: What is a filter?

A filter is process that removes random error from a dataset that is attributed to noise and tracks the variation of the measured quantity. There are many different types of filters that have been created over the years, such as: average filter, moving average filter, low-pass filter (exponential moving average), Kalman Filter and others. Filters are necessary to improve the quality of information that is noisy.

On Uncertainty in Measurements

For any project or experiment, data is gathered through observation or a measurement instrument. This data is not perfect and contains errors. There are two types of errors: random noise and bias.

Random noise can be thought of as the error attributed to inaccurate measurement. For example, if you had ten people measure the length and height of a car to the nearest centimeter. Even though the cars length and height did not change, your measurements would be different across the ten data points. The error between each measurement and the true length and height of the vehicle is considered random noise. Random noise is considered to be a standard normal distribution. Figure 1 shows a standard normal distribution plot.

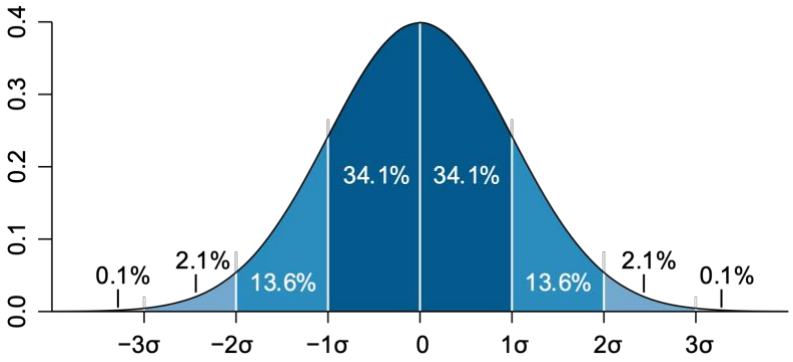


Figure 1: Standard Normal Distribution

Bias is a systematic error that causes measurements to be different from the true values. Let's stick with the same example from above. What if the measuring device used by the ten participants was off by 5 centimeters? This would mean that in addition to the random error, each measurement was actually 5 centimeters shorter than was reported. This bias error would be consistent across all measurements.

Continuing on with this example, if you took a look at the actual car length and car height and compared it with the measurements. It could be seen that each measurement was missing ~5 centimeters in both dimensions as well as each measurement would be different then the next. So even though the car's height and length did not change, it seems as though 10 different people

computed 10 different measurements. This example was a simple one but hopefully you can see what **random noise** and **bias error** represent when breaking up the error in a measurement into multiple components.

There are many other examples of measurement error that take place in complex systems across the world and across a range of industries. Based on what these systems need to do, they may choose to implement a Kalman Filter to receive and process these measurements real-time to remove that random error and compute accurate estimates.

Types of Filters

As mentioned above, there are many types of filters. I will now go through a few of them and explain which applications they work well for and the applications they don't work well for.

Average Filter

You already know what the average filter is but you may not know it by this name. The average filter is a routine that computes the average value of a data set. Simple as that. For applications where the end result will be constant and not vary, for example a digital food scale measuring the weight of an apple, this approach works well. The scale will come up with multiple measurements from the time the apple hits the surface until the apple settles its weight on the surface**. Eventually the measurements will converge on a single point. If enough measurements are taken consecutively, the average of all of these values will be near the actual weight of the apple and noise of the measurements will be removed. We will call the result the mean instead of the average.**

Figure 2 below displays how the measured weight of the apple settles over time based on how much variance there is in the preceding measurements. If a measurement is taken every 100^{th} of a second, this settling can occur in just one second. Next time you are using a digital scale, take note of the initial variance in measurements before it settles down to its final number.

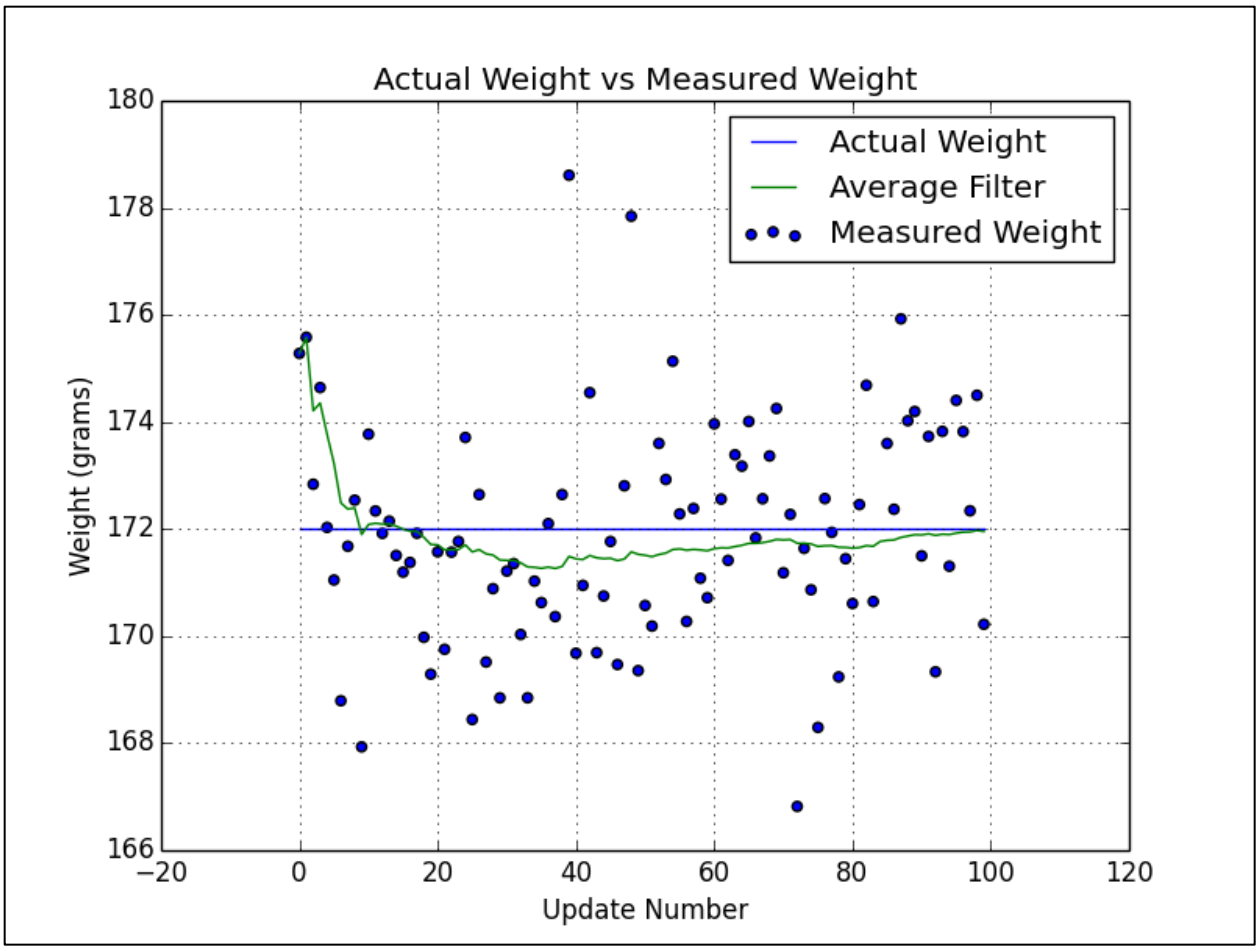


Figure 2: Weighing an Apple with Average Filter

The average filter can be implemented recursively, Eqn. 2, or with the traditional batch approach, Eqn. 1; where the subscript ***k*** is the number of data points being averaged, ***y-bar*** is the mean, and ***y*** is the measurement. The recursive approach is preferred because it is less memory intensive but it is harder to derive and implement.

$$\bar{y}_k = \frac{y_1 + y_2 + \cdots + y_k}{k} \tag{1}$$

$$\bar{y}_k = \frac{k-1}{k}\bar{y}_{k-1} + \frac{1}{k}y_k \tag{2}$$

So why don't we use the average filter approach for all problems that we need to remove error? What happens when a measured quantity is varying with time? Is the average filter the best method to remove noise and estimate the mean? **The answer is no.** That is why the moving average filter was created.

Moving Average Filter

Have you ever seen a stock price graph? If you have, then you are most likely familiar with the 5-day and 7-day moving average plots that are plotted right next to the actual stock price of that day. They filter out the noise and show the general trend of the stock price. The purpose of these filters is remove noise and track the variation of the stock price as it changes.

As the stock price jumps or drops, the moving average follows it unlike the average filter. So when a measured quantity is changing over time, the moving average is preferred over the average filter because it will move with the actual value. Figure 3 shows the difference between using an average filter and a moving average filter if you were trying to remove the noise from the opening stock price for GameStop (GME) in the beginning of 2021. It was during that time frame where the stock price saw massive swings in values.

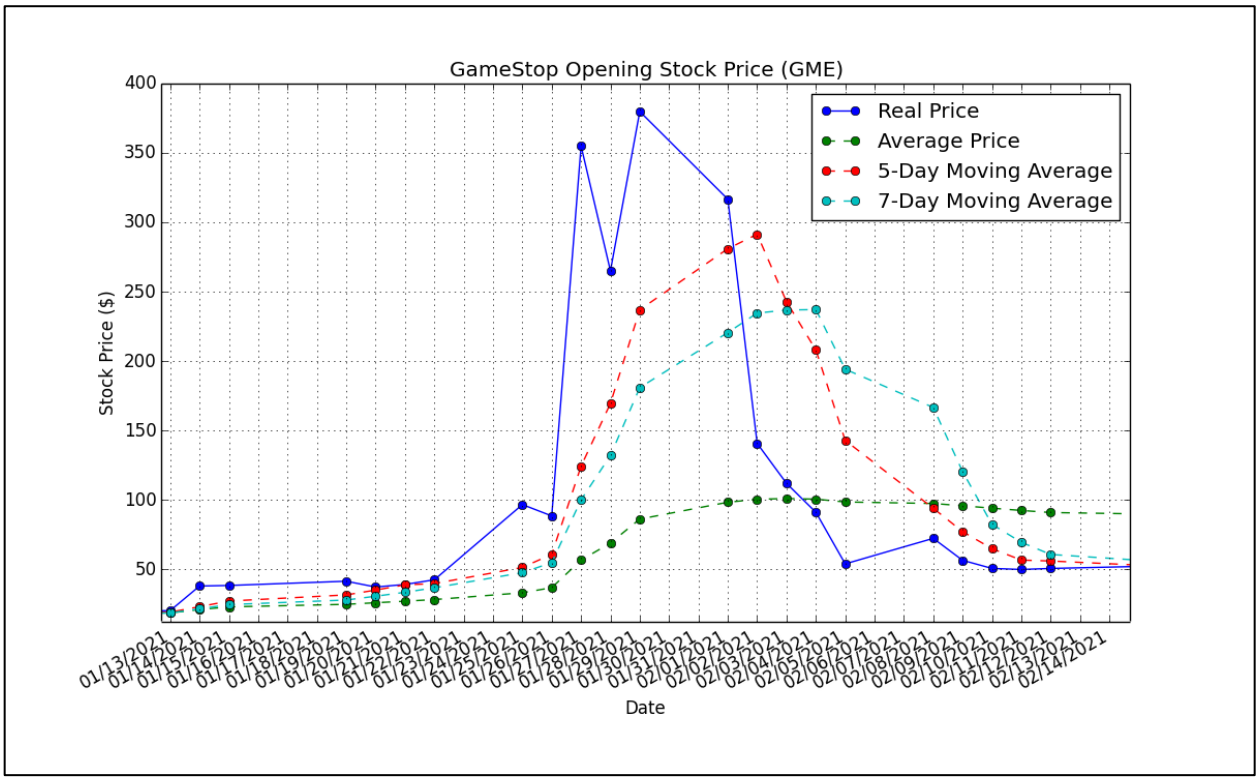


Figure 3: Average Filter vs Moving Average Filter

As you can see with a quick glance at Figure 3, the average filter does not respond to the massive price swing like the moving averages do. Additionally, you can see how the 7-day moving average does not respond as fast as the 5-day moving average to large price changes. The more days (data points) you use for the average, the less it responds to variation. The less days (data points) you use for the average, the more it responds to price changes. So when a moving average is being used, it depends on what the you are looking to gain from the result. If you want a general trend of the price, then maybe the difference between the 5-day or 7-day doesn't matter. But if you want something even more accurate, then you will have to consider using a different number of data points to compute the mean or a different filter type. **The moving**

averages are not perfect but are better than the average filter when the quantities actual value varies over time.

The moving average can be implemented with a recursive approach, Eqn. 4, or a batch approach, Eqn. 3; where the subscript ***k*** is the total number of data points, ***y-bar*** is the mean, ***y*** is the measurement, and ***n*** is the number of data points being averaged. Again, the recursive approach is more difficult to implement but requires less memory. Note, the recursive approach is not preferred for this approach. The moving average is based on a constant number of measurements that will be averaged. So, it is much simpler to maintain the last 5 measurements and average those then it is implementing the recursive approach and save that little bit of memory. Each application is different but in general, the batch approach is easier for the moving average.

$$\bar{y}_k = \frac{y_{k-n+1} + y_{k-n+2} + \cdots + y_k}{n} \tag{3}$$

$$\bar{y}_k = \bar{y}_{k-1} + \frac{y_k - y_{k-n}}{n} \tag{4}$$

When does the moving average not perform well? In cases of high variation, the filter will perform well or unwell based on the number of data points being averaged. The mean values will most likely lag because each of the data points is equally weighted even though the most recent data point is the most accurate. Figure 3 shows this lagging. Depending on your use case, this may or may not be acceptable.

So how can we do better? What if we had an algorithm that weighted the most recent data point higher than the previous data point, and that data point higher than the one before it? This sounds like it will work, let us look at the low-pass filter or as its more commonly known, the exponential moving average filter.

Exponential Moving Average Filter

The exponential moving average filter is based on the fact that an alpha value is defined. The alpha value serves as a weight that determines how much of the new measurement information should contribute to the current mean value. Eqn 5 shows the exponential moving average filter equation; where the subscript ***k*** is the k-th data point being processes, ***y-bar*** is the mean, ***y*** is the measurement, and $\boldsymbol{\alpha}$ is the weight. As you can see, as the alpha value approaches one, the measurement contributes less to the new mean value. Conversely, as alpha approaches zero, the measurement contributes much more to the new mean value.

The alpha value is not something that you will know ahead of time. You will have to evaluate multiple values to see how well it performs based on your data set. The two performance goals that you are trying to achieve are: removing the noise and tracking variation. **These are competing requirements and over optimizing for one will most likely hurt the other, so finding that balance that meets your overall goals for your project is what will determine**

the final alpha value. Figure 4 builds upon Figure 3 and shows how the exponential moving average filter results compare to the moving average.

$$\bar{y}_k = \alpha \bar{y}_{k-1} + (1-\alpha) y_k \tag{5}$$

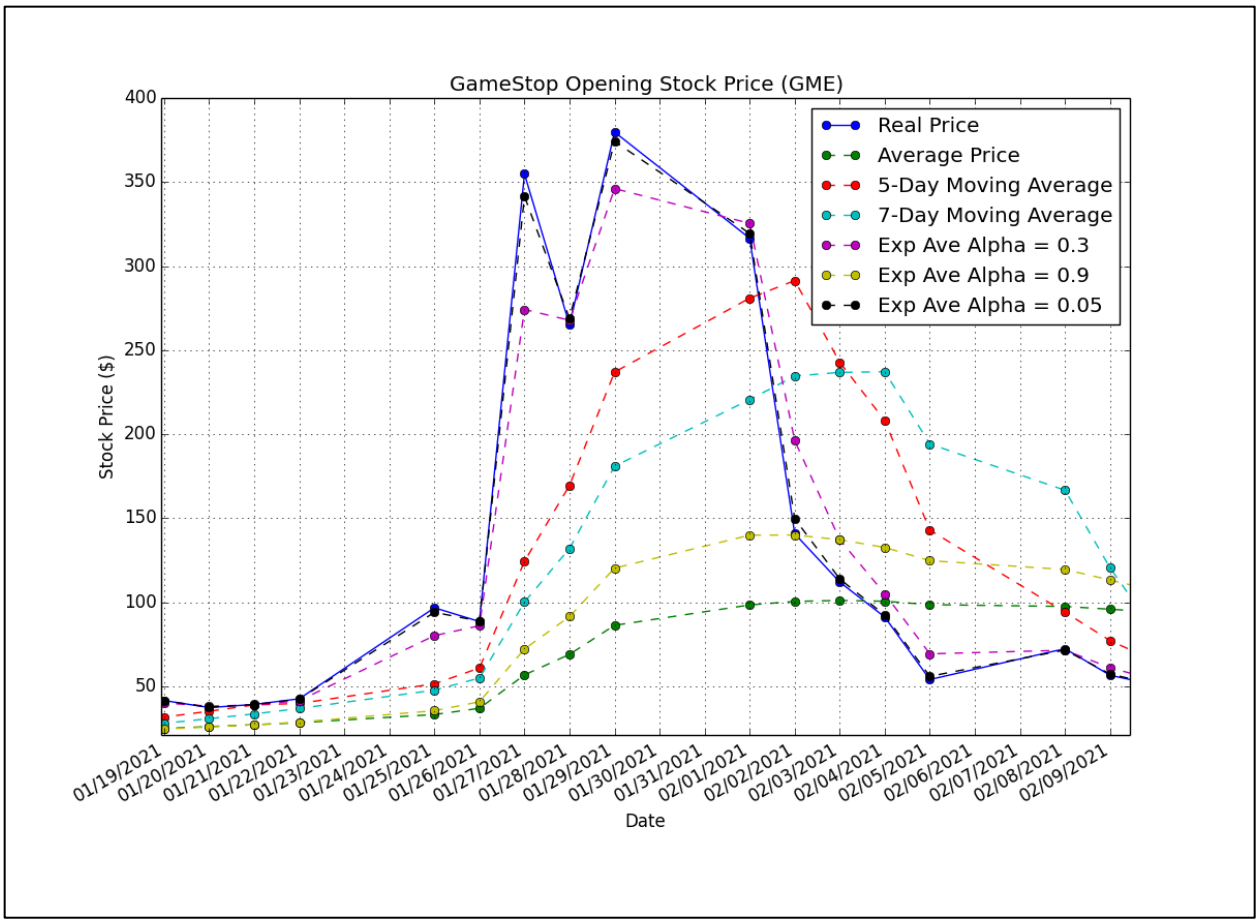


Figure 4: Exponential Weighted Average

In Figure 4, it can be seen that the alpha value of 0.9 does not track variation well and that an alpha value of 0.05 does not remove any noise. An alpha value of 0.3 tracks well and removes noise. Again, depending on your use case, this may or may not be good enough.

Can we do better than this? The answer is yes, what if instead of weighting every new measurement the same way, i.e. alpha value, a weight is computed for each measurement. This is the approach the Kalman Filter takes. **The Kalman Filter has a very similar estimation approach as the exponential moving average filter <u>but the weighting of measurements and the previous estimate is determined by the KalmanGain rather than by a predetermined alpha value.</u>**

Kalman Filter

As stated in the last paragraph, the Kalman Filter has a very similar estimation approach as the exponential moving average. I know that's hard to believe because the Kalman Filter algorithm has multiple steps and is in matrix form but if you follow me through this explanation, you will realize that these two approaches aren't all that different.

For starters, let us take a look at the Kalman Filter algorithm as a whole. Figure 5 below shows a common form of the algorithm. And I know, this is very intimidating but give me chance to explain.

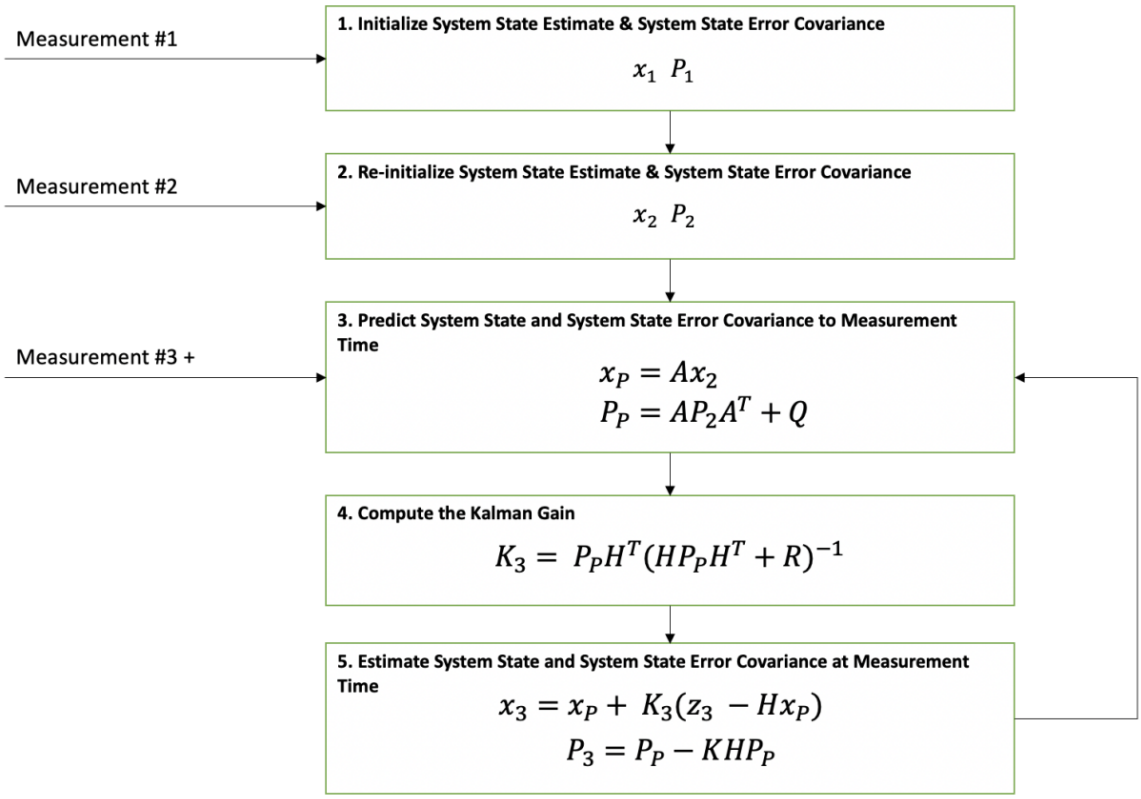


Figure 5: Kalman Filter Overview

Let us start by noting that the Kalman Filter includes a covariance matrix ,***P***, along with its estimate, ***x***. From this point forward I will use the term estimate in place of the mean for the Kalman Filter. The covariance matrix describes the error or uncertainty of the estimate. To start with, we could ignore all of the equations for the covariance matrix, ***P***. We will learn about that in later chapters.

Next let us look at Steps 1 and 2 in Figure 5, these steps are simply initializing the estimate with the input measurement. Steps 3 and 4 compute the Kalman Gain, ***K***, for the input measurement. You can think of the Kalman Gain as the alpha weight from the exponential moving average

filter. The Kalman Gain is used in Step 5 to weight the new measurements contribution to the existing estimate held by the filter. Speaking of Step 5, take a look at the estimate equation in Step 5. Does that look familiar? It should, it is almost identical to the exponential moving average equation. Let us look at these two equations more closely.

Exponential Moving Average Equations

The equation for the exponentially weighted average filter is expressed as:

$$\bar{y}_k = \alpha\bar{y}_{k-1} + (1-\alpha)y_k \tag{6}$$

Where ***y-bar*** is the mean value, the subscript ***k*** is the k-th data point being processed, and $\boldsymbol{\alpha}$ is the predetermined weight. Take note that the range of values for alpha are $0 < \boldsymbol{\alpha} < 1$ and that as alpha gets closer to zero, the new measurement information will contribute more to the new mean value. Conversely, as the alpha value gets closer to 1, the new measurement information is filtered out.

If you look at the first four measurements as they passed through this recursive process, you will see the following four equations.

$$\bar{y}_1 = y_1 \tag{7}$$

$$\bar{y}_2 = \alpha\bar{y}_1 + (1-\alpha)y_2 \tag{8}$$

$$\bar{y}_3 = \alpha\bar{y}_2 + (1-\alpha)y_3 \tag{9}$$

$$\bar{y}_4 = \alpha\bar{y}_3 + (1-\alpha)y_4 \tag{10}$$

If you substitute Eqns. 7, 8 and 9 into Eqn. 10, then the mean value computed by the exponentially weighted average will look like Eqn. 11.

$$\bar{y}_4 = y_1\alpha^3 + y_2\alpha^2 - y_2\alpha^3 + y_3\alpha - y_3\alpha^2 + y_4 - y_4\alpha \tag{11}$$

Eqn. 11 can be rearranged to look like Eqn. 12. You can see that Eqn. 12 uses the alpha weight to influence the measurements contribution to the mean. You can see that the alpha weight has an increasing exponent value along with a decreasing measurement number that it is the coefficient for.

$$\bar{y}_4 = y_4 + \alpha(y_3 - y_4) + \alpha^2(y_2 - y_3) + \alpha^3(y_1 - y_2) \quad (12)$$

When the alpha exponent increases, the resulting value decreases exponentially. This is shown in Table 1 and Figure 6 below where the alpha value from above (α = 0.3) is used for an example. So taking another look at Eqn. 12, you can see that the mean value is computed by weighting the most recent data points greater than the previous ones. This allows the exponentially weighted filter to track variation well when sharp variations occur, if the alpha value is set correctly.

Exponential Alpha Values					
α	α^2	α^3	α^4	α^5	α^6
0.3	0.09	0.0081	0.000729	0.0002187	0.00006561

Table 1: Exponential Alpha Values

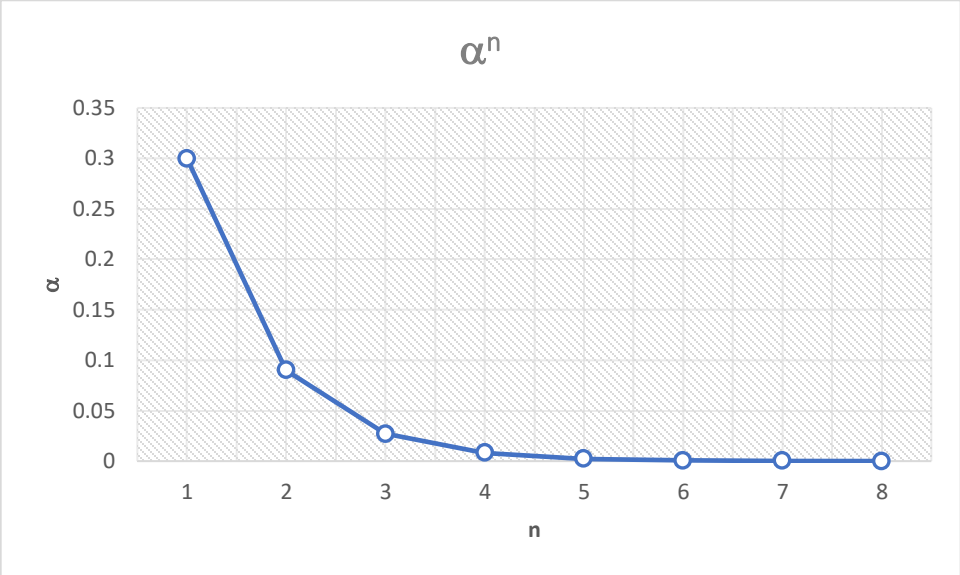


Figure 6: Exponential Alpha Values

Kalman Filter Equations

The Kalman Filter uses a very similar approach to computing the mean except it uses a Kalman Gain value that varies on each update in place of the static alpha value. Let us consider the first four measurements passing through the Kalman Filter as we did above with the exponential moving average filter.

Let us look at the estimate equation from Step 1 and 2 in Figure 5. If you notice there are no equations because those are initialization and reinitialization steps. To simplify this example, we'll assume that reinitialization is not necessary so Step 1 and 2 can be described with Eqn. 13.

$$\bar{x}_1 = z_1 \quad (13)$$

Where ***x-bar*** is the estimate and z is the measurement. Again, the subscript denotes the measurement number and consequently the estimate number.

For the purposes of this exercise we are going to ignore Steps 3 and 4 and assume that a Kalman Gain, ***K***, was computed during those steps.

Step 5 occurs on the second measurement update. And the equation we want to focus on is the estimate, here it is.

$$\bar{x}_2 = \bar{x}_P + K_2(z_2 - H\bar{x}_P) \tag{14}$$

Where ***x-bar*** is the state estimate and the subscript ***P*** indicates the value was the result of the prediction step (Step 3). For the purposes of this exercise, we can assume that ***x-bar$_P$*** is really ***x-bar$_1$***. Additionally, ***H*** is the state-to-measurement transition matrix. We will assume that the measurement and estimate are in the same space so that could be removed because it is as if we multiplied ***x-bar$_P$*** by 1. If we apply those assumptions, we get the following Eqn. 15.

$$\bar{x}_2 = K_2 z_2 + (1 - K_2)\bar{x}_1 \tag{15}$$

Look at Eqn. 15 above. When the predicted estimate is replaced and the equation is rearranged, it looks very similar to the exponential moving average Eqn. 6. Making the same assumptions, you could see that measurements 3 and 4 would go through that estimation equation in the Kalman Filter. If those equations were written out and then substituted back in to the estimation equation for measurement 4, you would come up with Eqn. 16. Where the omega, **ω**, values represent constant values as the result of multiple Kalman Gains being added, subtracted or multiplied. See Eqns. 17, 18, and 19.

You could now see that Eqn. 16 from the Kalman Filter and Eqn. 12. from the exponential moving average are very similar, the difference lies in how these processes weight the incoming measurements information.

$$\bar{x}_4 = z_1 + \omega_1(z_2 - z_1) + \omega_2(z_3 - z_1) + \omega_3(z_4 - z_1) \tag{16}$$

$$\omega_1 = K_2K_3 - K_2 + K_2K_4 - K_2K_3K_4 \tag{17}$$

$$\omega_2 = K_3 - K_3K_4 \tag{18}$$

$$\omega_3 = K_4 \tag{19}$$

With all of that said, hopefully you can see that the exponentially weighted average and the Kalman Filter are closely related and that the primary difference between the two lies in the fact that the Kalman Filter provides a way to compute the weight for each measurement contribution to the estimate where the exponentially weighted average has a predetermined weight. **Other key differences include the Kalman Filter outputting an uncertainty matrix that is extremely valuable but not always necessary and its ability to estimate values of unobservable quantities.**

Hopefully at this point you have seen that the Kalman Filter is not as scary as it seems. I will take you through a deeper dive of the algorithm in Chapter 2 and show you how the weights are computed. From there we will look at examples to see how it is implemented.

Chapter 1 Key Points

- Measurements are not perfect and have errors associated with them. Noise error is the random standard normal error associated with the measurement that could be from many sources e.g. human error. Bias error is a systematic error that is persistent throughout all measurements could be from many different sources e.g. uncalibrated instruments.
- Filters are used to remove noise and track variation of a parameters value. Popular filters include the average filter, the moving average filter, and the exponential moving average filter.
- The Kalman Filter processes data in a very similar way as the exponential moving average filter except it computes its measurement weights for each new measurement rather than using the same weight for all measurements. (Remember that even though the Kalman Filter has matrices, more steps, and more variables – we are doing very similar math to achieve the output!)
- The Kalman Filter can provide the same type of filtering as the others discussed in this chapter but the Kalman Filter can do more.
    - **The Kalman Filter has the ability to provide estimates for parameters that are not able to be observed.**
    - **The Kalman Filter outputs a state estimate along with its covariance matrix, thus providing a probability distribution for the error at that moment in time.**

Chapter 2: The Kalman Filter Explained Simply

In Chapter 1, we discussed the range of filter options you have when you want to remove the noise and track the variation of a data set.

You saw that the **average filter** does not track variation well when the measured quantity is changing over time. You saw that the **moving average** removes noise and tracks variation but it does not track variation well when large variations occur over a small set of measurements, in other words, the filter mean lags behind. You saw that the **exponentially weighted average** worked well to remove noise and track variation when the alpha value was 0.3 for this data set. In these examples, it's important to note that these filters are removing noise and tracking variation of a measured quantity.

So why do we need the Kalman Filter if we have all of these other options? In some cases, these filters will be good enough; in others, they will not. You should use the Kalman Filter when you need to remove noise and track variation of a measured quantity and **quantities that are not measured.** This is a very important distinction and the reason the Kalman Filter is used in so many applications. Again, it provides a way for you to estimate a system variable that you are not measuring or observing explicitly and it provides the quantities probability distribution.

Kalman Filter Algorithm Overview

I will show the same diagram from Chapter 1 which outlines the Kalman Filter algorithm step by step. This may look different from some of the other schematics you have seen but I can assure you it's the same.

At a high level, you'll see that the Kalman Filter algorithm has 5 steps.

(1) Initialization
(2) Reinitialization
(3) Prediction
(4) Compute the Kalman Gain
(5) Estimation

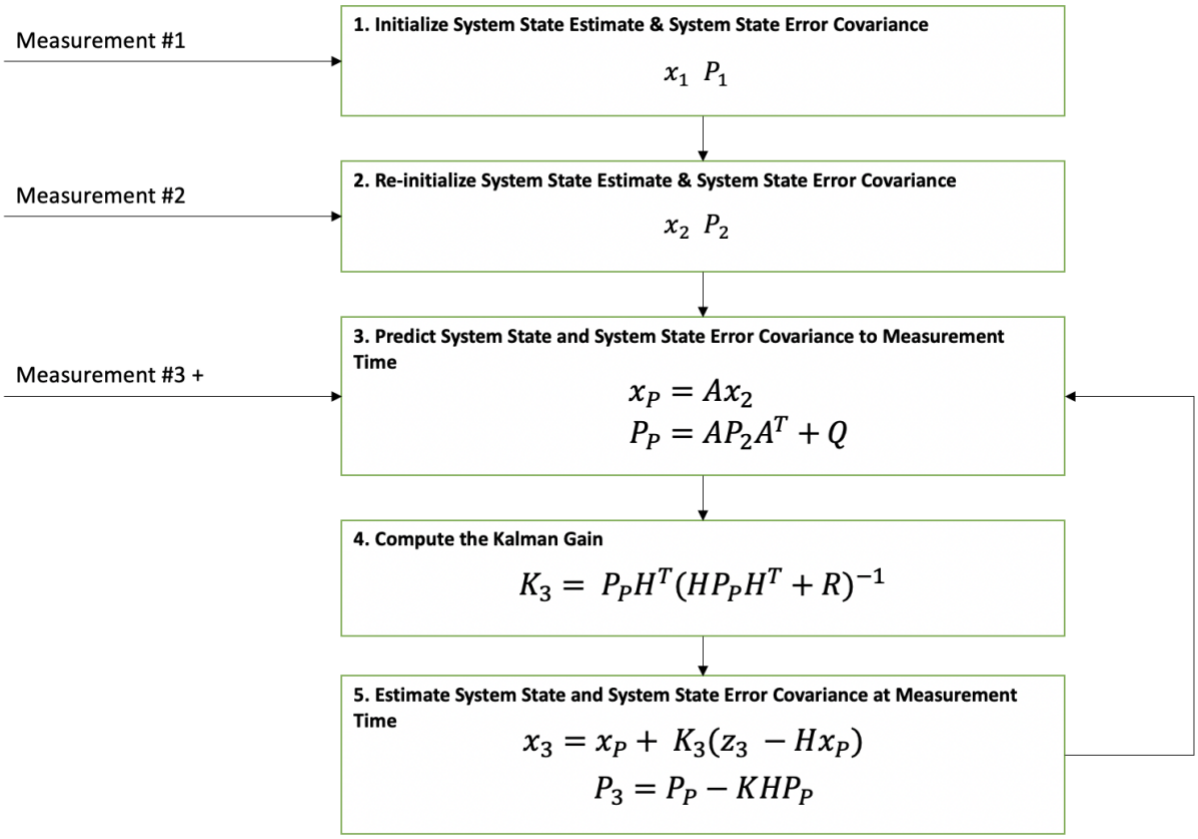


Figure 7: Kalman Filter Overview

The process diagram above shows the Kalman Filter algorithm step by step. I know those equations are intimidating but I assure you this will all make sense by the time you finish reading this chapter. Let's look at this process one step at a time. For your reference, here is a table of definitions that will be referred to throughout the rest of the book.

X	State Estimate Variable	n x 1 column vector	Output
P	State Covariance Matrix	n x n matrix	Output
Z	Measurement	m x 1 column vector	Input
A	State Transition Matrix	n x n matrix	System Model
H	State-to-Measurement Matrix	m x n matrix	System Model
R	Measurement Covariance Matrix	m x m matrix	Input
Q	Process Noise Covariance Matrix	n x n matrix	System Model
K	Kalman Gain	n x m	Internal

The table above identifies the variables used in the algorithm. Each variable listed has a structure type and category.

Initialization and Reinitialization

The parameters that are being initialized are $\boldsymbol{x}$ and $\boldsymbol{P}$, the state estimate and the state covariance matrix, respectively. As I stated earlier, these are the output of the filter. And they provide the user with a state estimate (or previously called "mean") for a parameter as well as the associated covariance matrix for that state estimate at that point in time. None of the other filters mentioned in Chapter 1 output a covariance matrix and this is a big deal. The covariance matrix extends the state estimate to be a probability distribution rather than just an estimate (mean). The probability distribution of the estimate gives the user the likelihood and range of values for the state estimate. If you think about the machines that employ Kalman Filters for real-time data processing, it makes sense that the probability distribution is invaluable when trying to prepare for future events e.g. autonomous vehicles predicting neighboring vehicles positions on the road before a maneuver.

So initialization can be very simple or it can be tricky, most of the time it's very simple. If you think about it, when the first measurement is processed, that's the only information you have. You don't have a previous estimate you are updating which means you can't predict anything or compute a Kalman Gain. So in the simplest cases, the state estimate is the input measurement and the state covariance is the input measurement covariance. In more complicated cases where multiple measurements are needed, reinitialization occurs and again no prediction or Kalman Gain computation is needed e.g. initializing a velocity term from position information needs a reinitialization step.

Lastly there will be cases where the state estimate is a function of the measurement. In those cases, a translation may be necessary to initialize the state properly. For most applications, a range of initialization options will be okay. While in other applications, its necessary for the filter to settle as soon as possible. In those applications, the user might have to make some additional considerations during this initialization and reinitialization steps.

Prediction

The next step in the Kalman Filter is prediction. Specifically, when a new measurement is processed, the Kalman Filter routine requires that the previous state estimate and covariance is predicted forward to align in time with the new measurement. The two equations used for predicting the state estimate and the state covariance matrix are the following:

$$x_P = Ax_k \tag{20}$$

$$P_P = AP_kA^T + Q \tag{21}$$

Where ***A*** represents the state transition matrix, ***Q*** represents the process noise matrix, the ***P*** subscript stands for predicted and the ***k*** subscript is the measurement number.

The A Matrix

The *A* matrix, or the state transition matrix is part of the system model and the set of equations that predict the state estimate forward. In order to predict the previous estimate forward in time, you must understand how the system behaves. In some cases, it's a fair assumption to assume that the mean didn't change between the time of the previous estimate and the new measurement. In other cases, where you are estimating the position and velocity of a vehicle, you will want to use equations of motion to predict where the vehicle is in space after a new measurement is received. In this vehicle case, you can assume linear motion and use the previous estimate of position and velocity along with classical equations of motion to predict the previous estimate forward. Eqn. 22 is sample equation for motion that would be part of the system model.

$$p = p_0 + vt \tag{22}$$

Where ***p*** is the final position of the vehicle, p_0 is the previous estimates position of the vehicle, ***v*** is the previous estimates velocity of the vehicle and ***t*** is the difference in time between the previous estimate and the new measurement. This equation can be captured by the state transition matrix, ***A***. You will see in the next chapter what the matrix looks like and how prediction is achieved through matrix multiplication.

The Q Matrix

The other matrix you see in the prediction equations is the process noise matrix, ***Q***. Since the prediction step is based on a system model that is not perfect, the ***Q*** matrix allows you to increase the state covariance matrix for the predicted estimate with the ***Q*** matrix. This may be the part of the system model that the designer plays with the most in order to meet design requirements. The reason for this is because the ***Q*** matrix has a role in computing the Kalman Gain for each step. It may be hard to see now but as we explore the Kalman Gain equation, it will be obvious how the ***Q*** matrix contributes the gains for each measurement. Consider the example from the paragraph above. A vehicles position is predicted forward in time at a constant velocity. Although this prediction is based on good assumptions, it is not perfect. In the real world, the vehicle most likely had some sort of acceleration or deceleration during that time. This is why the **Q** matrix exists. You will use the **Q** matrix to increase the predicted state covariance matrix to account for its potential acceleration or deceleration.

Additional examples in other chapters will show how the ***Q*** matrix is set based on other use cases.

Compute Kalman Gain

As each measurement is processed, the Kalman Filter algorithm determines how much it should contribute to the existing state estimate of the system. It does this by comparing the input measurement error with the current state estimate error. If the measurement is more accurate than the current state estimate predicted forward in time, then the algorithm will weight this new information higher than the predicted state. Alternatively, if the new measurement is less accurate then the predicted state estimate, then the algorithm will weight the new information less than the predicted state estimate. And this makes sense, right? We want the filter to update the estimate when accurate new information is available but we don't want to update the state with inaccurate new information.

Okay so now that we understand that the Kalman Gain is used to weight new measurement information based on what the input error is, let us look at how the Kalman Filter does this with math in a generic way.

First, let us look at the equation and variables:

$$K = P_P H^T (H P_P H^T + R)^{-1} \tag{23}$$

Where **K** is the Kalman Gain, $\mathbf{P_P}$ is the predicted state covariance matrix, **H** is the state to measurement transition matrix, and **R** is the measurement covariance matrix. Since this is a matrix equation, each application could have different matrix sizes. On first glance, this is a very intimidating equation. There are 3 different matrices along with operations like transpose and inverse. But let me assure you, we could break this down so it is a lot simpler to understand.

Let us consider what this would look like if the measurement was one dimensional and the state estimate was one dimensional. Let us say you are measuring the weight of an apple like in Chapter 1. In this example, the following structures would represent the matrices in Eqn. 23.

$$P_P = \sigma^2_{p,weight} \tag{24}$$

$$H = 1 \tag{25}$$

$$R = \sigma^2_{m,weight} \tag{26}$$

With these definitions in Eqns. 24, 25 and 26. Eqn. 23 can be rewritten as the following Eqn. 27:

$$K = \frac{\sigma_{p,weight}^2}{\sigma_{p,weight}^2 + \sigma_{m,weight}^2} \quad (27)$$

So let us take a look at Eqn. 27. It can be seen that if variance of the measurement is small compared to the variance of the predicted estimate, the gain will be closer to 1. Alternatively, if the variance of the measurement is large compared to the variance of the predicted estimate, the gain will trend towards 0.

It will make more sense in the estimation section but as the Kalman Gain gets larger, the measurement is weighted heavier and as the Kalman Gain gets closer to 1, the measurement information is weighted less. By weighted heavier I mean the measurement information is contributing to the new state estimate more than the previous state estimate information. And vice versa. This makes sense, right?

So let us take a step back. On each new measurement, the Kalman Filter computes a gain based on the measurement's accuracy and the predicted estimate's accuracy. If the measurement is more accurate, the filter uses that information to update the state estimate. Otherwise, the inaccurate measurement is filtered out. Hopefully that paragraph was an Aha! moment for you. (if you haven't already had one) If not, reread it!

Now, the case we just looked at was simple because it was one dimensional. Most likely your application will have multiple dimensions. So instead of having a single Kalman Gain value, you will have a matrix of gain values. This gets tricky to think about because your Kalman Gain matrix may be a 4x2 matrix while your state estimate is a 4x1 vector. Other chapters will explore these structures in more detail but for now, let us remember that the output of the Kalman Filter is a state estimate and its state covariance matrix (or probability distribution). And the covariance matrix has cross terms that indicate how correlated or uncorrelated two parameters are. So when the Kalman Gain is used to update the estimate, each parameter could contribute to the other parameter's value based on correlation between the two. And those gains are determined in the Kalman Gain equation.

Estimation

Okay we are finally at the estimation step of the Kalman Filter. I know it seems like we did a lot leading up to this but we really didn't do anything crazy. Let us go over it quickly.

- The **first measurement** was used to initialize the state estimate and state covariance matrix
- The **second measurement** was used to reinitialize the state estimate and state covariance matrix (if necessary for your application)
- The **third measurement** was received and will actually go through the filter process:
    - First the state estimate and state covariance will be predicted forward in time to align with the measurement time e.g. linear motion equation.
    - Second the measurement covariance and state covariance will be used to compute the Kalman Gain for the input measurement.

    - Lastly the predicted state estimate and state covariance will be used with the Kalman Gain to compute the new estimate in a very similar way as the exponential moving average as discussed in Chapter 1.
- The **fourth measurement** and all subsequent measurements will be processed like the third measurement

Okay that was it. We did the prediction and computed the Kalman Gain. Now let us look at the estimation step. Here are the equations:

$$x_k = x_P + K_k(z_3 - Hx_P) \tag{28}$$

$$P_k = P_P - KHP_P \tag{29}$$

Where the ***x*** is the state estimate, ***K*** is the Kalman Gain, ***z*** is the measurement vector, ***H*** is the state-to-measurement transition matrix, ***P*** is the state covariance matrix, subscript ***k*** is the estimate number or measurement number, and subscript ***P*** represents predicted values of the estimate.

Eqns. 28 and 29 are the final result or output of the Kalman Filter. And again, they are the state estimate and the associated state covariance matrix. Together they represent the parameters being estimated values and the probability distribution at that point in time.

Eqn. 29 has not been discussed yet. In other chapters where examples are being examined, it will be easier to see how it works. But for now, let us just understand one thing about it. It can be rewritten as Eqn. 30 below.

$$P_k = (I - KH)P_P \tag{30}$$

Where ***I*** is the identity matrix.

Let us make the same one dimensional assumption as we did earlier. In that case, the identity matrix would be the integer 1 and the state to measurement transition matrix, ***H***, would also be the integer 1. If we rewrite the equation again, it becomes Eqn. 31.

$$P_k = (1 - K)P_P \tag{31}$$

Thinking back to how the Kalman Gain affects the state estimate, we recall that as the gain approaches 1, the measurement is weighted heavier; meaning its more accurate information then the current state estimate. As the gain approaches 0, the measurement gets filtered out and does not contribute to the new state estimate. Based on Eqn. 31, we can see that as the gain approaches 1, the new state covariance gets reduced from its current value i.e. making the state

estimate more accurate. As the gain approaches 0, the new state covariance stays the same i.e. does not get more accurate.

The big takeaway from these equations was discussed in detail but I will reiterate it again.

Each new measurement processed by a Kalman Filter provides an opportunity for the current state estimate and state covariance matrix to be improved. It is the Kalman Filters job to determine whether or not the input measurement information is used or whether it will just be filtered out as noise. The Kalman Filter achieves this by computing the Kalman Gain based on the comparison of the input measurement covariance matrix and the current state estimates covariance matrix. The Kalman Gain is used in the estimation step to compute the new state estimate and state covariance. If this paragraph doesn't make sense, you have missed something along the way. Please reread to ensure were on the same page. (literally and figuratively)

Chapter 2 Key Points

- The Kalman Filter algorithm has been in use across engineers and industries for the past 60 years! It is well understood and well documented. We do not need to verify that it works or dive deep into the math to understand its derivations. We can use it by trusting that the past 60 years confirmed its validity!
- The difference between each application is the system model. That includes the following matrices: state transition matrix, A; state to measurement transition matrix, H; measurement covariance matrix, R; and the process noise matrix, Q.
- Your goal as a designer is to construct a system model as accurate as possible to the real life system dynamics. The closer you get to modeling the real life systems behaviors, the better performance you will see. These are the design knobs you will use to affect performance!
- The state estimate and the state covariance are the final product or output of your filter. Together they represent the probability distribution for the parameters you are estimating. These can be used individually or in combination based on your application.

Chapter 3: Image Object Tracking Tutorial with a Kalman Filter

Most tutorials for the Kalman Filter are difficult to understand because they require advanced math skills to understand how the Kalman Filter is derived. If you have tried to read Rudolf E Kalman's 1960 Kalman Filter paper, you know how confusing this concept can be. But do you need to understand how to derive the Kalman Filter in order to use it?

No. If you want to design and implement a Kalman Filter, you do not need to know how to derive it, you just need to understand how it works.

The truth is, anybody can understand the Kalman Filter if it is explained in small digestible chunks. The goal of this book is to explain the Kalman Filter to anybody!

This chapter will go through the step-by-step process of a Kalman Filter being used to track objects in an image. The output track states can be used for downstream processing such as counting the number of objects in a frame, executing object avoidance algorithms, and other uses.

Symbolic Kalman Filter Tutorial

This tutorial assumes that an object detection algorithm is being used to create measurements based on detections. There are open-source options for this or you can design one from scratch.

The object detection algorithms used for this example will output its measurements in 2D cartesian coordinates, x and y. These measurements will be represented as a 2-by-1 column vector, $\boldsymbol{z}$. The associated variance-covariance matrix for these measurements, $\boldsymbol{R}$, will also be provided by the detection algorithm along with the time tag for when the measurement occurred, $\boldsymbol{t}$. The subscript $\boldsymbol{m}$ denotes the measurement parameters. And the $\boldsymbol{k}$ subscript denotes the k-th measurement being processed.

Measurement	**Measurement Time Stamp**
$z_k = \begin{bmatrix} x_{m_k} \\ y_{m_k} \end{bmatrix}$	$t_k = t_{m_k}$
Measurement Covariance Matrix	
$R_k = \begin{bmatrix} \sigma_{x_m}^2 & \sigma_{xy_m} \\ \sigma_{xy_m} & \sigma_{y_m}^2 \end{bmatrix}$	

This Kalman Filter estimates the objects position and velocity based on the detection measurements. The estimate is represented by a 4-by-1 column vector, $\boldsymbol{x}$. It's associated variance-covariance matrix for the estimate is represented by a 4-by-4 matrix, $\boldsymbol{P}$. Additionally, the state estimate has a time tag denoted as $\boldsymbol{T}$.

State Estimate	State Time Stamp
$x_k = \begin{bmatrix} x \\ y \\ \dot{x} \\ \dot{y} \end{bmatrix}$	T_k
State Covariance Matrix	
$P_k = \begin{bmatrix} \sigma_x^2 & \sigma_{xy} & \sigma_{x\dot{x}} & \sigma_{x\dot{y}} \\ \sigma_{xy} & \sigma_y^2 & \sigma_{y\dot{x}} & \sigma_{y\dot{y}} \\ \sigma_{x\dot{x}} & \sigma_{y\dot{x}} & \sigma_{\dot{x}}^2 & \sigma_{\dot{x}\dot{y}} \\ \sigma_{x\dot{y}} & \sigma_{y\dot{y}} & \sigma_{\dot{x}\dot{y}} & \sigma_{\dot{y}}^2 \end{bmatrix}$	

Step 1: Initialize System State

Initializing the system state of a Kalman Filter varies across applications. In this tutorial, the Kalman Filter initializes the system state with the first measurement.

$$\boldsymbol{x_k, P_k}$$

In this object tracking example, the input measurements contain position only information. The output system state will contain the position and velocity of the object.

When the first measurement is received, the only information known about the object is the position at that point in time. The system state estimate will be set to the input position after the first estimate. The system state error covariance will be set to the first measurement's position accuracy.

Initialize System State in Equations

The following equations represent the input measurements as well as how they are mapped to the initial state estimate during the initialization step.

Measurement	Measurement Time Stamp	Measurement Covariance
$z_1 = \begin{bmatrix} x_{m_1} \\ y_{m_1} \end{bmatrix}$	$t_1 = t_{m_1}$	$R_1 = \begin{bmatrix} \sigma_{x_m}^2 & \sigma_{xy_m} \\ \sigma_{xy_m} & \sigma_{y_m}^2 \end{bmatrix}$
State Estimate		**State Time Stamp**
$x_1 = \begin{bmatrix} x \\ y \\ \dot{x} \\ \dot{y} \end{bmatrix} = \begin{bmatrix} x_{m_1} \\ y_{m_1} \\ 0 \\ 0 \end{bmatrix}$		$T_1 = t_{m_1}$
State Covariance		
$P_1 = \begin{bmatrix} \sigma_x^2 & \sigma_{xy} & \sigma_{x\dot{x}} & \sigma_{x\dot{y}} \\ \sigma_{xy} & \sigma_y^2 & \sigma_{y\dot{x}} & \sigma_{y\dot{y}} \\ \sigma_{x\dot{x}} & \sigma_{y\dot{x}} & \sigma_{\dot{x}}^2 & \sigma_{\dot{x}\dot{y}} \\ \sigma_{x\dot{y}} & \sigma_{y\dot{y}} & \sigma_{\dot{x}\dot{y}} & \sigma_{\dot{y}}^2 \end{bmatrix} = \begin{bmatrix} \sigma_{x_m}^2 & \sigma_{xy_m} & 0 & 0 \\ \sigma_{xy_m} & \sigma_{y_m}^2 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$		

Step 2: Reinitialize System State

In order to properly initialize the system state, a second measurement is used to compute the first velocity values.

$$\boldsymbol{x_k, P_k}$$

Velocity is estimated with a linear approximation. As you most likely recall from high school physics, velocity is equal to the distance traveled divided by the time it took to travel that distance.

$$average\ velocity = \frac{\Delta x}{\Delta t} = \frac{displacement}{change\ in\ time}$$

The updated system state estimate will be the second measurement's position and the computed velocity. The updated system state error covariance will be the second measurement's position accuracy and an approximated velocity accuracy. Note that this velocity accuracy approximation is something that can be tuned and adjusted after running data through your filter. There are different ways of approximating these values so if this doesn't match your approximation, that's okay!

Reinitialize System State in Equations

These equations show the input and output values for this Kalman Filter after receiving the second measurement. Note the velocity variance terms in the state covariance matrix. These values are being set to 10^4. In other words, this value indicates a large uncertainty for the velocity state values. In this example, the units are meters per second.

<table>
<tr><th>Measurement</th><th>Measurement Time Stamp</th><th>Measurement Covariance</th></tr>
<tr><td>$z_2 = \begin{bmatrix} x_{m_2} \\ y_{m_2} \end{bmatrix}$</td><td>$t_2 = t_{m_2}$</td><td>$R_2 = \begin{bmatrix} \sigma_{x_m}^2 & \sigma_{xy_m} \\ \sigma_{xy_m} & \sigma_{y_m}^2 \end{bmatrix}$</td></tr>
<tr><th colspan="2">State Estimate</th><th>State Time Stamp</th></tr>
<tr><td colspan="2" rowspan="2">$x_2 = \begin{bmatrix} x \\ y \\ \dot{x} \\ \dot{y} \end{bmatrix} = \begin{bmatrix} x_{m_2} \\ y_{m_2} \\ \frac{x_{m_2} - x_{m_1}}{\Delta T} \\ \frac{y_{m_2} - y_{m_1}}{\Delta T} \end{bmatrix}$</td><td>$T_2 = t_{m_2}$</td></tr>
<tr><th>Time Delta</th></tr>
<tr><td colspan="2">$\Delta T = T_2 - T_1$</td></tr>
<tr><th colspan="3">State Covariance</th></tr>
<tr><td colspan="3">$P_2 = \begin{bmatrix} \sigma_x^2 & \sigma_{xy} & \sigma_{x\dot{x}} & \sigma_{x\dot{y}} \\ \sigma_{xy} & \sigma_y^2 & \sigma_{y\dot{x}} & \sigma_{y\dot{y}} \\ \sigma_{x\dot{x}} & \sigma_{y\dot{x}} & \sigma_{\dot{x}}^2 & \sigma_{\dot{x}\dot{y}} \\ \sigma_{x\dot{y}} & \sigma_{y\dot{y}} & \sigma_{\dot{x}\dot{y}} & \sigma_{\dot{y}}^2 \end{bmatrix} = \begin{bmatrix} \sigma_{x_m}^2 & \sigma_{xy_m} & 0 & 0 \\ \sigma_{xy_m} & \sigma_{y_m}^2 & 0 & 0 \\ 0 & 0 & 10^4 & 0 \\ 0 & 0 & 0 & 10^4 \end{bmatrix}$</td></tr>
</table>

Quick Note on Initialization

Steps 1 and 2 used the first couple measurements to initialize and re-initialize the system estimate. Each application of the Kalman Filter may do this differently but the goal is to have a system state estimate that can be updated for future measurement with the Kalman Filter equations.

Steps 3 through 5 demonstrate how measurements are filtered in and the state estimate is updated.

Step 3: Predict System State Estimate

When the third measurement is received, the system state estimate is propagated forward in time to align with the measurement. This alignment is done so that the measurement and state estimate can be combined.

$$\boldsymbol{x_p = Ax_{k-1}}$$

$$\boldsymbol{P_P = AP_{k-1}A^T + Q}$$

The system model is used to perform this prediction. In this example, a constant velocity linear motion model is used to approximate the objects position change over a time interval. Note that a constant velocity model does assume zero acceleration. Remember this because it will resurface later.

$$\boldsymbol{p = p_0 + vt}$$

The constant velocity linear motion model is something you may also remember from your high school physics class. The equation states that the position of an object is equal to its initial position plus its displacement over a specified time period assuming a constant velocity.

A state transition matrix represents these equations. This matrix is used to propagate the state estimate and state error covariance matrix appropriately. You may be wondering why the state error covariance matrix is propagated. The reason for this is because when a state estimate is propagated in time, the uncertainty about its state at this future time step is inherently uncertain, so the error covariance grows.

On the Q Matrix

The $\boldsymbol{Q}$ matrix represents process noise for the system model. The system model is an approximation. Throughout the life of a system state, that system model fluctuates in its accuracy. Therefore, the $\boldsymbol{Q}$ matrix is used to represent this uncertainty and adds to the existing noise on the state. For this example, the systems actual accelerations and decelerations contribute to this error.

On the H Matrix

The Kalman Filter uses the state-to-measurement matrix, ***H***, to convert the system state estimate from the state space to the measurement space. For some Kalman Filter applications, this is a matrix of zeros and ones. For other applications that use the Extended Kalman Filter, the ***H*** matrix is populated with differential equations.

In this tutorial, the ***H*** matrix is a simple matrix that is set up to reduce the state estimate and error covariance to position only values rather than position and velocity.

Predict System State in Equations

Measurement	Measurement Time Stamp	Measurement Covariance
$z_3 = \begin{bmatrix} x_{m_3} \\ y_{m_3} \end{bmatrix}$	$t_3 = t_{m_3}$	$R_3 = \begin{bmatrix} \sigma_{x_m}^2 & \sigma_{xy_m} \\ \sigma_{xy_m} & \sigma_{y_m}^2 \end{bmatrix}$

Predicted State Estimate	State Time Stamp
$x_p = Ax_2 = \begin{bmatrix} x + \dot{x}\Delta T \\ y + \dot{y}\Delta T \\ \dot{x} \\ \dot{y} \end{bmatrix}$	$T_3 = t_{m_3}$
	Time Delta
	$\Delta T = T_3 - T_2$

State Transition Matrix	Process Noise Matrix
$A = \begin{bmatrix} 1 & 0 & \Delta T & 0 \\ 0 & 1 & 0 & \Delta T \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	$Q = \begin{bmatrix} 10 & 0 & 0 & 0 \\ 0 & 10 & 0 & 0 \\ 0 & 0 & 25 & 0 \\ 0 & 0 & 0 & 25 \end{bmatrix}$

Predicted State Covariance

$$P_P = AP_2A^T + Q$$

$$= \begin{bmatrix} 1 & 0 & \Delta T & 0 \\ 0 & 1 & 0 & \Delta T \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \sigma_{x_m}^2 & \sigma_{xy_m} & 0 & 0 \\ \sigma_{xy_m} & \sigma_{y_m}^2 & 0 & 0 \\ 0 & 0 & 10^4 & 0 \\ 0 & 0 & 0 & 10^4 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ \Delta T & 0 & 1 & 0 \\ 0 & \Delta T & 0 & 1 \end{bmatrix}$$

$$+ \begin{bmatrix} 10 & 0 & 0 & 0 \\ 0 & 10 & 0 & 0 \\ 0 & 0 & 25 & 0 \\ 0 & 0 & 0 & 25 \end{bmatrix}$$

$$P_P = AP_2A^T + Q$$

$$= \begin{bmatrix} \sigma_x^2 + 10000 \cdot \Delta T^2 + 10 & \sigma_{xy} & 10000 \cdot \Delta T & 0 \\ \sigma_{xy} & \sigma_y^2 + 10000 \cdot \Delta T^2 + 10 & 0 & 10000 \cdot \Delta T \\ 10000 \cdot \Delta T & 0 & 10000 + 25 & 0 \\ 0 & 10000 \cdot \Delta T & 0 & 10000 + 25 \end{bmatrix}$$

Step 4: Compute the Kalman Gain

The Kalman Filter computes a Kalman Gain for each new measurement that determines how much the input measurement will influence the system state estimate. In other words, when a really noisy measurement comes in to update the system state, the Kalman Gain will trust its current state estimate more than this new inaccurate information.

This concept is the root of the Kalman Filter algorithm and why it works. It can recognize how to properly weight its current estimate and the new measurement information to form an optimal estimate.

$$\boldsymbol{K_k = P_P H^T (H P_P H^T + R_k)^{-1}}$$

Compute the Kalman Gain in Equations

Measurement	**Measurement Time Stamp**	**Measurement Covariance**
$z_3 = \begin{bmatrix} x_{m_3} \\ y_{m_3} \end{bmatrix}$	$t_3 = t_{m_3}$	$R_3 = \begin{bmatrix} \sigma_{x_m}^2 & \sigma_{xy_m} \\ \sigma_{xy_m} & \sigma_{y_m}^2 \end{bmatrix}$
State to Measurement Transition Matrix		**Predicted State Covariance**
$H = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$		$P_P = \begin{bmatrix} \sigma_x^2 & \sigma_{xy} & \sigma_{x\dot{x}} & \sigma_{x\dot{y}} \\ \sigma_{xy} & \sigma_y^2 & \sigma_{y\dot{x}} & \sigma_{y\dot{y}} \\ \sigma_{x\dot{x}} & \sigma_{y\dot{x}} & \sigma_{\dot{x}}^2 & \sigma_{\dot{x}\dot{y}} \\ \sigma_{x\dot{y}} & \sigma_{y\dot{y}} & \sigma_{\dot{x}\dot{y}} & \sigma_{\dot{y}}^2 \end{bmatrix}$
Kalman Gain		

$$K_3 = P_P H^T (H P_P H^T + R_3)^{-1}$$

$$K_3 = \begin{bmatrix} \sigma_x^2 & \sigma_{xy} & \sigma_{x\dot{x}} & \sigma_{x\dot{y}} \\ \sigma_{xy} & \sigma_y^2 & \sigma_{y\dot{x}} & \sigma_{y\dot{y}} \\ \sigma_{x\dot{x}} & \sigma_{y\dot{x}} & \sigma_{\dot{x}}^2 & \sigma_{\dot{x}\dot{y}} \\ \sigma_{x\dot{y}} & \sigma_{y\dot{y}} & \sigma_{\dot{x}\dot{y}} & \sigma_{\dot{y}}^2 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \left(\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} \sigma_x^2 & \sigma_{xy} & \sigma_{x\dot{x}} & \sigma_{x\dot{y}} \\ \sigma_{xy} & \sigma_y^2 & \sigma_{y\dot{x}} & \sigma_{y\dot{y}} \\ \sigma_{x\dot{x}} & \sigma_{y\dot{x}} & \sigma_{\dot{x}}^2 & \sigma_{\dot{x}\dot{y}} \\ \sigma_{x\dot{y}} & \sigma_{y\dot{y}} & \sigma_{\dot{x}\dot{y}} & \sigma_{\dot{y}}^2 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} + \begin{bmatrix} \sigma_{x_m}^2 & \sigma_{xy_m} \\ \sigma_{xy_m} & \sigma_{y_m}^2 \end{bmatrix} \right)^{-1}$$

$$K_3 = \begin{bmatrix} \sigma_x^2 & \sigma_{xy} \\ \sigma_{xy} & \sigma_y^2 \\ \sigma_{x\dot{x}} & \sigma_{y\dot{x}} \\ \sigma_{x\dot{y}} & \sigma_{y\dot{y}} \end{bmatrix} \left(\begin{bmatrix} \sigma_x^2 + \sigma_{x_m}^2 & \sigma_{xy} + \sigma_{xy_m} \\ \sigma_{xy} + \sigma_{xy_m} & \sigma_y^2 + \sigma_{y_m}^2 \end{bmatrix} \right)^{-1}$$

$$K_3 = \begin{bmatrix} \sigma_x^2 & \sigma_{xy} \\ \sigma_{xy} & \sigma_y^2 \\ \sigma_{x\dot{x}} & \sigma_{y\dot{x}} \\ \sigma_{x\dot{y}} & \sigma_{y\dot{y}} \end{bmatrix} \begin{bmatrix} \frac{\sigma_y^2 + \sigma_{y_m}^2}{(\sigma_x^2 + \sigma_{x_m}^2)(\sigma_y^2 + \sigma_{y_m}^2) - (\sigma_{xy} + \sigma_{xy_m})^2} & \frac{-(\sigma_{xy} + \sigma_{xy_m})}{(\sigma_x^2 + \sigma_{x_m}^2)(\sigma_y^2 + \sigma_{y_m}^2) - (\sigma_{xy} + \sigma_{xy_m})^2} \\ \frac{-(\sigma_{xy} + \sigma_{xy_m})}{(\sigma_x^2 + \sigma_{x_m}^2)(\sigma_y^2 + \sigma_{y_m}^2) - (\sigma_{xy} + \sigma_{xy_m})^2} & \frac{\sigma_x^2 + \sigma_{x_m}^2}{(\sigma_x^2 + \sigma_{x_m}^2)(\sigma_y^2 + \sigma_{y_m}^2) - (\sigma_{xy} + \sigma_{xy_m})^2} \end{bmatrix}$$

Step 5: Estimate System State and System State Error Covariance Matrix

The Kalman Filter uses the Kalman Gain to estimate the system state and error covariance matrix for the time of the input measurement. After the Kalman Gain is computed, it is used to weight the measurement appropriately in two computations.

The first computation is the new system state estimate. The second computation is the system state error covariance.

$$\boldsymbol{x_k = x_P + K_k(z_k - Hx_P)}$$

$$\boldsymbol{P_k = KHP_P}$$

The state estimate computed above is the only state history the Kalman Filter retains. As a result, Kalman Filters can be implemented on machines with low memory restrictions.

Compute State Estimate in Equations

Measurement	**Measurement Time Stamp**	**Measurement Covariance**
$z_3 = \begin{bmatrix} x_{m_3} \\ y_{m_3} \end{bmatrix}$	$t_3 = t_{m_3}$	$R_3 = \begin{bmatrix} \sigma_{x_m}^2 & \sigma_{xy_m} \\ \sigma_{xy_m} & \sigma_{y_m}^2 \end{bmatrix}$
Predicted State Estimate		**State Time Stamp**
$x_p = Ax_2 = \begin{bmatrix} x + \dot{x}\Delta T \\ y + \dot{y}\Delta T \\ \dot{x} \\ \dot{y} \end{bmatrix}$		$T_3 = t_{m_3}$
		Time Delta
		$\Delta T = T_3 - T_2$

Predicted State Covariance Matrix	**State to Measurement Transition Matrix**
$P_P = \begin{bmatrix} \sigma_x^2 & \sigma_{xy} & \sigma_{x\dot{x}} & \sigma_{x\dot{y}} \\ \sigma_{xy} & \sigma_y^2 & \sigma_{y\dot{x}} & \sigma_{y\dot{y}} \\ \sigma_{x\dot{x}} & \sigma_{y\dot{x}} & \sigma_{\dot{x}}^2 & \sigma_{\dot{x}\dot{y}} \\ \sigma_{x\dot{y}} & \sigma_{y\dot{y}} & \sigma_{\dot{x}\dot{y}} & \sigma_{\dot{y}}^2 \end{bmatrix}$	$H = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$

State Estimate

$$x_3 = x_P + K_k(z_k - Hx_P)$$

$$x_3 = \begin{bmatrix} x + \dot{x}\Delta T \\ y + \dot{y}\Delta T \\ \dot{x} \\ \dot{y} \end{bmatrix} + K_3\left(\begin{bmatrix} x_{m_3} \\ y_{m_3} \end{bmatrix} - \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}\begin{bmatrix} x + \dot{x}\Delta T \\ y + \dot{y}\Delta T \\ \dot{x} \\ \dot{y} \end{bmatrix}\right)$$

$$x_3 = \begin{bmatrix} x + \dot{x}\Delta T \\ y + \dot{y}\Delta T \\ \dot{x} \\ \dot{y} \end{bmatrix} + K_3\left(\begin{bmatrix} x_{m_3} - (x + \dot{x}\Delta T) \\ y_{m_3} - (y + \dot{y}\Delta T) \end{bmatrix}\right)$$

State Covariance Estimate

$$P_3 = KHP_P$$

$$P_3 = K_3\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}\begin{bmatrix} \sigma_x^2 & \sigma_{xy} & \sigma_{x\dot{x}} & \sigma_{x\dot{y}} \\ \sigma_{xy} & \sigma_y^2 & \sigma_{y\dot{x}} & \sigma_{y\dot{y}} \\ \sigma_{x\dot{x}} & \sigma_{y\dot{x}} & \sigma_{\dot{x}}^2 & \sigma_{\dot{x}\dot{y}} \\ \sigma_{x\dot{y}} & \sigma_{y\dot{y}} & \sigma_{\dot{x}\dot{y}} & \sigma_{\dot{y}}^2 \end{bmatrix}$$

$$P_3 = K_3\begin{bmatrix} \sigma_x^2 & \sigma_{xy} & \sigma_{x\dot{x}} & \sigma_{x\dot{y}} \\ \sigma_{xy} & \sigma_y^2 & \sigma_{y\dot{x}} & \sigma_{y\dot{y}} \end{bmatrix}$$

Chapter 3 Key Points

Chapter 3 went over the step-by-step process of filtering object detection measurements from images. One of the key takeaways from this chapter is the high level understanding of the algorithm itself. Even though this tutorial is for object detection in images, the same steps and equations will be used in other examples. To recap, they are the following:

- Initialize the System State
- Re-initialize the System State (not always necessary)
- Predict the System State to Align with the Current Measurement Timestamp
- Compute the Kalman Gain for the Current Measurement Timestamp
- Estimate the System State for the Current Measurement Timestamp

Understanding these steps conceptually will aid in your implementation and adjusting the performance as you improve your Kalman Filters.

Chapter 4: Kalman Filter Basic Example

A basic example of using a Kalman Filter is to filter out the noise from a data set. For this example, we will use the weekly closing price of Bitcoin as the data set. As you may or may not know, this cryptocurrency price is very volatile which makes it an interesting example to work on.

To start, I downloaded the weekly Bitcoin closing prices for 2021. (this information is widely available on the internet) Since I don't know anything about this data set, I pulled the data into Microsoft Excel and did some basic statistical analysis on it to characterize the data.

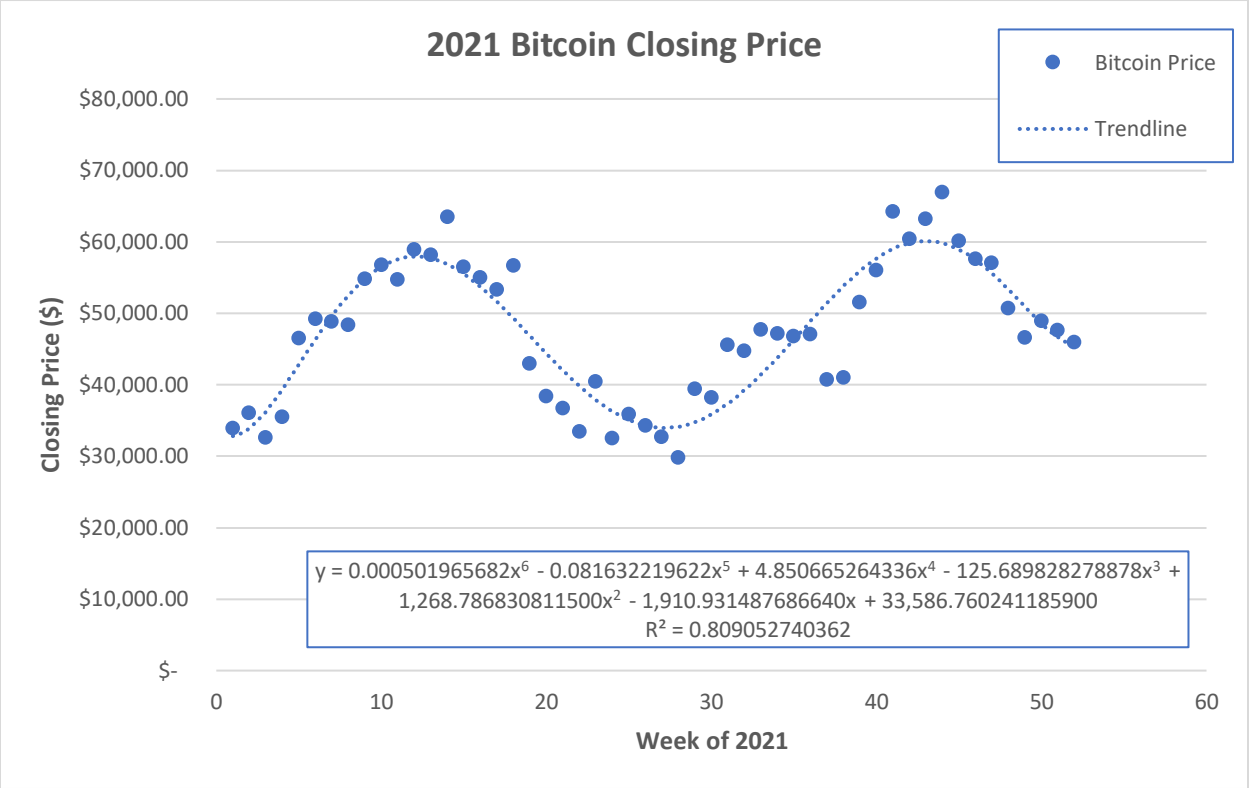


Figure 8: Bitcoin Closing Price

Figure 8, above, shows the Bitcoin weekly closing price for 2021. Additionally, it includes a trendline I created with Microsoft Excel. This trendline is a 6th order polynomial fit (see the equation on the chart). The reason I did this is because I wanted to come up with the "actual" weekly closing price. The "actual" price is what I am going to consider the true price at that point in time, while the measured closing price represents the "actual" price, plus some market noise.

$$z_{meas} = actual + market\ noise$$

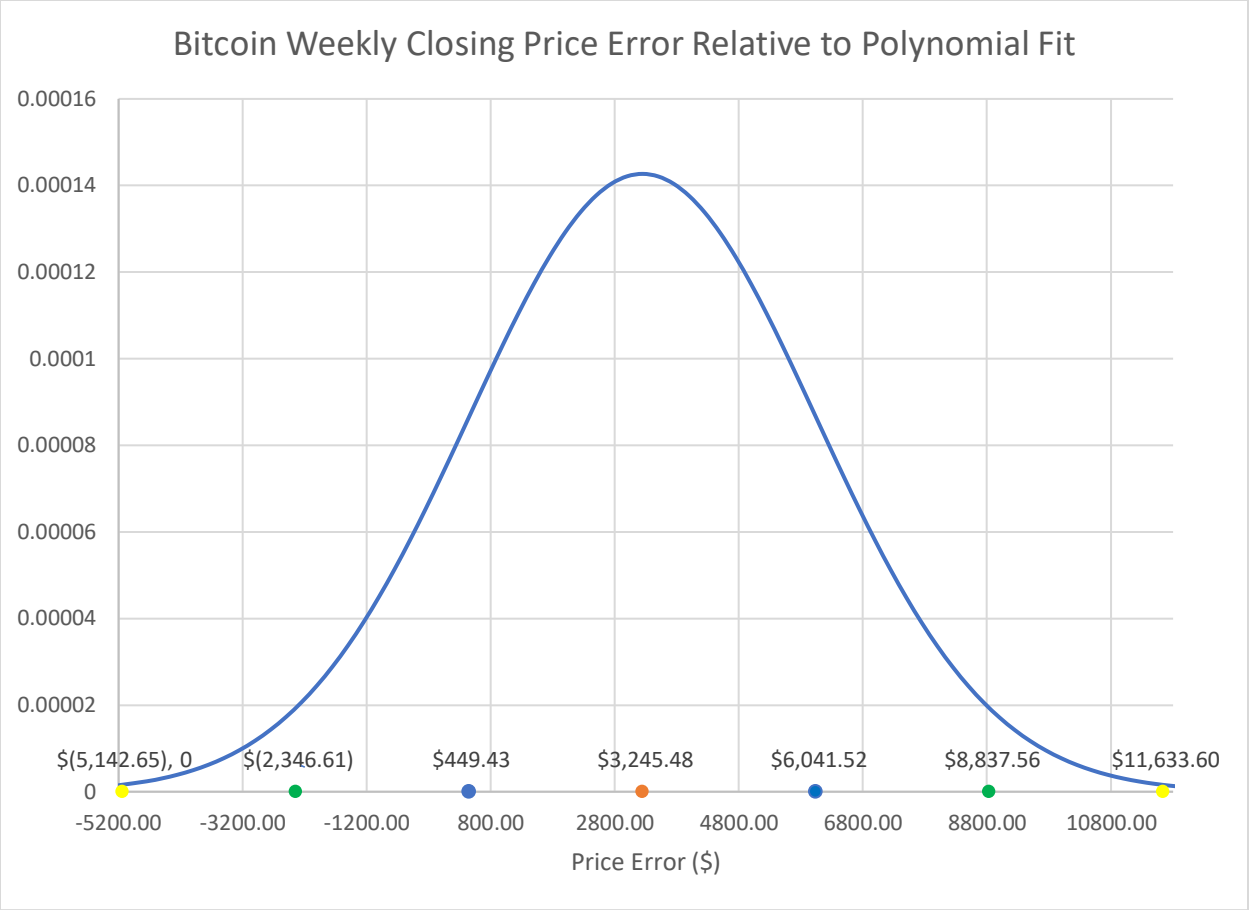


Figure 9: Price Error Distribution

If we assume that the trendline represents the actual price, then we could identify what the error is between the measured closing price and the actual price. If we assume that this error is normally distributed, we can compute the mean (average) and the standard deviation of the error and plot it as a bell curve like Figure 9. As you can see in Figure 9, the mean and standard deviation are equal to the values in Table X, below.

Error Analysis of Measured Weekly Closing Price		
Average	$	3,245.48
Standard Deviation	$	2,796.04

Table 2: Error Analysis

Without having any form of accuracy information on these price numbers. We have to come up with them ourselves. There are different ways of approaching this. But the analysis I have showed you in this section is one reasonable way to go about it.

Now that we have analyzed the measurement price data, we can start designing the system model.

Designing the System Model

When I say "designing the system model". I am really talking about how to define the following variables: A, Q, R, and H for our filter.

Because this is a simple example, all of these variables will not be matrices like in some of our previous examples.

System Model Parameter	Variable Symbol	Value	Reasoning
State Transition Matrix	***A***	1	The state transition matrix is used in the prediction step. Our best assumption for Bitcoin price is for it to stay the same as it was the week before. So we set it to 1 to keep the predicted the same as the previous.
State-to-Measurement Matrix	***H***	1	Since our data sets measurement (i.e. Bitcoin weekly price) is in the same units and space as our filter estimate output, there is no translation that needs to occur. So we set this value to 1 to keep it the same after multiplication.
Measurement Covariance Matrix	***R***	$\$3{,}245.48^2 =$ $\$10{,}533{,}140.43$	Using the mean error value determined above, we will square that term and use it as the expected measurement variance.
Process Noise Matrix	***Q***	$\$2{,}796.04^2 =$ $\$7{,}817{,}847.74$	Without doing a deep dive into Bitcoin's price action based on its market cap or other market indicators, we are going to set process noise variance to the standard deviation of the measurement error squared. *We don't know all of the market conditions that contribute to this but we do know we need to account for additional error throughout the week. This is a reasonable value to start with.

Executing the Filter

Now that we came up with a system model, we can run the filter. I created this one in Microsoft Excel because it was very simple. Figure 10, shows the results of this filter and Table 3 shows

the actual values for each variable at each step. As you can see in Figure 10, this filter is able to process these measurements, track the price variation as well reduce the overall noise.

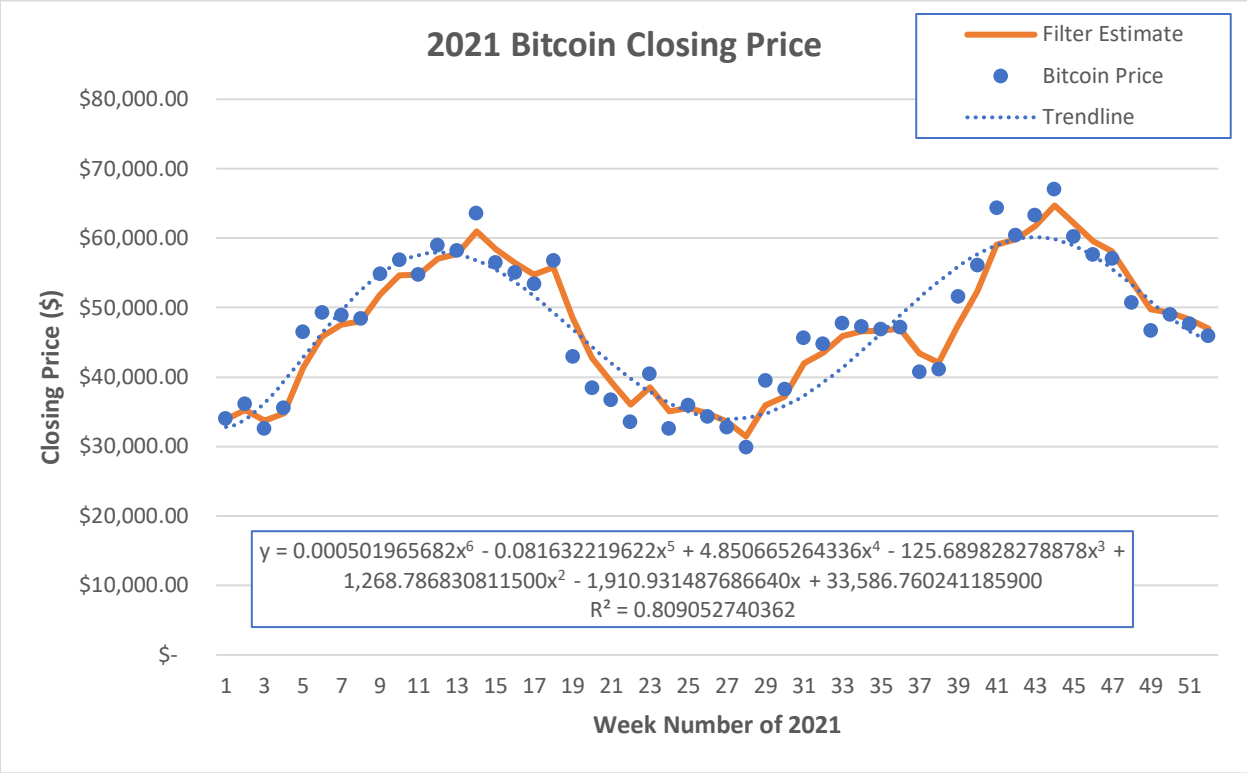


Figure 10: Bitcoin Weekly Closing Price Estimate

Table 3 shows the value of each variable through each processing step. Each week or row represents a new measurement processed by the Kalman Filter. What you can't see are the excel formulas in each of these cells that directly align with the Kalman Filter equations discussed in earlier chapters. Take a look at this table and then recreate it yourself in Excel. Or any other similar programs.

Week	z (closing price)	R	Q	x_P	P_P	K	x	P
1	$33,922.96	$10,533,140.43	$7,817,847.74				$33,922.96	$10,533,140.43
2	$36,069.80	$10,533,140.43	$7,817,847.74	$33,922.96	$18,350,988.17	0.6353	$35,286.92	$6,692,032.78
3	$32,569.85	$10,533,140.43	$7,817,847.74	$35,286.92	$14,509,880.52	0.5793	$33,712.65	$6,102,882.29
4	$35,510.29	$10,533,140.43	$7,817,847.74	$33,712.65	$13,920,730.04	0.5692	$34,735.98	$5,996,147.10
5	$46,481.11	$10,533,140.43	$7,817,847.74	$34,735.98	$13,813,994.84	0.5673	$41,399.89	$5,976,257.41

Table 3: Bitcoin Excel Kalman Filter

Figure 11, shows the Kalman Gain throughout the first half of these measurement price updates. As you can see, it quickly settles to a point of K = 0.5669. This value infers that on each update, the filter is going to trust the measurement just slightly more than the predicted state estimate and use it to contribute to the new estimate. This can be observed in Figure 10.

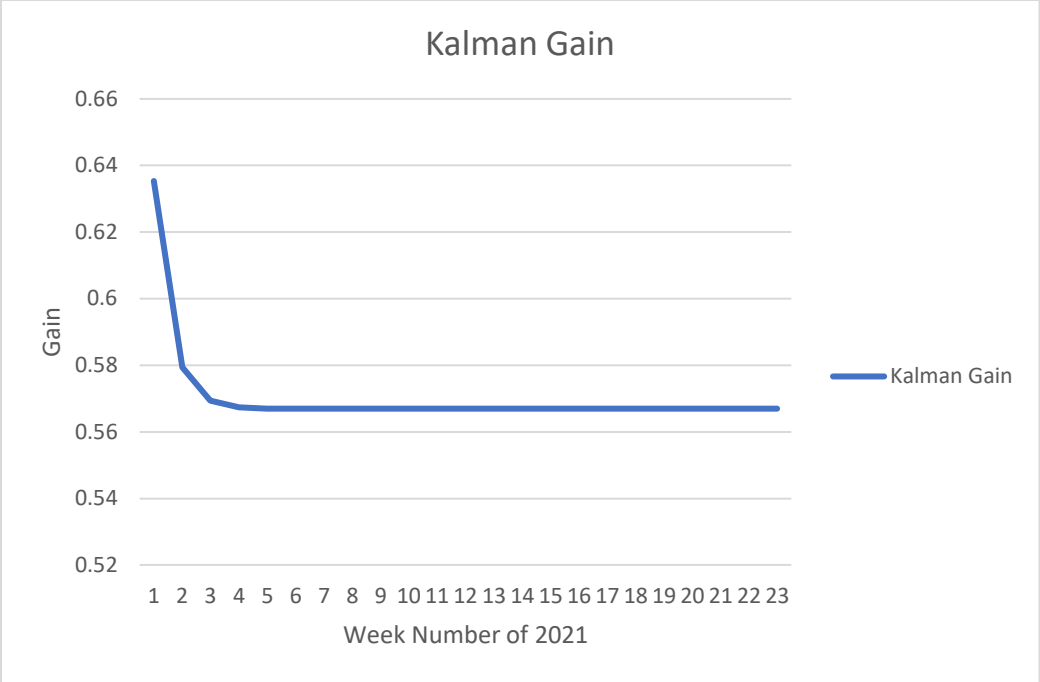


Figure 11: Kalman Gain

Figure 12 is the same chart as Figure 10 with the addition of 3 sigma error bounds that are based on the state covariance matrix ***P***. Why is this important? What does this tell us? So going back to how the Kalman Filter differs from the average, moving average, and exponential moving average filters; the Kalman Filter outputs a probability distribution for the state estimate. Using that covariance matrix, I added the 3 sigma error bounds to show whether or not the state estimate and the covariance matrix contains the actual value of the weekly price. (Remember the actual value is the just the 6^{th} order polynomial fit we applied earlier) One of the design criteria for Kalman Filters is that the state estimate's output contains the actual value of what is being estimated.

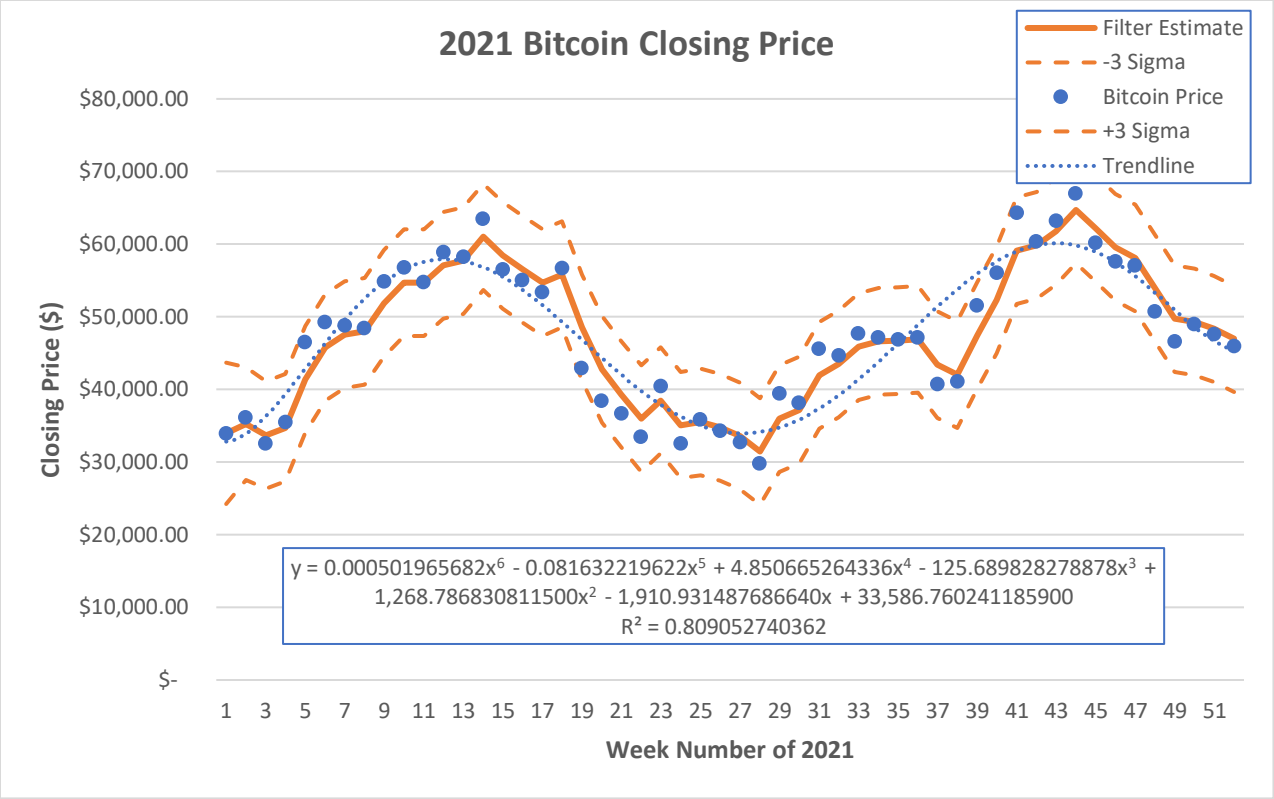


Figure 12: Kalman Filter Estimate with 3 Sigma Bounds

So if you look back at Figure 12, you can see that around weeks 37 through 39, the actual value falls outside the error bounds. That means at that point in time, our filter output was not right. Is this acceptable? Well, it depends on your application. I know you hate hearing that but it's the truth. Most systems expect this type of filter behavior from time to time but some require that the actual value is always contained.

One final thought on this is that our "actual" value is not true. So not containing this value on a 3 of 52 updates can be ruled as acceptable. Let us take a look at using a different process noise value, $\boldsymbol{Q}$, and how that affects our filter results.

Changing the System Model

The system model used above worked well to remove noise and track variation. Let us look at some data for a different system model. Specifically, what would it look like if we used a Q value of $\$1{,}000^2$ instead of $\$2{,}796.04^2$. What does that really mean? It means that we expect less process noise in the prediction step than the system model above. Take a look below at Figure 13, notice that the filter estimate has more points that don't contain the actual value and that the filter estimate lags behind the actual values.

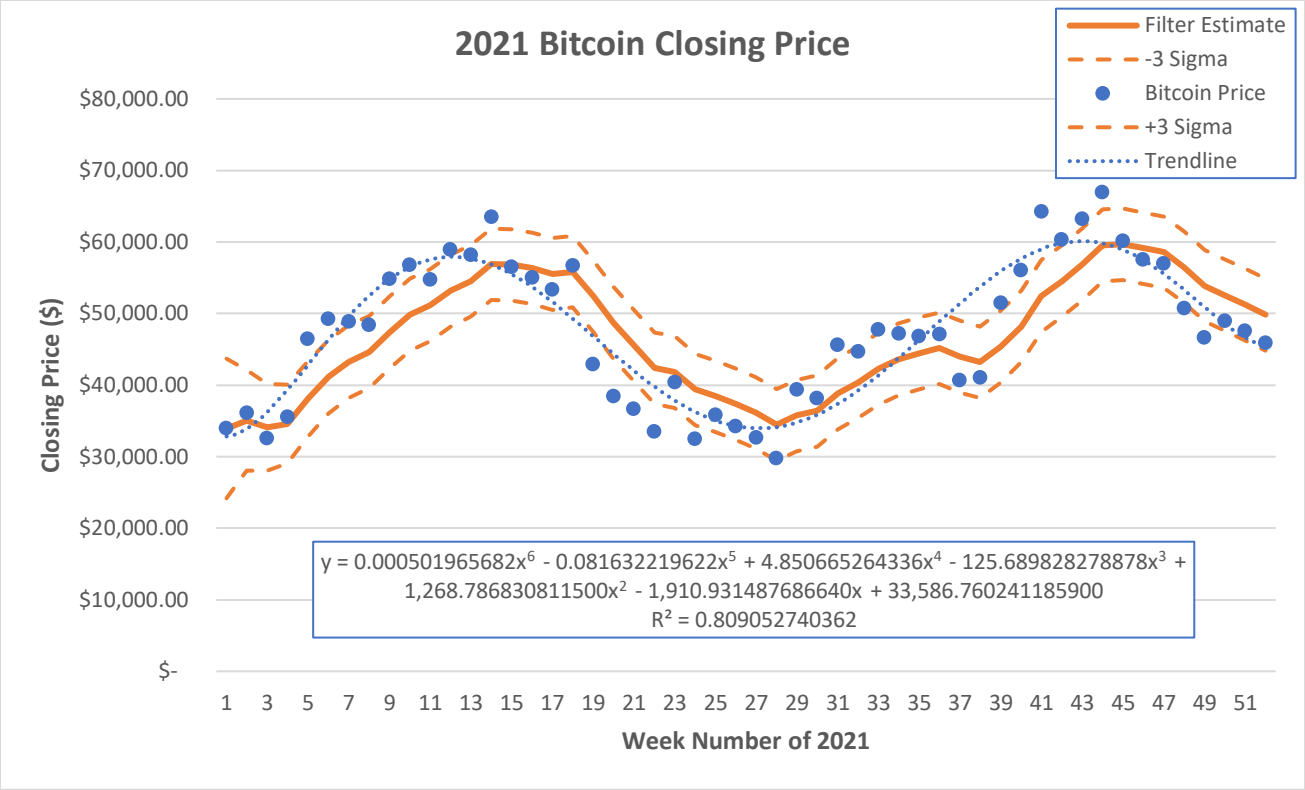


Figure 13: Bitcoin Filter Output with New Q Value

The reason that the filter estimate is lagging is that the filter gain is low. The reason the filter gain is low is because the process noise, $\boldsymbol{Q}$, does not accurately describe the system behavior.

A low filter gain means that the predicted estimate will be weighted heavier than the measurement values.

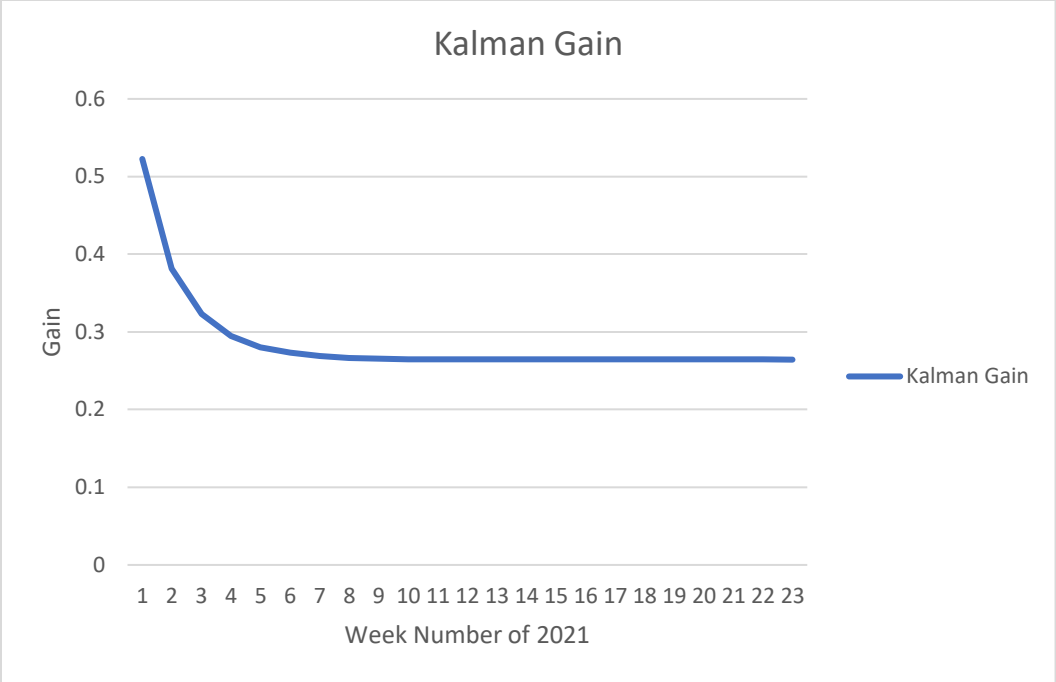


Figure 14: Filter Gain for Updated System Model

Chapter 4 Key Points

Chapter 4 covered the system model design for a simple Kalman Filter. We saw how a simple Kalman Filter can be designed and implemented in Microsoft Excel. We also saw how sometimes we have data sets that we don't know anything about so we need to do some analysis to come up with a system model that describes the data set. In summary, here is what we accomplished in Chapter 4:

- Performed statistical analysis on new data set to define system model
- Implemented system model and filter in Microsoft Excel
- Plotted results to analyze effectiveness of filter
- Modified system model and compared results to see if it was a better design

Chapter 4 implemented a simple Kalman Filter that removed noise and tracked variation for a measured parameter. In Chapter 5 we will use a Kalman Filter to do the same as Chapter 4 but we will also look at how the filter can be used to estimate other system state variables that are not being measured.

Chapter 5: Kalman Filter - Python Example – Estimating Velocity From Position

This chapter demonstrates how to implement a Kalman Filter in Python that estimates velocity from position measurements.

This example will use two Python libraries. I will use the NumPy Python library for arrays, matrices, and operations on those structures. And I will use Matplotlib to plot input and output data. Assuming these packages are installed, this is the code you will use to import and assign a variable name to these packages.

```
#!/usr/bin/python

# Import Libraries for Linear Algebra and Plotting
import numpy as np
from numpy.linalg import inv
import matplotlib.pyplot as plt

```

Kalman Filter Python Implementation

Implementing a Kalman Filter in Python is simple if it is broken up into its component steps. The component steps are modeled with individual functions. Note that these functions can be extended or modified to be used in other Kalman Filter applications. The algorithm framework remains the same.

Computing Measurements

In order to build and test a Kalman Filter, a set of input data is needed. For this example, the **getMeasurement(…)** function is used to simulate a sensor providing real-time position measurements of a performance automobile as it races down a flat road with a constant velocity of 60 meters per second.

This function initializes the position at 0 and velocity at 60 meters per second. Next, random noise, **v**, is computed and added to position measurement. Additional random noise, **w**, is computed and added to the velocity to account for small random accelerations. Lastly, the current position and current velocity are retained as truth data for the next measurement step.

```python
# Function to generate noisy measurements
def getMeasurement(updateNumber):
    # Initialize the function variables to align with true values
    if updateNumber == 1:
      getMeasurement.currentPosition = 0
      getMeasurement.currentVelocity = 60 # m/s

    # If the measurement is updated 10 times a second, the time difference
    # between updates will be 0.1 seconds
    dt = 0.1

    # Compute random noise to add to position and velocity
    # Note: np.random.randn(1) will return 1 value on the standard normal
    # distribution with mean of 0. The "8" multiplier serves as a way to scale
    # that error. The number 8 was chosen at random to add a visible amount of
    # of noise.
    w = 8 * np.random.randn(1)
    v = 8 * np.random.randn(1)

    # z = pos + vel*dt + noise error
    z = getMeasurement.currentPosition + getMeasurement.currentVelocity*dt + v

    # Reset true position to be the current position minus the noise error
    getMeasurement.currentPosition = z - v
    # Update velocity to account for real changes in accelerations
    getMeasurement.currentVelocity = 60 + w
    return [z, getMeasurement.currentPosition, getMeasurement.currentVelocity]
```

Filtering Measurements

Now that you have input measurements to process with your filter, it's time to code up your python Kalman Filter. The code for this example is consolidated into one function.
When the first measurement is reported, the filter is initialized. The measurement is in the following structures. **z** is the position measurement, **R** is the position variance, and **t** is the timestamp of the measurement.

$$z = pos_m \qquad R = \sigma^2_{pos_m} \qquad t = t_m$$

x is the two element state vector for position and velocity. **P** is the 2×2 state covariance matrix representing the uncertainty in **x**. **T** is the timestamp for the estimate.

$$x = \begin{bmatrix} pos \\ vel \end{bmatrix} \qquad P = \begin{bmatrix} \sigma^2_{pos} & \sigma_{posVel} \\ \sigma_{posVel} & \sigma^2_{vel} \end{bmatrix} \qquad T$$

Prediction

A is the state transition matrix for a system model that assumes constant linear motion. **H** is the state to measurement transition matrix. $\mathbf{H}^{\mathbf{T}}$ is the **H** matrix transposed. **R** is the input measurement variance. **Q** is the 2×2 system process noise covariance matrix. **Q** accounts for inaccuracy in the system model.

$$H = \begin{bmatrix} 1 & 0 \end{bmatrix} \qquad A = \begin{bmatrix} 1 & \Delta T \\ 0 & 1 \end{bmatrix} \qquad Q = \begin{bmatrix} 1 & 0 \\ 0 & 3 \end{bmatrix}$$

All measurements, after the first one, are filtered into the track state. First, the predicted state, $\mathbf{x_P}$, and predicted state covariance, $\mathbf{P_P}$, are computed by propagating the existing state and state covariance in time to align with the new measurements.

$$x_P = Ax = \begin{bmatrix} 1 & \Delta T \\ 0 & 1 \end{bmatrix} \begin{bmatrix} pos \\ vel \end{bmatrix} = \begin{bmatrix} pos + vel \cdot \Delta T \\ vel \end{bmatrix}$$

$$P_P = APA^T + Q \qquad P_P = \begin{bmatrix} 1 & \Delta T \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \sigma_{pos}^2 & \sigma_{posVel} \\ \sigma_{posVel} & \sigma_{vel}^2 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ \Delta T & 1 \end{bmatrix} + \begin{bmatrix} 1 & 0 \\ 0 & 3 \end{bmatrix}$$

$$P_P = \begin{bmatrix} \sigma_{pos}^2 + 2 \cdot \Delta T \cdot \sigma_{posVel} + \sigma_{vel}^2 \cdot \Delta T^2 + 1 & \sigma_{posVel} + \Delta T \cdot \sigma_{vel}^2 \\ \sigma_{posVel} + \Delta T \cdot \sigma_{vel}^2 & \sigma_{vel}^2 + 3 \end{bmatrix}$$

Compute Kalman Gain

For this example, it is assumed that measurements have an update rate of 10 per second and therefore a delta time, **dt**, of 0.1 seconds is used. Next the Kalman Gain, **K**, is computed using the input measurement uncertainty, **R**, and the predicted state covariance matrix, $\mathbf{P_P}$. This computation was broken down into two steps, first compute the innovation matrix , **S**, and then compute the Kalman Gain, **K**, with the innovation.

$$S = HP_PH^T + R$$

$$S = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} \sigma_{pos}^2 + 2 \cdot \Delta T \cdot \sigma_{posVel} + \sigma_{vel}^2 \cdot \Delta T^2 + 1 & \sigma_{posVel} + \Delta T \cdot \sigma_{vel}^2 \\ \sigma_{posVel} + \Delta T \cdot \sigma_{vel}^2 & \sigma_{vel}^2 + 3 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} + 10$$

$$S = \begin{bmatrix} \sigma_{pos}^2 + 2 \cdot \Delta T \cdot \sigma_{posVel} + \sigma_{vel}^2 \cdot \Delta T^2 + 1 + 10 \end{bmatrix}$$

$$K = PH^T S^{-1}$$

Estimation

Lastly, the Kalman Gain, **K**, is used to compute the new state, **x**, and state covariance estimate, **P**.

$$x = x_P + K \cdot (z - Hx_P)$$

$$P = P_P - K \cdot H \cdot P_P$$

```python
def filter(z, updateNumber):
    dt = 0.1
    # Step 1: Initialize Filter Parameters:
    # System Model Variables, State Estimate, and State Covariance
    if updateNumber == 1:
        filter.x = np.array([[0],
                             [20]])
        filter.P = np.array([[5, 0],
                             [0, 5]])

        filter.A = np.array([[1, dt],
                             [0, 1]])
        filter.H = np.array([[1, 0]])
        filter.HT = np.array([[1],
                              [0]])
        filter.R = 10
        filter.Q = np.array([[1, 0],
                             [0, 3]])

    # Predict State Forward
    x_prime = filter.A.dot(filter.x)
    # Predict Covariance Forward
    P_prime = filter.A.dot(filter.P).dot(filter.A.T) + filter.Q
    # Compute Kalman Gain
    S = filter.H.dot(P_prime).dot(filter.HT) + filter.R
    K = P_prime.dot(filter.HT)*(1/S)

    # Estimate State
    residual = z - filter.H.dot(x_prime)
    filter.x = x_prime + K*residual

    # Estimate Covariance
    filter.P = P_prime - K.dot(filter.H).dot(P_prime)

    return [filter.x[0], filter.x[1], filter.P, K];

```

Testing Kalman Filter

After creating a simulation to generate measurements and coding up a Kalman Filter to process those measurements, it is now time to test your Kalman Filter. It is a pretty simple function that loops through the expected number of measurements, “gets” the current measurement and passes it into the filter function. Subsequently, the data saved off in arrays so that it can be plotted.

```
74  def testFilter():
75      # Define Range of Measurements to Loop Through
76      dt = 0.1
77      t = np.linspace(0, 10, num=300)
78      numOfMeasurements = len(t)
79
80      # Initialize arrays to save off data so it could be plotted
81      measTime = []
82      measPos = []
83      measDifPos = []
84      estDifPos = []
85      estPos = []
86      estVel = []
87      posBound3Sigma = []
88      posGain = []
89      velGain = []
90
91      # Loop through each measurement
92      for k in range(1,numOfMeasurements):
93          # Generate the latest measurement
94          z = getMeasurement(k)
95          # Call Filter and return new State
96          f = filter(z[0], k)
97          # Save off that state so that it could be plotted
98          measTime.append(k)
99          measPos.append(z[0])
100         measDifPos.append(z[0]-z[1])
101         estDifPos.append(f[0]-z[1])
102         estPos.append(f[0])
103         estVel.append(f[1])
104         posVar = f[2]
105         posBound3Sigma.append(3*np.sqrt(posVar[0][0]))
106         K = f[3]
107         posGain.append(K[0][0])
108         velGain.append(K[1][0])
109
110     return [measTime, measPos, estPos, estVel, \
111             measDifPos, estDifPos, posBound3Sigma, \
112             posGain, velGain];
```

Plot Kalman Filter Results

Figure 15 shows the position measurement error and estimate error relative to the actual position of the vehicle. This plot shows how the Kalman Filter smooths the input measurements and reduces the positional error. This first plot is a quick and simple way to see if tour filter is working correctly. If something is visibly off in this plot, you can be sure something is not working right.

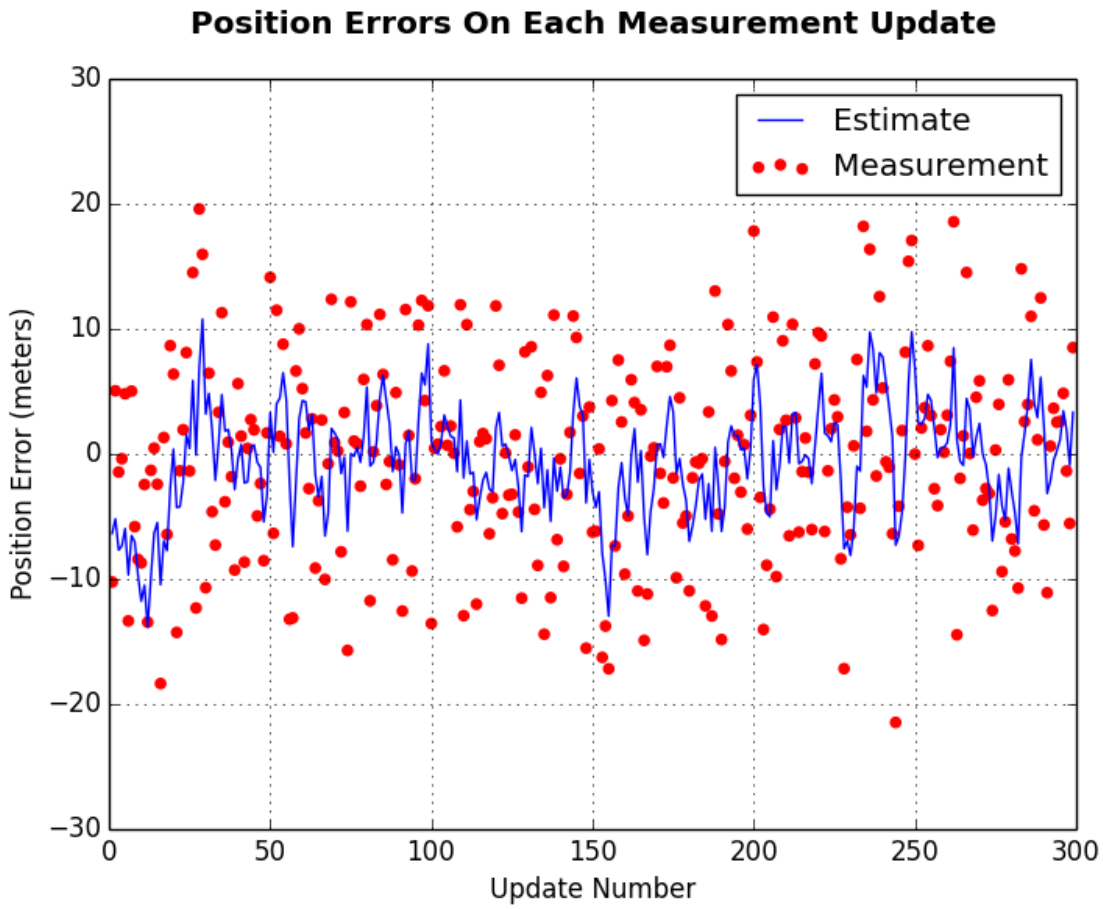


Figure 15: Position Error Analysis

Figure 16 shows the velocity estimate for the vehicle based on the input measurements. It can be seen that after the first five or so measurements the filter starts to settle on the vehicles actual speed which is 60 meters per second.

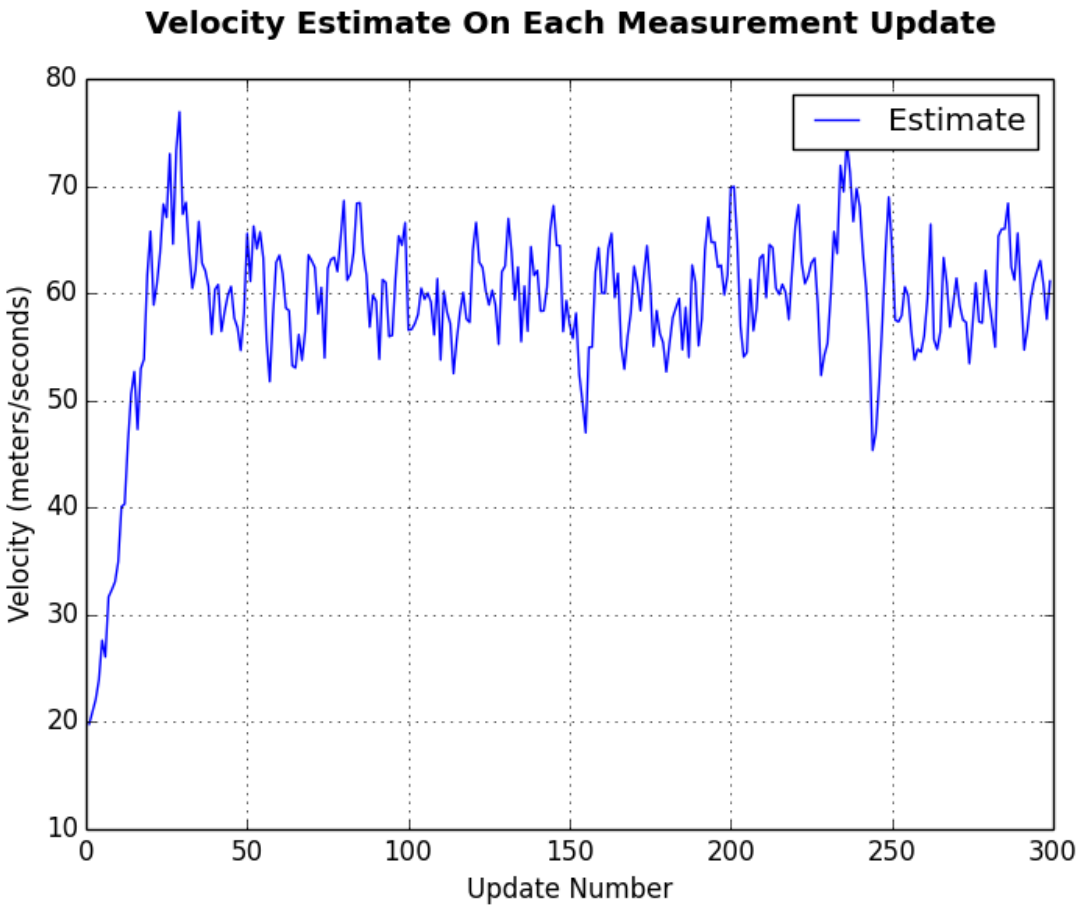


Figure 16: Velocity Estimate

Figure 17 shows the position and velocity gains. As you can see, they quickly settle and stay steady throughout the filtering process. Because these values are less than 0.5, that infers that the state estimate has a larger weight to it when linearly combining the predicted state estimate with the input measurement.

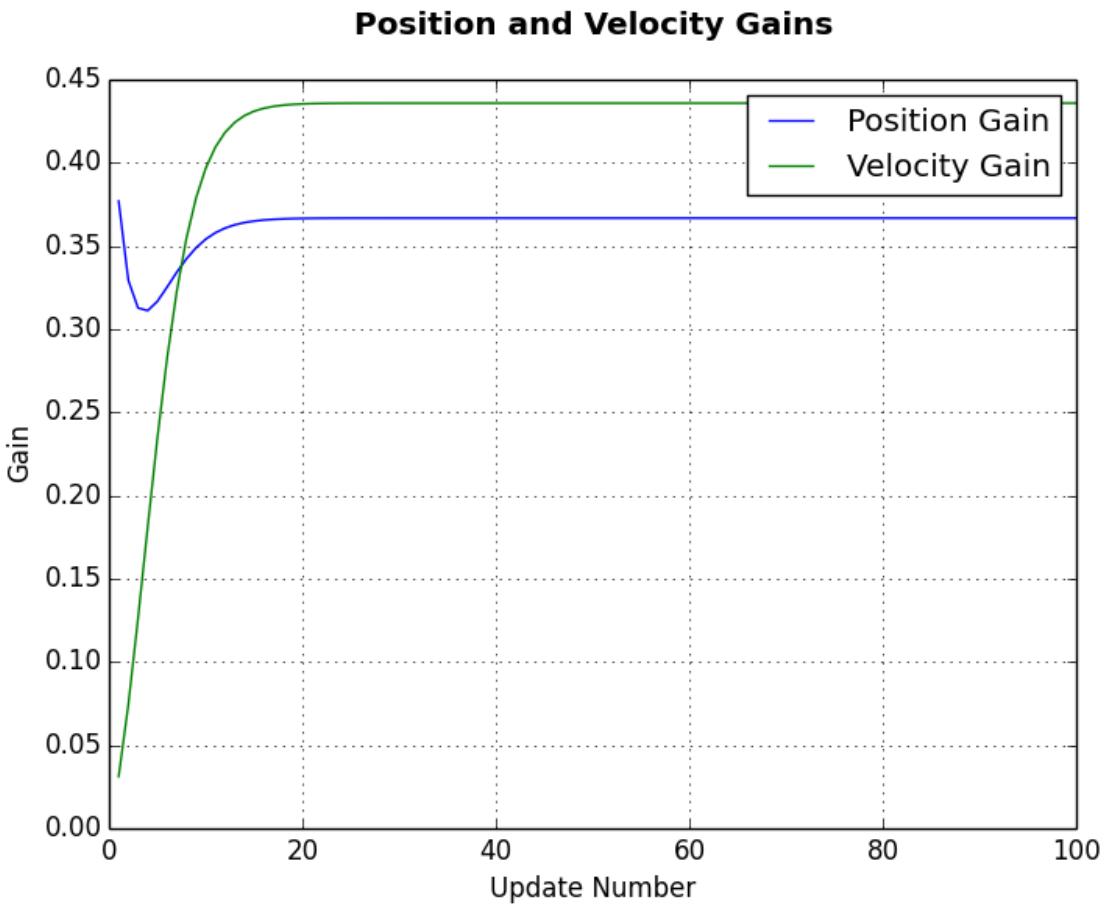


Figure 17: Kalman Gains

The Python code used to create these plots is below. The variables used below come from the functions in the above source code.

```
# Execute Test Filter Function
t = testFilter()
# Plot Results
plot1 = plt.figure(1)
plt.scatter(t[0], t[1])
plt.plot(t[0], t[2])
plt.ylabel('Position')
plt.xlabel('Time')
plt.grid(True)

plot2 = plt.figure(2)
plt.plot(t[0], t[3])
plt.ylabel('Velocity (meters/seconds)')
plt.xlabel('Update Number')
plt.title('Velocity Estimate On Each Measurement Update \n', fontweight="bold")
plt.legend(['Estimate'])
plt.grid(True)

plot3 = plt.figure(3)
plt.scatter(t[0], t[4], color = 'red')
plt.plot(t[0], t[5])
plt.legend(['Estimate', 'Measurement'])
plt.title('Position Errors On Each Measurement Update \n', fontweight="bold")
plt.ylabel('Position Error (meters)')
plt.xlabel('Update Number')
plt.grid(True)
plt.xlim([0, 300])

plot4 = plt.figure(4)
plt.plot(t[0], t[7])
plt.plot(t[0], t[8])
plt.ylabel('Gain')
plt.xlabel('Update Number')
plt.grid(True)
plt.xlim([0, 100])
plt.legend(['Position Gain', 'Velocity Gain'])
plt.title('Position and Velocity Gains \n', fontweight="bold")
plt.show()

```

Analyze Kalman Filter Results

The plots generated by this Python example clearly show that the Kalman Filter is working. Based on your application and use cases, this performance may not be suitable. If it does not meet your requirements then it will have to be tuned. When I say “tuned”, I am referring to adjusting the system model.

As part of this chapter, I encourage you to experiment with the above example. Modify the Q matrix values by making them larger and smaller. What you will see is that the larger they get, the larger the gains get and the smaller they get the smaller the gains become.

Chapter 6: Extended Kalman Filter – Python Example

What is the Extended Kalman Filter?

The Extended Kalman Filter is a special Kalman Filter used when working with nonlinear systems. Since the Kalman Filter can not be applied to nonlinear systems, the Extended Kalman Filter was created to solve that problem. Specifically, it was used in the development of navigation control systems aboard Apollo 11. And like many other real world systems, the behavior is not linear so a traditional Kalman Filter can not be used. This chapter will walk you through an Extended Kalman Filter Python example so you can see how one can be implemented.

Figure 18 below, it can be seen that the Extended Kalman Filter's general flow is exactly the same as the Kalman Filter. The main two differences between the Kalman Filter and the Extended Kalman Filter are represented in red font. In addition to these differences, there are some lower level differences that I will cover in more detail below.

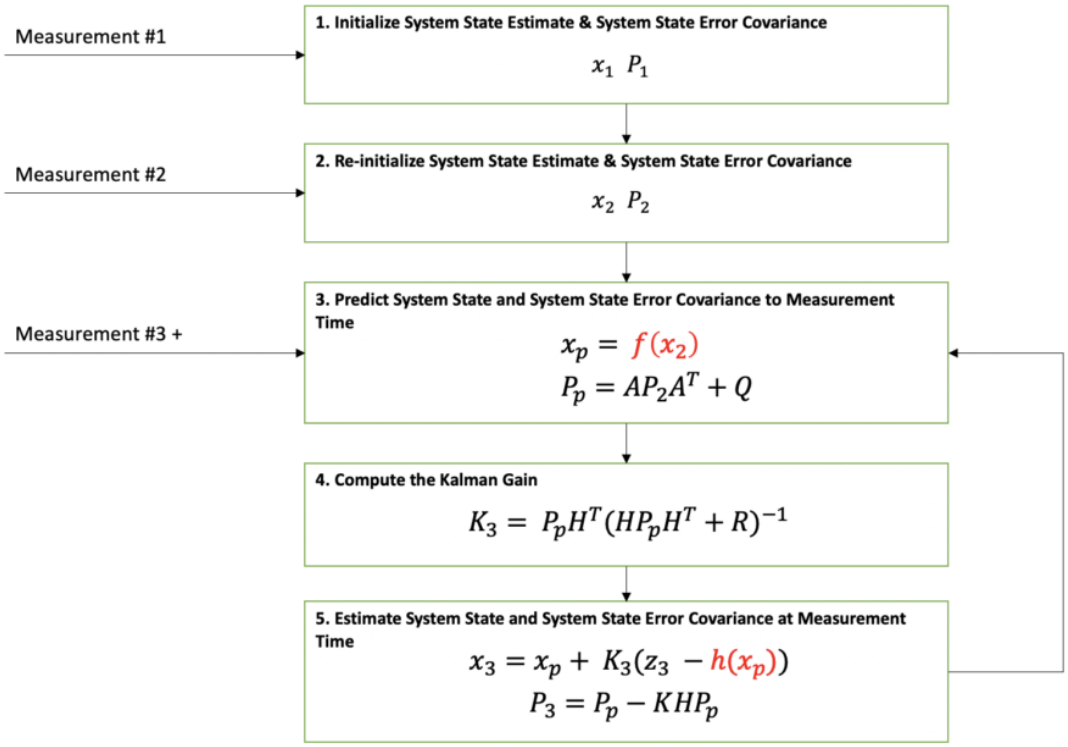


Figure 18: Extended Kalman Filter Process Overview

Python Example Overview

The Extended Kalman Filter Python example chosen for this chapter takes in measurements from a ground based radar tracking a ship in a harbor and estimates the ships position and velocity. The radar measurements are in a local polar coordinate frame and the filter's state estimate is in a

local cartesian coordinate frame. The measurement to state estimate relationship i.e. polar to cartesian is nonlinear.

In this example, the ship is traveling in a straight line at constant velocity of 20 meters per second or about 50 miles per hour. It is located about 4100 meters away or about 2.5 miles. The measurements are being reported at 1 hertz or once per second. The measurements are being reported as the range and azimuth (θ) values in local polar frame.

Figure 19 provides a visual for this example. It can be seen that the ground based radar is monitoring the ship as it travels linearly in the x direction.

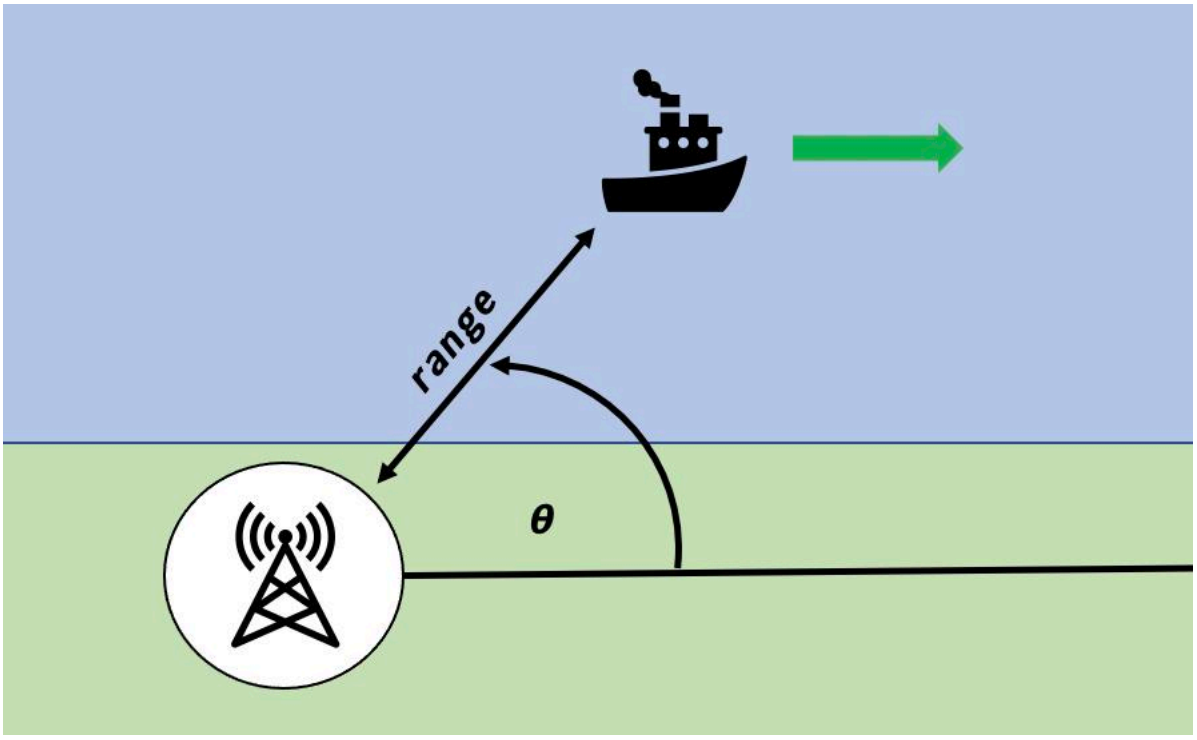


Figure 19: Extended Kalman Filter Example Diagram

Figures 20 and 21 show the actual or true values for the ships range and azimuth vs their measured values. Note, I will use azimuth and bearing interchangeably in this example. As you can see in these figures, the measured values are noisy compared to the actual positional values.

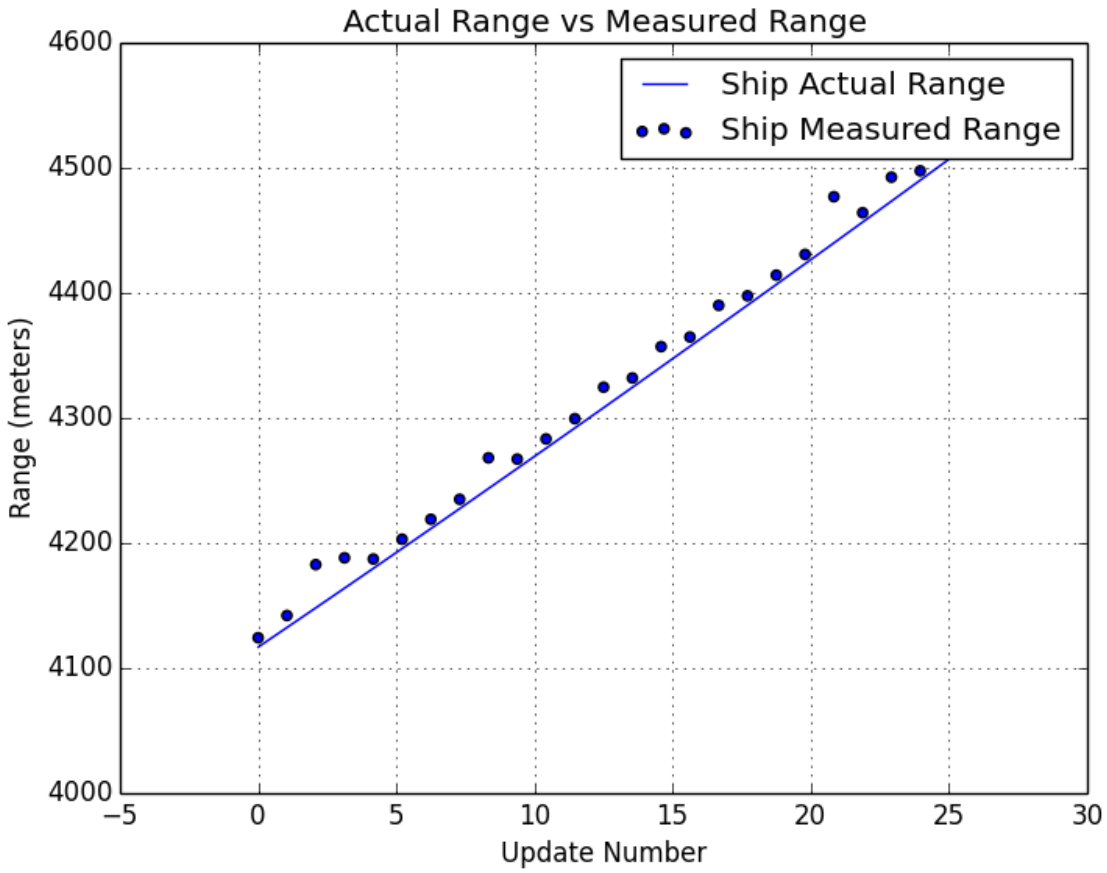


Figure 20: Actual vs Measured Range

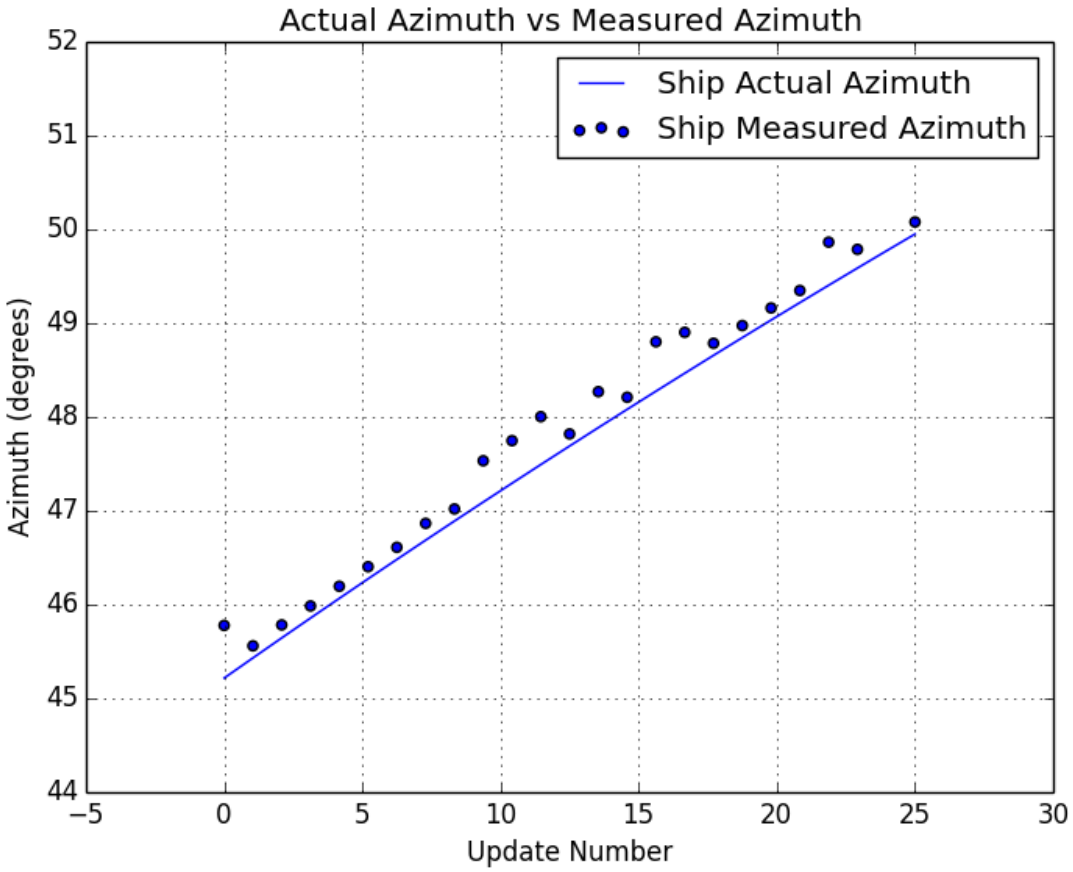


Figure 21: Actual vs Measured Azimuth

A simulation technique was used to generate these measurements and that will be explained in detail below.

On the Initialization of the Extended Kalman Filter

As can be seen in the Extended Kalman Filter diagram above, the first two measurements are needed before filtering can be begin. The reason for this is because the output state estimate needs to be initialized. There are different approaches to filter initialization based on your application but for this example, the first two position measurements will be used to form the state estimate which consists of the position and velocity.

Logically, one can make sense of this. The first position measurement received only tells you about the ships position at one point in time. Only after you receive the second measurement, can you compute a velocity i.e. change in position measurement divided by the time difference between the two measurements.

Extended Kalman Filter Python Example

Let's take a look at the input, output, and system model equations before we dive into the code. This should make reading the code much easier.

Input and Output of the Extended Kalman Filter

As mentioned above, the input measurements for this Extended Kalman Filter Python example are in the local polar coordinate frame. And these are the equations that represent these measurements.

$$z = \begin{bmatrix} r_m \\ az_m \end{bmatrix} \qquad R = \begin{bmatrix} \sigma_{r_m}^2 & \sigma_{raz_m} \\ \sigma_{raz_m} & \sigma_{az_m}^2 \end{bmatrix} \qquad t = t_m$$

$\boldsymbol{z}$ is the measurement vector, $\boldsymbol{R}$ is the measurement covariance matrix and $\boldsymbol{t}$ is the time at which the measurement was taken. $\boldsymbol{m}$ is used as a subscript to indicate measurements.
The above input measurements are filtered and a system state estimate is computed and output in the following form.

$$x = \begin{bmatrix} x \\ y \\ \dot{x} \\ \dot{y} \end{bmatrix} \qquad P = \begin{bmatrix} \sigma_x^2 & \sigma_{xy} & \sigma_{x\dot{x}} & \sigma_{x\dot{y}} \\ \sigma_{xy} & \sigma_y^2 & \sigma_{y\dot{x}} & \sigma_{y\dot{y}} \\ \sigma_{x\dot{x}} & \sigma_{y\dot{x}} & \sigma_{\dot{x}}^2 & \sigma_{\dot{x}\dot{y}} \\ \sigma_{x\dot{y}} & \sigma_{y\dot{y}} & \sigma_{\dot{x}\dot{y}} & \sigma_{\dot{y}}^2 \end{bmatrix} \qquad T$$

$\boldsymbol{x}$ is the state vector, $\boldsymbol{P}$ is the state covariance matrix, and $\boldsymbol{T}$ is the time at which the state estimate is valid. In this example, this time will reflect the last measurement time that was used to compute the state estimate.

System Model Equations for Prediction Step

As mentioned above, the system model equations are where the primary differences between the Extended Kalman Filter and the traditional Kalman Filter lie. For this example,
the $\boldsymbol{A}$ and $\boldsymbol{Q}$ matrices used are not any different than matrices that would be used in a Kalman Filter. Those equations are the following.

$$A = \begin{bmatrix} 1 & 0 & \Delta T & 0 \\ 0 & 1 & 0 & \Delta T \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad Q = \begin{bmatrix} 100 & 0 & 0 & 0 \\ 0 & 100 & 0 & 0 \\ 0 & 0 & 250 & 0 \\ 0 & 0 & 0 & 250 \end{bmatrix}$$

The $\boldsymbol{A}$ matrix above is the state transition matrix. This matrix is used to extrapolate the state vector and state covariance from the last data time to align with the data time of the new

measurement. This matrix was developed assuming the ship was traveling linearly at constant velocity i.e. using physics equations of motion.

The ***Q*** matrix is the system process noise matrix. This represents the uncertainty in the system prediction model. For this example, the ***A*** matrix assumes the ship is traveling in a linear motion at a constant velocity. But in reality, this is not always true. The ***Q*** matrix is used to add error to the state estimate prediction to account for those differences in linear motion and small accelerations or decelerations. The values used in this example are not optimized for any particular use case but seem to do a relatively good job.

Here are the prediction equations from Step 3 in the diagram above.

$$x_P = f(x) = Ax = \begin{bmatrix} x + \dot{x} \cdot \Delta T \\ y + \dot{y} \cdot \Delta T \\ \dot{x} \\ \dot{y} \end{bmatrix} \qquad P_P = APA^T + Q$$

System Model Equations for Estimation Step

The estimation process for the Extended Kalman Filter is performed in two steps. First, the Kalman Gain is computed. Second, the state vector and state covariance for the new measurement time is computed with the Kalman Gain. Both of these steps require the use of an ***H*** matrix. The ***H*** matrix is the state-to-measurement transition matrix. This matrix is used to transform the predicted state vector and covariance matrix into the same form as the input measurement. For this Extended Kalman Filter example, the state and measurement relationship is nonlinear. Because of this nonlinear relationship, the H matrix is different than that of a traditional Kalman Filter. Specifically, the ***H*** matrix is a ***jacobian*** matrix. The derivation and applications of the ***jacobian*** matrix are outside the scope of this book. The important features to note about the ***jacobian*** at this time is that is made up of first order partial differential equations. Here is the ***H*** matrix for this example.

$$H = \begin{bmatrix} \frac{\partial r}{\partial x} & \frac{\partial r}{\partial y} & \frac{\partial r}{\partial \dot{x}} & \frac{\partial r}{\partial \dot{y}} \\ \frac{\partial az}{\partial x} & \frac{\partial az}{\partial y} & \frac{\partial az}{\partial \dot{x}} & \frac{\partial az}{\partial \dot{y}} \end{bmatrix} = \begin{bmatrix} \frac{x}{r} & \frac{y}{r} & 0 & 0 \\ \frac{-y}{r^2} & \frac{x}{r^2} & 0 & 0 \end{bmatrix}$$

The partial differential equations were computed with the following polar to cartesian relationships for range and azimuth.

$$r = \sqrt{x^2 + y^2}$$
$$az = \tan^{-1}\frac{y}{x}$$

After the ***H*** matrix is computed, the Kalman Gain can be computed. The Kalman Gain equations that follow consist of two steps. First, compute the innovation matrix, ***S***. Then compute the gain, ***K***.

$$S = HP_P H^T + R$$

$$K = PH^T S^{-1}$$

The Kalman Gain, ***K***, is used in the estimation steps below. The estimation steps include estimating the new system state as well as the associated state covariance matrix. One difference you can see in these equations compared to the traditional Kalman Filter estimation equations, is the ***h(xP)*** function. This function represents the state-to-measurement nonlinear transformation.

$$x = x_P + K(z\ - h(x_P))$$

$$P = P_P - KHP_P$$

h(xP) is defined by the equation below for this example. These equations should look familiar because these are the same equations used to compute the ***jacobian H*** matrix.

$$h(x_P) = \begin{bmatrix} \sqrt{x^2 + y^2} \\ \tan^{-1}\frac{y}{x} \end{bmatrix}$$

That concludes the overview of the equations that will be used for this Extended Kalman Filter Python tutorial. Next, the implementation details will be reviewed with code snippets and comments.

Python Implementation for the Extended Kalman Filter Example

In order to develop and tune a Python Extended Kalman Filter, you need the following source code functionality:

- Input measurement data (real or simulated)
- Extended Kalman Filter algorithm
- Data collection and visualization tools
- Test code to read in measurement data, execute filter logic, collect data, and plot the collected data

For this tutorial, multiple Python libraries are used to accomplish the above functionality. The NumPy Python library is used for arrays, matrices, and linear algebra operations on those structures. The Matplotlib is used to plot input and output data. Assuming these packages are installed, this is the code used to import and assign a variable name to these packages.

```
#!/usr/bin/python

import numpy as np
from numpy.linalg import inv
import matplotlib.pyplot as plt

```

Extended Kalman Filter Algorithm

The algorithm is implemented using one function with two input arguments, the measurement data structure, ***z***, and the update number, ***updateNumber***. The update number is not a necessary input variable. But for this example, it was easier to implement. Additionally, the update number was used as the time stamp for each measurement because the measurements were reported once a second. This can be updated to be more robust.

There are three main processing threads through this function determined by the update number: **initialization, re-initialization, and filtering**. Here is the code:

```
def ekfilter(z, updateNumber): # z = [r, b]
    dt = 1.0
    j = updateNumber
    # Initialize State
    if updateNumber == 0: # First Update
        # compute position values from measurements
        temp_x = z[0][j]*np.sin(z[1][j]*np.pi/180) # x = r*sin(b)
        temp_y = z[0][j]*np.cos(z[1][j]*np.pi/180) # y = r*cos(b)
        # State vector - initialize position values
        ekfilter.x = np.array([[temp_x],
                               [temp_y],
                               [0],
                               [0]])
        # State covariance matrix - initialized to zero for first update
        ekfilter.P = np.array([[0, 0, 0, 0],
                               [0, 0, 0, 0],
                               [0, 0, 0, 0],
                               [0, 0, 0, 0]])
        # State transistion matrix - linear extrapolation assuming constant velocity
        ekfilter.A = np.array([[1, 0, dt, 0],
                               [0, 1, 0, dt],
                               [0, 0, 1, 0],
                               [0, 0, 0, 1]])
        # Measurement covariance matrix
        ekfilter.R = z[2][j]
        # System error matrix - initialized to zero matrix for first update
        ekfilter.Q = np.array([[0, 0, 0, 0],
                               [0, 0, 0, 0],
                               [0, 0, 0, 0],
                               [0, 0, 0, 0]])
        # Residual and kalman gain
        # not computed for first update but initialized so it could be output
        residual = np.array([[0, 0],
                             [0, 0]])
        K = np.array([[0, 0],
                      [0, 0],
                      [0, 0],
                      [0, 0]])

```

```
    # Reinitialize State
    if updateNumber == 1: # Second Update
        # Get previous state vector values
        prev_x = ekfilter.x[0][0]
        prev_y = ekfilter.x[1][0]
        temp_x = z[0][j]*np.sin(z[1][j]*np.pi/180) # x = r*sin(b)
        temp_y = z[0][j]*np.cos(z[1][j]*np.pi/180) # y = r*cos(b)
        #  Compute velocity - vel = (pos2 - pos1)/deltaTime
        temp_xv = (temp_x - prev_x)/dt
        temp_yv = (temp_y - prev_y)/dt
        # State vector - reinitialized with new position and computed velocity
        ekfilter.x = np.array([[temp_x],
                               [temp_y],
                               [temp_xv],
                               [temp_yv]])
        # state covariance matrix - initialized to large values
        ekfilter.P = np.array([[100, 0, 0, 0],
                               [0, 100, 0, 0],
                               [0, 0, 250, 0],
                               [0, 0, 0, 250]])
        # State transistion matrix - linear extrapolation assuming constant velocity
        ekfilter.A = np.array([[1, 0, dt, 0],
                               [0, 1, 0, dt],
                               [0, 0, 1, 0],
                               [0, 0, 0, 1]])
        # Measurement covariance matrix - provided by the measurment source
        ekfilter.R = z[2][j]
        # System error matrix
        # adds 4.5 m std dev in position and 2 m/s std dev in velocity
        ekfilter.Q = np.array([[20, 0, 0, 0],
                               [0, 20, 0, 0],
                               [0, 0, 4, 0],
                               [0, 0, 0, 4]])
        # Residual and kalman gain-  initialized so it could be output
        residual = np.array([[0, 0],
                             [0, 0]])
        K = np.array([[0, 0],
                      [0, 0],
                      [0, 0],
                      [0, 0]])
```

```
    if updateNumber > 1: # Third Update and Subsequent Updates
      # Predict state and state covariance forward
      x_prime = ekfilter.A.dot(ekfilter.x)
      P_prime = ekfilter.A.dot(ekfilter.P).dot(ekfilter.A.T) + ekfilter.Q
      # Form state to measurement transition matrix
      x1 = x_prime[0][0]
      y1 = x_prime[1][0]
      x_sq = x1*x1
      y_sq = y1*y1
      den = x_sq+y_sq
      den1 = np.sqrt(den)
      ekfilter.H = np.array([[  x1/den1,    y1/den1, 0, 0],
                             [y1/den, -x1/den, 0, 0]])
      ekfilter.HT = np.array([[x1/den1, y1/den],
                              [y1/den1, -x1/den],
                              [0, 0],
                              [0, 0]])
      # Measurement covariance matrix
      ekfilter.R = z[2][j]
      # Compute Kalman Gain
      S = ekfilter.H.dot(P_prime).dot(ekfilter.HT) + ekfilter.R
      K = P_prime.dot(ekfilter.HT).dot(np.linalg.inv(S))
      # Estimate State
      temp_z = np.array([[z[0][j]],
                         [z[1][j]]])
      # Convert the predicted cartesian state to polar range and azimuth
      pred_x = x_prime[0][0]
      pred_y = x_prime[1][0]
      sumSquares = pred_x*pred_x + pred_y*pred_y
      pred_r = np.sqrt(sumSquares)
      pred_b = np.arctan2(pred_x, pred_y) * 180/np.pi
      h_small = np.array([[pred_r],
                          [pred_b]])
      # Compute residual difference between state and measurement for data time
      residual = temp_z - h_small
      # Compute new estimate for state vector using the Kalman Gain
      ekfilter.x = x_prime + K.dot(residual)
      # Compute new estimate for state covariance using the Kalman Gain
      ekfilter.P = P_prime - K.dot(ekfilter.H).dot(P_prime)
    return [ekfilter.x[0], ekfilter.x[1], ekfilter.P, ekfilter.x[2], ekfilter.x[3], K, residual];
```

Computing the Measurements

In order to test this algorithm implementation, real data or simulated data is needed. For this tutorial, simulated data is generated and used to test the algorithm.

Simulating this data was a little more complex than our previous example in Chapter 5. The code below is well commented, but I want to point out a few things to keep in mind while reviewing it:

- First, the ships position and velocity were initialized
- Next, the ships actual position data was computed by assuming linear motion in the x direction and then converted into the polar coordinate frame

- Next, using the actual polar position data, noise was added to each measurement using a function that randomly generates zero mean gaussian noise
    - This zero mean gaussian noise was scaled by the components standard deviation assigned
    - Note, the standard deviation values can be modified to produce a more accurate or less accurate sensor
- Lastly, the measured data is saved off in an array and ready to be processed by the filtering algorithm

Additional processing can be added to this simulation to make it more robust. For example, while computing the truth data, additional white noise can be added to the velocity to account or random accelerations. Your filter performance will only be as good as your simulation fidelity.

```
def getMeasurements():
    # Measurements are taken 1 time a second
    t = np.linspace(0, 25, num=25)
    numOfMeasurements = len(t)
    # Define x and y initial points -> Range = 4100 meters or 2.5 miles
    x = 2900
    y = 2900
    # Velocity magnitude = 22 meters per second OR 50 miles per hour
    vel = 22.0
    # Create storage arrays for true position data
    t_time = []
    t_x = []
    t_y = []
    t_r = []
    t_b = []
    # Compute the Real Position Data - This simulation use the cartesian
    # coordinate frame for computing new positional points and then converts
    # them to the polar coordinates.
    for i in range(0,numOfMeasurements):
        # Set the delta time to 1 second
        dT = 1.0
        # Store off the update time for this position update
        t_time.append(t[i])
        # Compute the new x and y position data with the assumption all of the
        # velocity motion is in the x direction i.e. linear motion
        x = x+dT*vel
        y = y
        # Store off the computed x and y data
        t_x.append(x)
        t_y.append(y)
        # Compute the sum of the squares of x and y as an intermediate step
        # before computing the range and storing it off
        temp = x*x + y*y
        r = np.sqrt(temp)
        t_r.append(r)
        # Compute the azimuth (or bearing) with the arctan2 function and convert
        # it to degrees. Then store this azimuth data
        b = np.arctan2(x, y) * 180/np.pi
        t_b.append(b)
```

```
47
48      # Create storage containers for polar measurement data
49      m_r = []
50      m_b = []
51      m_cov = []
52      # Bearing standard deviation = 9 milliradians (in degrees)
53      sig_b = 0.009*180/np.pi
54      sig_r = 30 # Range Standard Deviation = 30 meters
55      # Storage containers for cartesian measurements - for analysis purposes
56      m_x = []
57      m_y = []
58      for ii in range(0, len(t_time)):
59          # Compute the error for each measurement
60          # By taking the max between .25 of the defined standard deviation and
61          # the randomly generated normal error, it guarantees an error
62          temp_sig_b = np.maximum(sig_b * np.random.randn(), 0.25*sig_b)
63          temp_sig_r = np.maximum(sig_r * np.random.randn(), 0.25*sig_r)
64          # Save off the measurement values for bearing and range as a Function
65          # of the true value + the error generated above
66          temp_b = t_b[ii] + temp_sig_b
67          temp_r = t_r[ii] + temp_sig_r
68          # Save off the measurement data
69          m_b.append(temp_b)
70          m_r.append(temp_r)
71          m_cov.append(np.array([[temp_sig_r*temp_sig_r, 0],
72                                 [0, temp_sig_b*temp_sig_b]]))
73          m_x.append(temp_r*np.sin(temp_b*np.pi/180))
74          m_y.append(temp_r*np.cos(temp_b*np.pi/180))
75
76      return [m_r, m_b, m_cov, t_r, t_b, t_time, t_x, t_y, m_x, m_y]
77      #        0    1     2      3    4    5      6   7    8    9
78
```

Testing the Extended Kalman Filter

The test code below was used to test this algorithm. The **getMeasurements()** function returns an array of arrays of measurement information in the following form: [measurement_range, meas_azimuth, measurement_cov, meas_time]

```
f_x = []
f_y = []
f_x_sig = []
f_y_sig =[]
f_xv = []
f_yv = []
f_xv_sig = []
f_yv_sig =[]

z = getMeasurements()
for iii in range(0, len(z[0])):
  f = ekfilter(z, iii)
  f_x.append(f[0])
  f_y.append(f[1])
  f_xv.append(f[3])
  f_yv.append(f[4])
  f_x_sig.append(np.sqrt(f[2][0][0]))
  f_y_sig.append(np.sqrt(f[2][1][1]))

plot1 = plt.figure(1)
plt.grid(True)
plt.plot(z[5], z[3])
plt.scatter(z[5], z[0])
plt.title('Actual Range vs Measured Range')
plt.legend(['Ship Actual Range', 'Ship Measured Range'])
plt.ylabel('Range (meters)')
plt.xlabel('Update Number')

plot2 = plt.figure(2)
plt.grid(True)
plt.plot(z[5], z[4])
plt.scatter(z[5], z[1])
plt.title('Actual Azimuth vs Measured Azimuth')
plt.legend(['Ship Actual Azimuth', 'Ship Measured Azimuth'])
plt.ylabel('Azimuth (degrees)')
plt.xlabel('Update Number')

```

```
plot3 = plt.figure(3), plt.grid(True)
plt.plot(z[5], f_xv)
plt.plot(z[5], f_yv)
plt.title('Velocity Estimate On Each Measurement Update \n', fontweight="bold")
plt.legend(['X Velocity Estimate', 'Y Velocity Estimate'])

# Compute Range Error
e_x_err = []
e_x_3sig = []
e_x_3sig_neg = []
e_y_err = []
e_y_3sig = []
e_y_3sig_neg = []
for m in range(0, len(z[0])):
    e_x_err.append(f_x[m]-z[6][m])
    e_x_3sig.append(3*f_x_sig[m])
    e_x_3sig_neg.append(-3*f_x_sig[m])
    e_y_err.append(f_y[m]-z[7][m])
    e_y_3sig.append(3*f_y_sig[m])
    e_y_3sig_neg.append(-3*f_y_sig[m])

plot4 = plt.figure(4), plt.grid(True)
line1 = plt.scatter(z[5], e_x_err)
line2, = plt.plot(z[5], e_x_3sig, color='green')
plt.plot(z[5], e_x_3sig_neg, color='green')
plt.ylabel('Position Error (meters)')
plt.xlabel('Update Number')
plt.title('X Position Estimate Error Containment \n', fontweight="bold")
plt.legend([line1, line2,], ['X Position Error', '3 Sigma Error Bound'])

plot5 = plt.figure(5), plt.grid(True)
yline1 = plt.scatter(z[5], e_y_err)
yline2, = plt.plot(z[5], e_y_3sig, color='green')
plt.plot(z[5], e_y_3sig_neg, color='green')
plt.ylabel('Position Error (meters)')
plt.xlabel('Update Number')
plt.title('Y Position Estimate Error Containment \n', fontweight="bold")
plt.legend([yline1, yline2,], ['Y Position Error', '3 Sigma Error Bound'])
plt.show()
```

Filter Results and Analysis

There are many different ways to analyze your filter's performance. And every application will require different analysis because each will have its own performance requirements. Here are two plots that were used to understand if this filter was working.

Figure 22 is an error plot for the x position estimate. This plot includes the x position error, which is computed as the x position estimate minus the real x position that was used to generate the polar measurement. In addition to the x position error, the green lines represent the estimated error bounds computed from the state covariance estimate. You can see in this plot that sometimes the x position error falls outside the error bounds. Again, based on your application requirements, this may or may not be okay. After update number 15, it can be seen that x position error starts to settle within the error bounds.

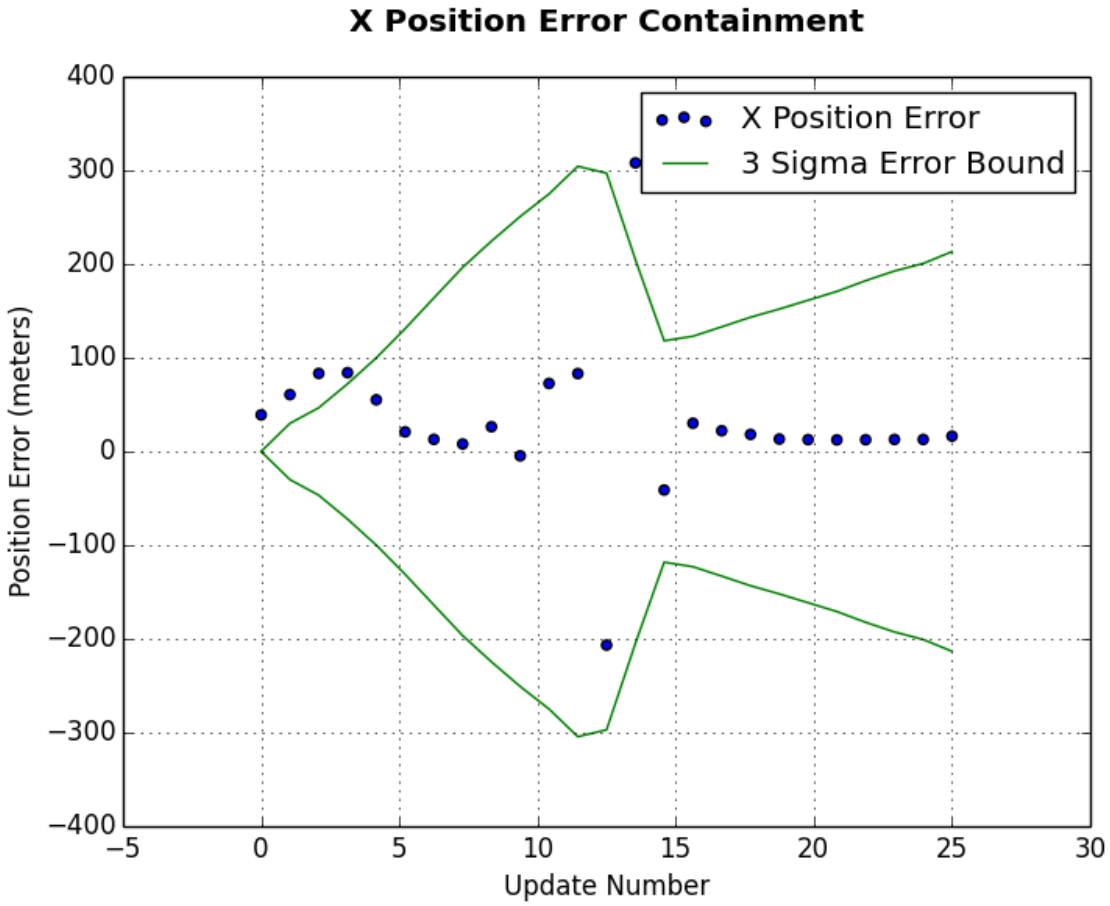


Figure 22: X Position Error Containment

Figure 23 shows the estimated velocity for the ship. It can be seen that the x and y velocity values vary during the first few updates but then begin to settle at the actual values used for measurement generation i.e. x velocity at 20 meters per second and y velocity at 0 meters per second. This is a good sign that the filter is working because it is able to accurately estimate the ships velocity based only on position measurements in a different coordinate frame.

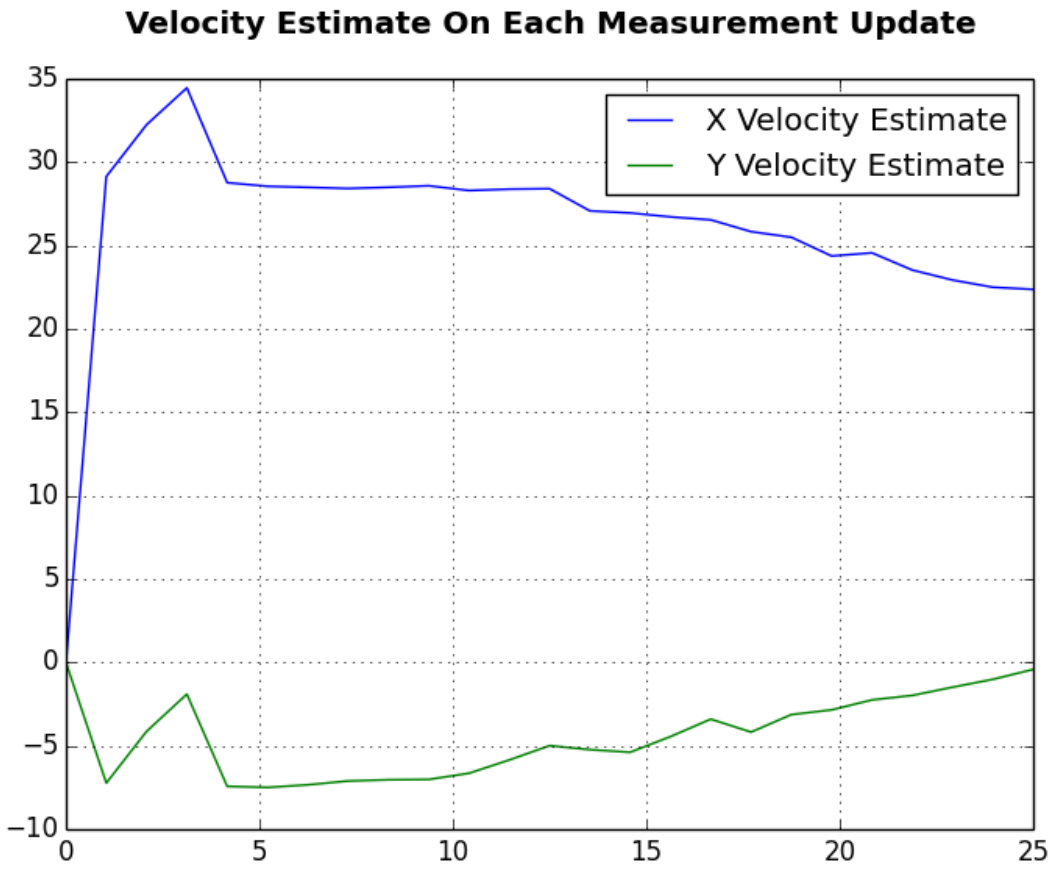


Figure 23: Velocity Estimates

There is a chance that as more measurements are filtered, the Extended Kalman Filter will diverge, that is, the state estimate will start to diverge from the input measurements. A lot of research has been done and is being done to find ways to correct that divergence. In future versions of this book, I will add in information about how to deal with this expected divergence.

Chapter 7: Getting Started with Data and Simulation

If you are designing and implementing a Kalman Filter, that means you are processing data. This may be data fed to your algorithm in a real-time system or it is data being post processed after collection. Whatever the case may be, when you start designing your filter, you need some sample data or the ability to run your algorithm in that real-time system immediately. In most cases, having your own set of data to run through your algorithm is the easiest to work with because it allows you to run and re-run while you work your way through any obvious bugs early on. But let us consider all of our options when starting our design process.

Option 1: Integrate Filter into System

The first and most obvious option you have for getting data into your Kalman Filter is by integrating it into your production system where it could process and record data real-time. Although this seems like a great idea, I caution against it to start because you may lose a lot of time and system resources working through obvious bugs before your algorithm is ready for this level of testing.

However, I do recognize every situation is different and that this approach may be the most cost effective and/or only option. So, if this is the case, I recommend recording as much data as you can for post analysis. At a minimum record your inputs and outputs but for a more in-depth analysis, record your gains.

Of course, the final step of your design journey will be to integrate it into your system and run it with real data, so whenever that occurs, be ready to capture data and use it for analysis. Be sure to identify and test as many different test cases as you can to cover the range of possible inputs.

Option 2: Use Data from Previous System Operation

There is a good chance if you are designing this filter, you have a set of data from the system you are designing it for. Whether it's from last week and two years ago, usually, there is sample of data hanging around. **This data, in my opinion, is the best possible data to use for filter design and development.**

If this is the case, take the data and format it into something your algorithm can process. I typically like to create commas separated values (csv) format. For most programming languages, there is an existing library that can read these files and generate data objects with that file's information. If you didn't already know this, an excel file can be saved off in the form of a csv file!

Keep in mind that this is a sample of data. So, when identifying data sets to use, seek out a variety of events and time periods. Try to aggregate multiple sets of data that cover a range of inputs your filter may encounter.

Option 3: Find Comparable Data from Different System

If you don't have access to data from previous system operation then maybe you can find some comparable data online. I understand this is not an option when working with a proprietary systems or new systems under development but you'd be surprised what you can find on the internet. When you don't have readily available data, try looking around for it. In recent years, data has become more accessible online. With that said, if you do take this approach, you will most likely need to reformat the data to fit your needs. This can be easy or difficult based on the dataset so be prepared for this.

Once you get your data set into the proper format, do some quick analysis to ensure it aligns with what you expect your algorithm to process. This may seem obvious but there is mislabeled data out there that I have started to use without checking. This mistake cost me many hours of headache!!

Option 4: Simulation

If you don't have access to existing data, you can generate some yourself with a simulation. This approach is used often. Typically, a quick simulation can be used to get you started. I like this approach because it allows you to work through the obvious coding bugs and generate useful plots without having real data. In essence, this approach allows you to get started when you don't have anything else to go on. In addition to getting started quickly, this approach also allows you to come up with different test cases for your filter. These can be extreme cases or invalid cases that will allow you to improve the existing code to be prepared for these types of cases.

Depending on your application, the simulation you need to create can be complex or it can be simple.

In Chapter 5, I used a simple simulation to generate position data from a car traveling in a straight line. This was easy to do because I understood the basic physics equations that modeled the cars behavior i.e. linear motion. In Chapter 6, I used a similar simple simulation to generate position data from a ship traveling in a linear motion. Although this simulation was a little different because the input ship position data was in two dimensions rather than one. Additionally, I had to report measurements in polar coordinates so I simulated the ships movement in a cartesian coordinate frame and then converted that to the polar frame to use as inputs. There are probably other ways to do that but that is the approach that made the most sense to me.

Let us consider the basic requirements for a simulation to be valuable to our filter design.

Simulation Requirements

The purpose of your simulation is to output set of measurements in the form of the expected measurements your filter will process. Here are the required elements of these measurements.

- **z**, an nx1column vector that contains the measurement variables

- **R**, an nxn measurement covariance matrix that characterizes the measurement data relative to the truth data
- **t**, the time tag associated with the measurement data

These can be generated at different levels of fidelity depending on your needs and/or know-how. But in all cases they are governed by the following equation.

$$z_t = H_t x_t + v_t$$

Where:

- **z**, an nx1column vector that contains the measurement variables
- **H**, is the nxm state-to-measurement transition matrix (or the observation matrix as it is commonly referred to)
- **x,** is the mx1 column vector that contains the true system state data at timestamp, **t**
- **v,** is the observation noise which is assumed to be zero mean gaussian distributed with covariance matrix, **R**

So, what does this really mean? It means that our simulation can be broken down into two parts: 1) generating true system state data over time and 2) adding noise to that true system state data.

Generating true system state data requires that you understand how the system will behave over time.

For example, in Chapter 5, the car followed a linear trajectory. So, to generate that truth data, there needed to be initial position and an initial velocity. From there, each time step assumed a linear path from the previous state. But in addition to that linear path, noise was added to both the position and velocity. You may be asking yourself but why velocity? Well its easy to understand that the position data will be noisy but let us also consider the fact that the real car will have small accelerations and decelerations that also contribute to the position of the car at each time stamp. This produces a higher fidelity model then just adding noise to the position but may not be necessary to get you started.

When "adding noise" to the true system data, I prefer to take the approach of using built in random functions. In Python's NumPy library, there is a **randn()** function. You will see this in my code snippets from Chapters 5 and 6. This function returns a value between -3 and 3 which is the range of values of a normal distribution. By taking this value and multiplying it by the standard deviation you expect from the measurement, you will get a gaussian distributed error with the standard deviation you desire that you can add onto each true value. This will make your measured values gaussian distributed. As can also see in Chapter 5 and 6, these error terms can be used to create the covariance matrix that gets processed alongside the measurements.

Chapter 8: Getting Started with Performance Analysis

Designing, implementing, and evaluating a Kalman Filter is not easy. They are all closely related to each other and require you to have an understanding of the system, the filter, and the software executing the filter.

Chapters 1-7 walked you through what a Kalman Filter is, what its different steps and parts are, how to simulate data to pass through it, and talked a little bit about analyzing your filter output. This chapter is going to diver deeper into analyzing your filtering results to determine whether or not your filter is "working."

Logging Data

In order to do any type of analysis, you will need to log your filter inputs and filter outputs, at a minimum. Most likely during your design journey, you will start logging intermediate step data e.g. Kalman Gain matrix or the predicted state estimate.

Based on your system, programming language, data logging abilities, and other factors; this may be easy or more of nuisance. In most cases you will want to log lots of data for debug purposes but in the production execution, you will trim down your logging needs.

Regardless of your development environment, you will need to record data in order to analyze it after the fact.

Keep in mind that if this is a production execution, logging all filter inputs will allow you to repackage that same data as inputs for a simulation you can run outside of real time system execution.

Defining Requirements

Okay, so assuming you have successfully run your filter and collected data, it is time to do an analysis.

Our goal for filter performance analysis is determine if it's "working". Our filter is considered to be working if it meets our system requirements. This will be application specific, but these requirements have to be defined in order for you to understand if your filter is working.

Requirements can be specific to the filter output or they can be a function of the filter output. Let's look at two examples to see the difference.

Requirement 1: The radar gun must determine the cars velocity within a standard deviation of 1 mph.
Requirement 2: An officer can only give a ticket to someone driving 5 mph over the speed limit with 95% accuracy.

Both of the requirements could be used for the same system design reason. In order for a radar gun to be used effectively, it has to be accurate enough to identify speeders. If the radar gun is accurate within a large standard deviation, then officers of the law would be seeing very large speed ranges making it difficult for them to determine who was speeding and who was not speeding.

Let us consider this example in more detail. Here are the facts:

- You are designing a Kalman Filter that takes in position measurements and outputs a system estimate for position and velocity.
- This Kalman Filter will live inside a radar gun used by highway officers that will be giving out speeding tickets if the radar gun informs them that the car, they are measuring is driving over the speed limit.
- The only requirement you know of is Requirement 2 up above. "An officer can only give a ticket to someone driving 5 mph over the speed limit with 95% accuracy."

With all of this information you go ahead and design a filter very similar to the one designed in Chapter 5. You go ahead and build out a small simulation to provide input data to your algorithm. You plot the data and see the position estimate is smoother than the position measurements but you wonder, is my filter meeting the requirements?

That's a great question. Let's look back at Requirement 2 and consider what it really means. Requirement 2 infers that a highway officer can look at his or his radar gun after estimating a cars speed and determine instantly whether they were speeding or not with confidence.

How does this relate to our filter? Let us consider what your filter output is. Your filter output contains a velocity estimate with a variance-covariance matrix i.e. probability distribution. So if we know what the velocity estimate is, what the actual car velocity is, and what our association variance is for that velocity we should be to determine when our filter meets these requirements.

Lets work backwards from the requirement, if the filter is 95% confident that the car is traveling 5 mph over the speed limit then we can say that the lower bound or -2-sigma bound of our velocity estimate will be at the 5 mph over the speed limit. If that is what we are working toward then we will want our standard deviation of our velocity estimate to be small enough so the officer doesn't see wide range of numbers. Let us consider the upper and lower bounds of a 95% confidence interval for a radar gun with varying degrees of accuracy. These are displayed in Table 4:

Std Deviation (mph)	95% Conf Lower Bound = 60 mph
1	62
2	64
3	66
4	68
5	70
6	72
7	74
8	76

Table 4: Radar Gun Values for 60 mph Tickets

As you can see in Table 4, as the filter outputs accuracy grows, the minimum speed number on the radar gun to issue a ticket starts to not make sense. For starters, this would be poor user experience for the officer to try to understand real speed threshold vs what the radar gun is outputting. But more importantly, as the standard deviation grows, so does the minimum speed estimate. This means that many cars who are speeding will not be picked up by officer. Based on this table we can go ahead and work towards a standard deviation of 1 mph. Funny right? That's what are Requirement 1 was. Technically it could be larger but that's what we are going to go with for this example.

Validating Requirements

Once you have requirements determined that are measurable, then it's time to determine if your filter meets them.

Considering the example we have been working with so far, how do we determine if our filter is estimating the cars velocity with a standard deviation of 1 mph or less?

This can be determined different ways but there is one plot that we can make that will allow us to validate this requirement rather simply.

Figure 24 shows an error plot where where the velocity error is defined as:

$$Velocity\ Error = Velocity\ Estimate - \ Actual\ Velocity$$

And the 3-sigma error bound is defined as:

$$3 - Sigma = \pm 3 * sqrt(Velocity\ Variance)$$

Where the velocity variance is taken from the state covariance matrix. There are 2 key takeaways from this plot: 1) the error is contained by state covariance and 2) the average standard deviation for this estimate is 3.5 mph. Seeing that the error is contained is only part of the solution. That's great but now we have an estimate with large uncertainty that does not meet our requirements.

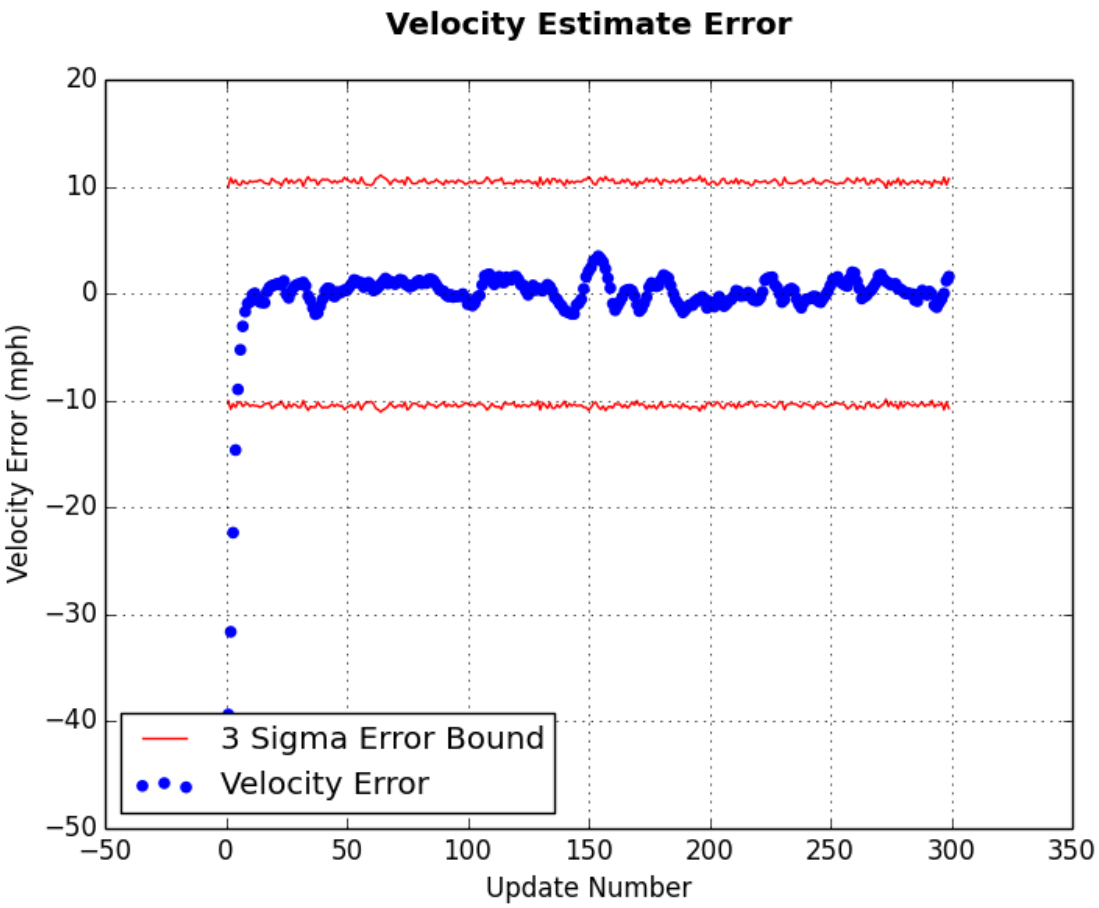


Figure 24: Velocity Error (Large)

After adjusting the Q matrix to filter out more noise than the previous system model, we get Figure 25. There are 2 key takeaways from this plot: 1) the error is contained by state covariance and 2) the average standard deviation for this estimate is ~1.0 mph. Check and check! Our state estimate for velocity contains the true velocity as well as having a small enough uncertainty to meet our requirements.

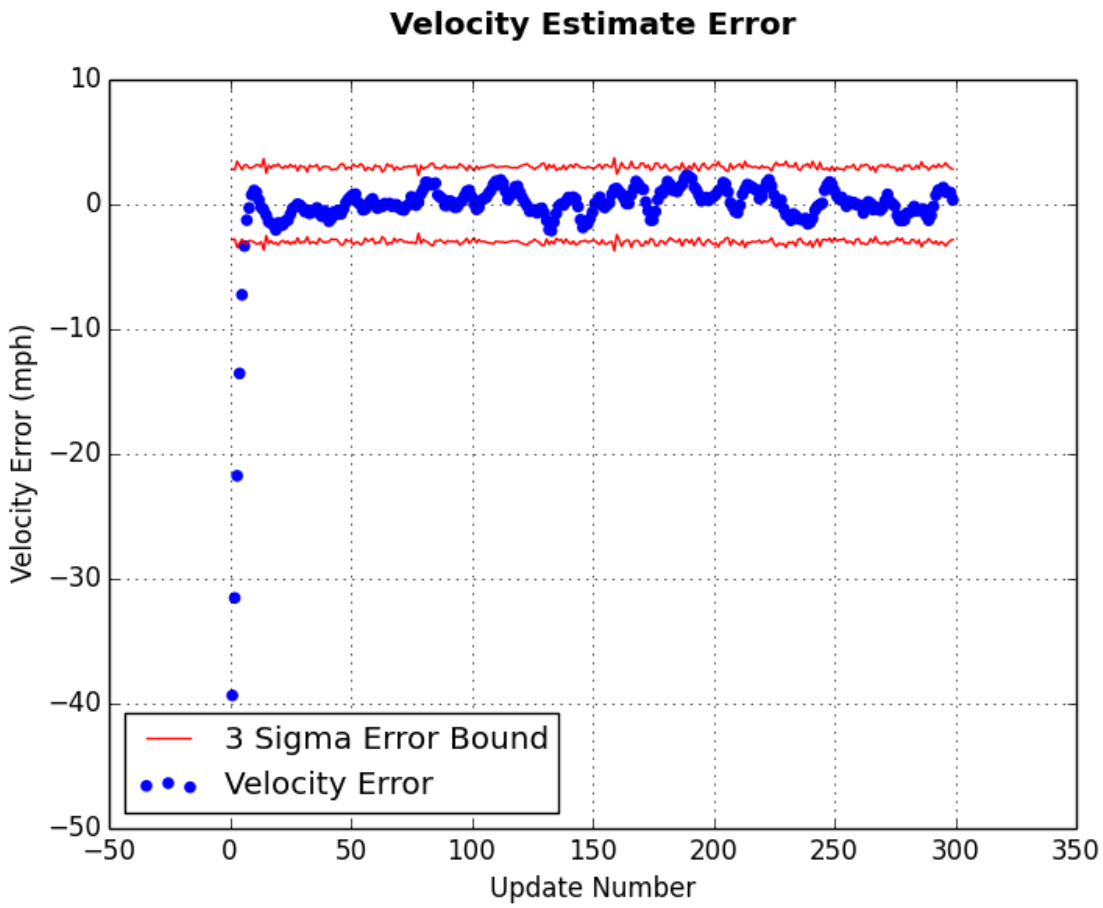


Figure 25: Velocity Error (small)

Summary

Performance analysis is one of the most difficult parts of the Kalman Filter design process. As stated in this chapter, you need to log the necessary data, generate plots that tell you what is happening, and evaluate whether or not your filter meets your defined requirements.

You can break this process down into 3 steps:

- Data Collection
- Requirements Definition
- Requirements Validation

A simple example was used in this chapter to illustrate this process. As you work on your Kalman Filters I am sure these tools will help you get started and headed in the right direction. But I will warn you this process is not as easy as it was for this example. You will have to spend more time understanding how your system model is affecting your results. And when you finally do crack that code, it will feel glorious! So enjoy it when you do!

Chapter 9: Sensor Fusion

Since this book is written for beginners, I do not have a detailed explanation of the Kalman Filters ability to do sensor fusion. I will however go over the basics of this topic.

What is Sensor Fusion?

Sensor fusion is when information is combined from two or more sensors in a way that minimizes the limitations of all sensors. There are many examples in real world settings where one sensor does not provide accurate enough information to solve you problem. Most times, the solution will mean using multiple sensors with known weaknesses and combining them together to make minimize them.

For example, there are many tech companies out there building autonomous vehicles. Each company has its own sensor suite with radar, lidar, and/or cameras. If you search around the internet you will stories about how each company prefers different sensors for different reasons. Regardless of these reasons, one thing is true for all of these companies. They all need multiple sensors on these vehicles that have different strengths and weaknesses. Some perform better in bad weather and at night, while others perform better on a nice sunny day. Some are better for detecting the range of an object or a person (i.e. radar or lidar) while others allow the system to understand what street signs are present like stop signs and stop lights (i.e. cameras). So when designing a complex system that can understand its surroundings and make decisions, more than likely a Kalman Filter or Kalman Filter derivative is being used to combine that information through sensor fusion for decision making.

Implementation Differences

So as you can imagine, setting up a Kalman Filter for sensor fusion is different than what we have been discussing. But don't get discouraged because a lot is the same between these two different approaches.

The main design problem is determining how to accommodate two different sensors that may provide different types of measurements at different frequencies. This is where your linear algebra skills and system modeling will be tested.

Additional Resources

If you are working on a sensor fusion problem and are having a hard time finding goo resources for your problem space, you may need to look into some of the common control theory textbooks. Unfortunately, those are usually expensive and overly dense on the math aspects.

If you can't find the right guidance online or in a textbook, send me an email at hello.kalmanfilter@gmail.com I can't promise I can help for your issue but I may be able to point you in the right direction.

References

This ebook was developed after researching and reading countless blog posts, books, and research papers as well as watching various tutorials on the topic. As I continue to further my understanding of this topic, I continue to find new resources. Here is list of sources I used when developing this ebook.

[1] Blackman, Samuel S., and Robert Popoli. *Design and Analysis of Modern Tracking Systems*. Artech House, 1999. 147-195

[2] Blackman, S. S., and J. W. Casey, "A Rating System for all Tennis Players," Operations Research, Vol. 28, No. 3, May-June 1980, pp. 489-502

[3] Gelb, A., Applied Optimal Estimation, Cambridge, MA: MIT Press, 1974.

[4] "Kalman Filter." *Wikipedia*, Wikimedia Foundation, 4 Apr. 2022, https://en.wikipedia.org/wiki/Kalman_filter.

[5] "Kalman." *MATLAB & Simulink*, https://www.mathworks.com/help/control/ug/kalman-filtering.html.

[6] R. E. Kalman, "A New Approach to Linear Filtering and Prediction Problems", *Research Institute for Advanced Study,* 1960.

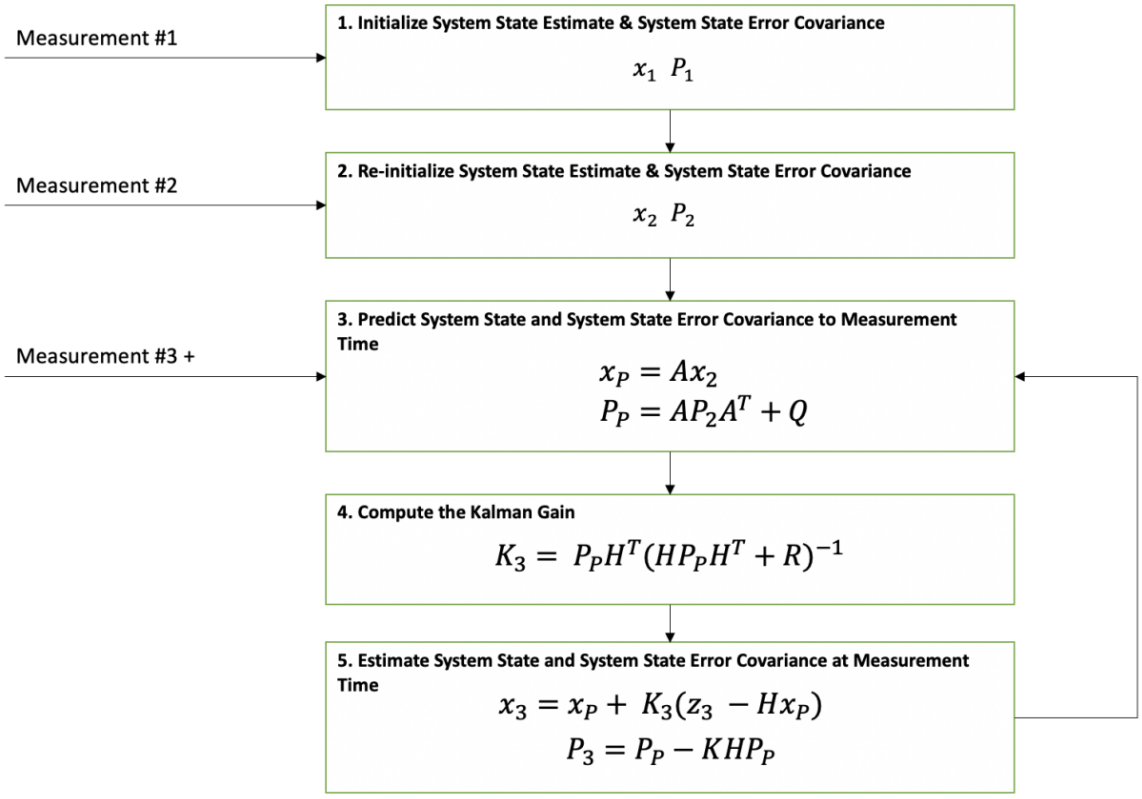
[7] Tbabb. *Bzarg*, 11 Aug. 2015, https://www.bzarg.com/p/how-a-kalman-filter-works-in-pictures/.

Appendix A: Kalman Filter Derivation

The Kalman Filter can be derived many ways. If you have tried researching this topic on the internet then you have probably come across different variations. I have done that same research myself and found the following derivation that makes the most sense to me.

What is being derived?

We are deriving the prediction equations, the optimal gain equation, and the estimation equations. These equations are shown in the Kalman Filter process diagram below in steps 3,4, and 5.



Assumptions

- The dynamic system under observation can be represented by a linear state space model
- Each estimate will be a linear combination of the predicted state and the new observed measurement
- All observed and non-observed state variables have gaussian distributions
- The Kalman Filter minimizes the mean square error

Prediction

As mentioned above, the Kalman Filter produces an estimate that is the linear combination of the predicted estimate and the current observation. In order to predict the previous estimate forward in time, we need to come up with equations that describe the systems behavior.

One of our assumptions is that the dynamic system under observation can be modeled as a linear state space model. The states of the system are time varying so the subscript of the state vectors, ***x***, will be denoted by ***t*** and ***t-1***, the system state at the current time and the system state at the previous time step, respectively.

Essentially what we are saying when we describe the system as a linear state space model is that the current state estimate is a linear combination of the previous state estimate and some system process noise drawn from a multivariate distribution, ***q***. Our gaussian distribution assumption above allows us to say that the system process noise is a gaussian distribution. The linear relationship between the estimate at ***t-1*** and the current estimate at time ***t*** is given by the ***A*** matrix. Together all of these assumptions lead to what we call the state equation.

A linear state space model can be defined with the following equation:

$$x_t = Ax_{t-1} + q_t$$

Where the respective distributions are defined as the following:

$$x_0 \sim \mathcal{N}(m_0, P_0)$$

$$q_0 \sim \mathcal{N}(0, Q_0)$$

Where $\boldsymbol{m_0}$ is the mean value of the initial multivariate state, ***x***. $\boldsymbol{P_0}$ is the covariance matrix of the multivariate distribution of the initial multivariate state, ***x***. And $\boldsymbol{Q_0}$ is the covariance matrix of the multivariate distribution from which the system noise comes from.

Given the linear state space model and the state variable distributions defined above, we can now define our prediction equations.

First, we will define the state estimate prediction equation as the following:

$$x_t = Ax_{t-1}$$

As you see this is almost exactly the same as the linear state space model except it does not contain the system noise. This makes sense right? Our prediction model is our system model, so we expect it to be similar. The reason we don't have the process noise added into this prediction step is because it is a zero mean distribution. Since its zero mean, the $\mathbf{q_t}$ will be 0 for this step.

Even though the process noise is zero mean, we still account for it in the predicted covariance equation. We account for it by adding the process noise covariance matrix to the predicted state covariance matrix as follows:

$$P_t = AP_{t-1}A^T + Q_t$$

We can see here that the predicted covariance is linear combination of previous estimates covariance matrix and the process noise.

Estimation

The estimation step has two parts to it: the state estimate and the state covariance matrix. Let us start with the state estimate.

One of our core assumptions is that the state estimate is a linear combination of the previous state estimate and the new measurement information. If you read Chapter 1, you will recall the exponential moving average filter. If you haven't read Chapter 1, take some time to read it if these next steps don't make sense.

The equation for the exponential moving average filter was a linear combination of the current mean and the new measurement. Here is that equation:

$$\bar{y}_t = \alpha\bar{y}_{t-1} + (1-\alpha)y_t$$

Where:

- **α** is the weight
- **t** is the update number
- **y** is the measurement value
- **y_hat** is the mean value

You can see that the previous mean value is scaled by alpha and that the new measurement value is scaled by the alpha value minus one. If alpha is zero, then the new mean is equal to the new measurement. If alpha is one, then the new mean is equal to the previous mean. And when alpha is between zero and one, then the both values contribute.

This same type of logic is applied when coming up with the state estimate equation. Here that equation is:

$$x_t = (I - KH)x_p + K_t z_t$$

Where:

- **x** is the state estimate column vector
- **K** is the optimal Kalman Gain matrix
- **z** is the measurement column vector
- **H** is the state to measurement matrix
- **t** is the subscript indicating the current time of the update

- **p** is the subscript indicating a variable that was predicted forward in time

You can see that this follows the same paradigm as the above. Since this formula is in matrix format, it looks slightly different. It's a linear combination of the previous state information and the current measurement. Its more common format is the following:

$$\boldsymbol{x_t = x_p + K_t(z_t - H_t x_p)}$$

Now let us talk about the state covariance.

$$P_t = cov(x_{truth} - x_t)$$

Where subscript truth refers to the true value of **x**. Substitute for $\mathbf{x_t}$ as we defined above:

$$P_t = cov(x_{truth} - [x_p + K_t(z_t - H_t x_p)]$$

Substitute for $\mathbf{z_t}$:

$$z_t = H_t x_t + v_t$$

$$P_t = cov(x_{truth} - [x_p + K_t(H_t x_t + v_t - H_t x_p)]$$

And by doing some algebra we can reduce this equation to the following:

$$P_t = cov[(I - K_t H_t)(x_{truth} - x_p) - K_t v_t]$$

Assuming the measurement error $\mathbf{v_t}$ is uncorrelated with other terms, and then using some properties of vector covariance, the equation can be rewritten as follows:

$$P_t = (I - K_t H_t) cov(x_{truth} - x_p)(I - K_t H_t)^T - K_t cov(v_t) K_t^T$$

Substitute in our definition for P_t and R_t

$$P_t = (I - K_t H_t) P_p (I - K_t H_t)^T - K_t R_t K_t^T$$

This is valid for any value of K. But If we determine the optimal value of K, we can simplify this equation. Lets find the optimal K and then come back to this.

Kalman Gain

The Kalman Filter is a minimum mean-square error estimator. The error in the posterior state estimation is as follows:

$$x_{truth} - x_p$$

As part of our assumptions, we want to minimize this error. This is equivalent to minimizing the trace of the estimates covariance matrix we just derived above. Remember that the trace of a matrix is the sum of the diagonal components. And the diagonal terms of the covariance matrix are the variance for each element of the system state.

Let's take another look at the covariance matrix we just derived.

$$P_t = (I - K_t H_t) P_p (I - K_t H_t)^T - K_t R_t K_t^T$$

Let's expand it and collect the appropriate terms:

$$P_t = P_p - K_t H_t P_p - P_p H_t^T H_t^T + K_t (H_t P_p H_t^T + R_t) K_t^T$$

Lets substitute in the innovation matrix, S_t:

$$S_t = H_t P_p H_t^T + R_t$$

$$P_t = P_p - K_t H_t P_p - P_p H_t^T K_t^T + K_t S_t K_t^T$$

In order to minimize we will take the derivative of the trace with respect to the gain matrix and set that to 0. Solving for the gain matrix will give us the optimal gain.

$$\frac{\partial\, tr(P_t)}{\partial\, K_t} = -2\left(H_t P_p\right)^T + 2K_t S_t = 0$$

Solve for K_t

$$K_t S_t = \left(H_t P_p\right)^T = P_p H_t^T$$

$$\boldsymbol{K_t = P_p H_t^T S_t^{-1}}$$

The optimal gain derived here will provide the minimum mean square error when used in the Kalman Filter.

Let us use this optimal gain equation to simplify the state estimate covariance.

Simplify State Estimate Covariance Matrix

Let us look back at the covariance matrix we derived:

$$P_t = P_p - K_t H_t P_p - P_p H_t^T K_t^T + K_t S_t K_t^T$$

If we multiply both sides of our optimal gain equation by $S_t\, K_t^T$

$$K_t S_t K_t^T = P_p H_t^T K_t^T$$

If we substitute the gain equation into the covariance equation, we get two terms that cancel out and simplify the equation.

$$P_t = P_p - K_t H_t P_p$$

$$\boldsymbol{P_t = (I - K_t H_t)\, P_p}$$

About the Author

This ebook is the work of William Franklin.

William is an experienced mechanical engineer who enjoys teaching, hobby projects, and using science to improve his day-to-day life.

William has a Bachelor of Science in Mechanical Engineering and a Master of Science in Mechanical Engineering.

His course concentrations were in Fluid and Mass Transfer and Manufacturing Technology.

He has a broad range of experience from working as a mechanical engineer. For instance, he has worked in the chemical processing, construction, and defense industries.

Currently, he works in the defense industry as a system's engineer that designs and analyzes Kalman Filter algorithms for target tracking problems.

Furthermore, he has also worked with Extended Kalman Filters and Unscented Kalman Filters.

William wants all engineers to be able to use the Kalman Filter in their engineering work because he believes it is a force multiplier.

William dedicates this book to his wife and family who have supported him throughout his life and believed he was capable of being an engineer.