Kalman Filter for SunSat

Why and How?

Rory Haggart

December 2020

1 Introduction

The SunSat avionics includes a dedicated subsystem for parachute deployment and attitude determination activities. Each of these two distinct tasks require some sort of state estimation based on the inputs from sensors during flight. This iteration of the Nova platform is using a passive parachute deployment system, but future iterations may use an active system. Active deployment should be triggered at the point of apogee in the flight to minimise loading and chances of damage, and for this to happen, sensor data needs to be processed appropriately such that the point of apogee can be detected to a sufficient degree of accuracy in real-time.

To estimate the states of a system, we would ideally use a huge suite of sensors to log every possible output over the period of interest. However, this is obviously a very complex, expensive, and sometimes impossible task. Instead, we want to be able to measure certain states of the vehicle by processing the data from existing, low-cost sensors. For example, we want to know the velocity of the satellite. We want to know velocity because we can then use the point of $u_z = 0$ to be the point of apogee (the vertical (z-axis) velocity is zero at the maximum height). However, we don't really have any practical means of directly measuring this.

- We can't use a rotary speed sensor - as used in cars - as we don't have any rotational motion that we can measure and translate into linear velocity
- We can't use a pitot tube - as used by aircraft - as we are enclosed in the rocket's payload section and so wind speed is not measurable.

We can, however, use an accelerometer. Acceleration is the first time derivative of velocity, and therefore, velocity is the integral of acceleration over the interval $[0, t]$.

$$a = \frac{du}{dt} \tag{1.1}$$

$$u = \int_0^t a \mathrm{d}t \tag{1.2}$$

And so if we want to measure the velocity of our satellite, surely we can just integrate the output from out accelerometer in the z-direction? That would be great if we were living in an ideal world, but sadly our measurements are quite far from ideal... We're just taking snapshots of the measurement at discrete time intervals. This means that when we integrate, we're missing out on a bunch of readings. In these readings there will be actual acceleration variation as well as a bunch of noise - such is life. So long story short, it's not quite as simple as just integrating our acceleration.

Additionally, it would be nice to have access to 'smoothed' or 'accurate' acceleration data. We know that the vehicle acceleration isn't actually varying as the output from our sensor tells us it is - the noise is giving us a false impression of what the vehicle is actually doing.

A Kalman Filter will (hopefully) help us solve these problems! ... but what is a Kalman Filter? I'll try to explain this without getting too far into the nitty gritty of the control theory.

2 What is a Kalman Filter?

Lets say we had a **perfect** model of our system. We modelled every single particle of fuel in the rocket motor, accounted for the gravitational pull of Jupiter, and even managed to mathematically describe the current status of the entire atmosphere, allowing us to predict in which direction the wind would be blowing in a few months time. Using all of this information, we should be able to know the state (consisting of acceleration, velocity, position, etc.) of our vehicle at some arbitrary time without even having to use any sensors! But back in the real world, this is completely ridiculous, of course.

Instead, we typically model our systems in some simplified way. Maybe we just consider our vehicle as a point mass, or neglect the change in air density as altitude increases, for example. This way, we can sensibly approximate our system, and we can use this simplified model to our advantage. By itself, this would allow us to at least make a guess about the acceleration of our vehicle at an arbitrary point in time. But remember, we have sensors to help us out! We can compare the acceleration as measured by the accelerometer to our guess based on the simplified model to refine our estimate and hopefully bring it as close to reality as possible. That's handy! It would also be helpful if we could account for the fact that we're not just going to be flying straight up - uncertainties in the environment (e.g. wind) are going to be pushing us around in some 'random' way that we aren't able to predict.

And **that** is what a Kalman Filter does for us! It is a way of using imperfect sensor data in a system full of uncertainties, along with a simplified model of the dynamics, in order to estimate the *actual* state of the system. If you'd like to look through some of the mathematical details, please feel free to read through the below section! If you'd rather not (can't blame you), then skip ahead to Chapter 3

Details

Okay so that's the more fluffy description out of the way. This section will be a little more specific and try to define the structure in more strict terms. The first implementation will be outlined in Chapter 4. I want to quickly highlight that the rest of this section is heavily based on a very intuitive guide to the Kalman Filter which can be found at https://www.bzarg.com/p/how-a-kalman-filter-works-in-pictures/. I have modified and simplified the description quite significantly but the flow is very much the same.

For the sake of illustration, let's limit our state variables to acceleration and velocity in the z-direction only. To more formally describe this, we'll define our state vector as:

$$\vec{x} = \begin{bmatrix} a_{\mathrm{z}} \\ u_{\mathrm{z}} \end{bmatrix} \tag{2.1}$$

Now, we can use a simplified model of our system to get an initial 'guess' for the actual acceleration of the vehicle - we know that it isn't necessarily accurate due to uncertainties in the system. The Kalman Filter will interpret our estimate as a random variable with a Gaussian noise distribution, which introduces uncertainty. We can say that the 'most likely' value of the variable is at the centre of the Gaussian distribution (we'll call this point μ), and that the uncertainty surrounds this centre point (the variance of the measurement σ^2).

In this case, the acceleration and velocity in the z-direction are directly correlated (i.e. an increase in measured acceleration necessarily means velocity has increased too). The way that the probability distributions of each state variable are correlated is described by an *error covariance matrix* (usually denoted P, with elements Σ_{ij} denoting the covariance between variables i and j).

$$P_{\mathrm{k}} = \begin{bmatrix} \Sigma_{\mathrm{a_z a_z}} & \Sigma_{\mathrm{a_z u_z}} \\ \Sigma_{\mathrm{u_z a_z}} & \Sigma_{\mathrm{u_z u_z}} \end{bmatrix} \tag{2.2}$$

Our current state estimate is at point $k-1$ in time. Now with the above knowledge of the system, the Kalman Filter will try to determine a new probability distribution for the system state at time k. By applying some matrix, F_{k}, to the current state, it can transform the old distribution into a new distribution. If $\hat{x}_{\mathrm{k\text{-}1}}$ is our current state estimation, then we can describe our new estimation as:

$$\hat{x}_{\mathrm{k}} = F_{\mathrm{k}}\hat{x}_{\mathrm{k\text{-}1}} \tag{2.3}$$

In some systems, the relationship between variables may change over time, but with velocity and acceleration being directly related, we can assume that P is time invariant:

$$P_{\mathrm{k}} = P_{\mathrm{k\text{-}1}} = P \tag{2.4}$$

We might also have some information about the inputs to the system (again, the specifics will be described in Chapter 4), and we can define these as the control matrix u_{k} (not to be confused with velocity, u!). This then allows us to account for new inputs to the system which might change its state.

But then what about the uncertainties that we described earlier? For example, the wind might impart some force on the vehicle that we should really account for! The wind would actually modify the distribution of our state estimation (yet we can still assume it to be Gaussian), and so we model this as our *process noise.*

Using all of this information, we can make a pretty decent guess about the new state of our system at time k. To really hone in our predictions, we can then use sensor measurements to improve our accuracy. But again, these are full of noise (and guess what? we're going to assume it's Gaussian!). This noise is then the *measurement noise*, and is associated with the measurements of the state, not the state itself.

The data sheet of a sensor will typically give us information about the measurement noise, and so this is relatively simple to put in to our model. Process noise is a bit more difficult, as it really depends on the system. Often it has to be a case of trial and error, but sometime you might be able to look at previous examples of similar systems.

With all of this taken into account, we essentially have two estimates for our state. One as predicted by our modelling of the system and it's evolution over time, and another as shown to us by our noisy sensors. Combining the two can really refine our estimate to something sensible and (hopefully) accurate.

The block diagram for the Kalman Filter is illustrated in Figure 2.1.

And now considering the equations that describe our filter (note that these are descriptive equations, but are not the ones that we'll use for implementation):

System Model:

$$x_{\mathrm{k}} = F_{\mathrm{k}} x_{\mathrm{k\text{-}1}} + B_{\mathrm{k}} u_{\mathrm{k}} + w_{\mathrm{k\text{-}1}} \tag{2.5}$$

Measurement Model:

$$z_{\mathrm{k}} = H_{\mathrm{k}} x_{\mathrm{k}} + v_{\mathrm{k}} \tag{2.6}$$

x :	the system state vector,
F_{k} :	the transformation from the state at $k-1$ to the state at k,
B_{k} :	the control input model matrix
u_{k} :	the control input vector
w :	a term representing the entry of process noise into the system,
z :	the measurement,
H_{k} :	the transformation from the system state into the measurement,
v :	a term representing the consideration of measurement noise

Now with these models defined, we can look at how the states are estimated mathematically. State estimation works in two phases - *predict* and *update*. The *predict* phase involves using the previous state estimate and the underlying dynamical model to generate an estimate for the current state of the system. Then, the *update* phase uses sensor measurements to refine the current state estimate. In this phase we 'inject' the process and measurement noise into the estimation through a gain term, K.

The prediction of the state at k, using the information of the state at $k-1$ is defined as in Equation 2.7. Note that the mathematical details are omitted here for some level of simplicity.

$$\hat{x}_{\mathrm{k|k\text{-}1}} = F_{\mathrm{k}}\hat{x}_{\mathrm{k\text{-}1|k\text{-}1}} + B_{\mathrm{k}}u_{\mathrm{k}} \tag{2.7}$$

Which basically tells us that we are estimating the state by considering a transformation of the previous state plus some sort of known control input.

And then the update using sensor measurements is defined as in Equation 2.8. Again, some details are omitted for simplicity.

$$\hat{x}_{\mathrm{k|k}} = \hat{x}_{\mathrm{k|k\text{-}1}} + K_{\mathrm{k}}[z_{\mathrm{k}} - H_{\mathrm{k}}\hat{x}_{\mathrm{k|k\text{-}1}}] \tag{2.8}$$

And this tells us that we should update our estimate by taking our prediction, and adding a term that considers the error between our measurement, z, and what we expect, $H_{\mathrm{k}}\hat{x}_{\mathrm{k|k\text{-}1}}$. Applied to this second term is a gain, K, which is our 'Kalman Gain'. The Kalman Gain is essentially a measure of how much we 'trust' our measurements as compared to our predictions. For example, if we had a particularly inaccurate model, but really high quality sensors, then K should be quite high, as we have more trust in our sensors than we do in our predictions.

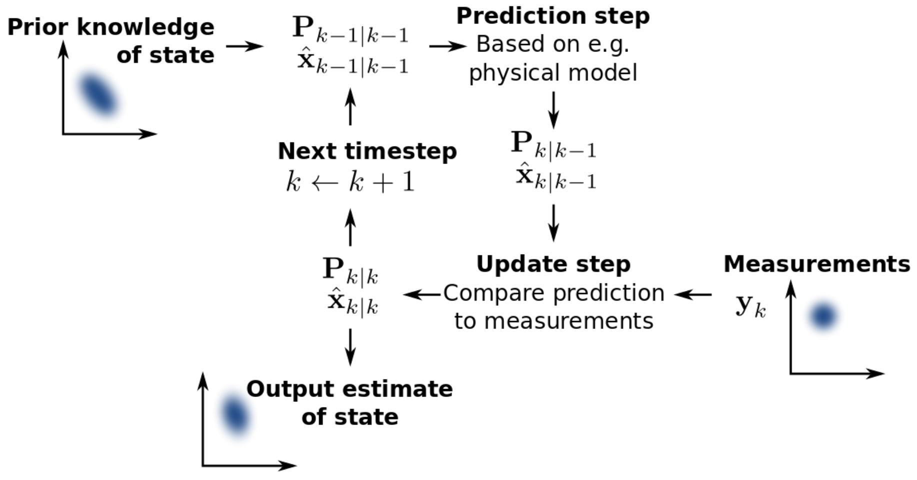
Prior knowledge of state
$\mathbf{P}_{k-1|k-1}$ $\hat{\mathbf{x}}_{k-1|k-1}$
Prediction step
Based on e.g. physical model
$\mathbf{P}_{k|k-1}$ $\hat{\mathbf{x}}_{k|k-1}$
Next timestep
$k \leftarrow k+1$
$\mathbf{P}_{k|k}$ $\hat{\mathbf{x}}_{k|k}$
Update step
Compare prediction to measurements
Measurements
$\mathbf{y}_k$
Output estimate of state

Figure 2.1: A generalised block diagram of a system implementing a Kalman Filter

Strictly speaking, the Kalman Gain is time variant, as its optimal value is one which minimises the residual error at each time step.

The time variant definition of K is:

$$K_{\mathrm{k}} = P_{\mathrm{k|k\text{-}1}} H_k^T [H_{\mathrm{k}} P_{\mathrm{k|k\text{-}1}} H_k^T + R_k]^{-1} \tag{2.9}$$

and really this isn't too difficult to calculate if we know all the components. BUT, we are working on an embedded system with real time constraints - matrix operations can be very time consuming, especially if we have a large number of state variables, which will increase the dimensions of all of these matrices. Since H just describes the mapping between our states and our measurements, it is time invariant. In Equation 2.4 we decided that keeping P (the covariance matrix which estimates the accuracy of our estimates) time invariant would simplify our implementation significantly. And R (which is the covariance of the measurement noise), is also time-invariant.

These assumptions form the basis of the 'steady-state Kalman Filter', which is the case when sensor covariances are time-invariant and the underlying models are time-invariant (i.e. the equations do not change over the operation of the system). Some other conditions that must be true for the steady-state condition to be valid are that the state variables must be observable, the model must be linear. Using a steady-state Kalman Gain allows us to calculate the Kalman Gain at the beginning of operations one time only, and then use this value for every iteration of the filter, drastically reducing the computational complexity.

The steady-state Kalman Gain can be recursively computed as follows (verification and citation needed!):

$$K = PH^T[HPH^T + R]^{-1} \tag{2.10}$$
$$P = [I - KH]P \tag{2.11}$$
$$P = FPF^T + Q \tag{2.12}$$

Where after a small number of iterations, K will converge to a constant value.

And so now we're pretty much ready to actually start implementing some of this in the PDC! As I keep saying, Chapter 4 will go into even further specifics as we start modelling the system and accounting for putting all of this together, but I hope this section served as a decent 'technical introduction'.

3 So why should we use a Kalman Filter?

Properly designed Kalman Filters do a pretty great job of estimating the system states. They are able to account for dynamic changes in the system that might have completely slipped your mind during the planning phase. Obviously there are limitations, and for sure this iteration will not be perfect, but that's why we test and develop these things! Especially at this point in which the apogee detection and attitude determination subsystems are not mission-critical, it is the perfect time to test new things out.

The Kalman Filter will allow us to process the data from our sensors and turn it into something much more useful at a low cost. We won't need a fancy sensor suite, and hopefully it will tell us a lot about the overall performance for analysis after the flight. It's also very computationally lightweight. It doesn't require much memory as you only need to keep hold of information about the current and previous states, and the calculations at each time step are not overly complex (though this depends on the model to an extent...).

In some applications, a Kalman Filter might be a bit overkill as we'd only have one noisy sensor and some measurement that we want out of it. In these scenarios, a more rudimentary filter might suffice (e.g. low-pass filter to smooth out higher frequency noise). Here, we have a pretty time-critical decision to make, and one preventative factor against us using a low-pass filter is the time delay that they introduce - we could miss the point of apogee by several seconds depending on our smoothing window!

With all of this in mind, the Kalman Filter is a perfect candidate for getting as much useful data as we can. It's a method that is well suited for high-speed embedded systems, and one that is used frequently in space systems (even in the Apollo guidance computers!).

4 Kalman Filter as a part of the parachute deployment activities

This section is going to try and get right into the details of the implementation of using a Kalman Filter to obtain a good estimate of velocity from the accelerometer during ascent. The basic premise is this:

- Derive a simplified mathematical model of the ascent phase of the rocket
- Collect information about the characteristics of the different sensors
- Unify all of this information into a Kalman Filter for the ascent phase

So first, we have to model our system. There are two main phases during ascent: the propelled phase, and the coast phase. The former occurs between lift off and engine burnout, and the latter begins once the engine is burnt out, and lasts until apogee.

Propelled phase

During the propelled phase, the vertical acceleration can be approximated by considering the vertical components of thrust and drag, and the gravity vector. A really important consideration here is the simplifications that can be reasonably applied to this system. The model should be linear as the Kalman Filter is a linear estimator, though the system is non-linear due to drag and rotational kinematics. However, a non-linear model would require an Extended Kalman Filter which will significantly increase computational load.

One nice part of the Kalman Filter is that we can really simplify a model, as long as we correspondingly increase the process noise matrix. Obviously this is less ideal than a more accurate model for which we can decrease the process noise (and therefore reduce the probability distribution of our estimates), but in terms of computational efficiency and simplicity, a cruder model is probably the better option.

Reasonable assumptions in the modelling of the propelled phase:

- **The rocket maintains vertical-pointing attitude.** In reality the angle of flight will deviate from vertical, but this should be minimal enough to neglect, and the prediction of these angles would be near impossible to model.
- **Lift forces can be ignored.** Due to the above assumption, the lift forces that might be expected in non-vertical flight can be disregarded.
- **The gravitational acceleration vector remains constant.** In reality, the magnitude of the downward acceleration due to gravity will decrease as the rocket gains altitude, but the reduction is nominally $\Delta g < 0.05ms^{-2}$.

The next section will describe the use of a dynamical model that assumes constant acceleration (downward, due to gravity). For this purpose it should provide a decent approximation near apogee, but the estimate during this propelled phase may deviate significantly from reality due to the obviously variable acceleration. One thing that should be kept in consideration is the application of different models for the propelled and coast phases. This could then involve approximating the acceleration of the rocket in a linear way during the propelled phase, but it should be worth noting that we then do not have an LTI system that can be altogether approximated by the steady-state Kalman Gain approximation, and so extra steps would be required in computing a new gain for this phase, and this may not be worth the effort at this stage.

Coast phase

The coast phase is really the phase of interest. This is where apogee will occur, and so the accuracy of Kalman Filter estimates are most important here. Luckily, this is where we can further simplify our model. Since drag varies with the square of the velocity (making it a non-linear component of the system), it will have a very low magnitude near apogee as the velocity approaches zero. For this reason, our underlying dynamical model of the system can be linearised by neglecting drag. This may result in poor prediction during the propelled phase, but consider two things:

- We do not **need** super accurate predictions during the propelled phase. Sure, it would be nice, but it is not crucial.
- Our defined process noise and sensor measurements will help improve upon the estimations provided by the model alone, and so hopefully we will see a decent estimation in any case.

With all of this in mind, we can define the underlying mathematical model for this implementation.

In a constant acceleration model, the only force acting on our vehicle is the downward acceleration due to gravity ($g = 9.81ms^{-2}$). The governing equations of motion in the z-direction are then very simple, as we can just use the so-called $SUVAT$ equations for kinematics under constant acceleration to define acceleration (a), velocity (u), and position (s).

$$a_{\mathrm{t}+\Delta \mathrm{t}} = a_{\mathrm{t}} \tag{4.1}$$

$$u_{\mathrm{t}+\Delta \mathrm{t}} = u_{\mathrm{t}} + a_{\mathrm{t}}\Delta t \tag{4.2}$$

$$s_{\mathrm{t}+\Delta \mathrm{t}} = s_{\mathrm{t}} + u_{\mathrm{t}}\Delta t + a_{\mathrm{t}}\frac{\Delta t^2}{2} \tag{4.3}$$

Sensor Information

In REF we showed that for apogee detection, an accelerometer alone would not have allowed us to use a linear, steady-state approximation of our model. For this reason, we are using the accelerometer in combination with the altimeter to make our system state variables observable. We need to account for the noise in these sensors in our model, and so here we will outline the important characteristics of the selected sensors.

The subsystem uses a *BMP388* altimeter, and an *LSM6DSO32* accelerometer. The data sheets for these sensors provide information about the

Sensor	RMS Noise	Variance	Operating Conditions
BMP388 (pressure)	0.6Pa	0.36Pa^2	IIR Coefficient = 8; high resolution mode
BMP388 (temperature)	0.005°C	0.000025°C^2	high resolution mode
LSM6DSO32	0.0032g	0.01024g^2	$\pm 4g$ range; normal/low-power mode

Table 4.1: Some limited noise information about the sensors in use.

noise in their measurements. The amount of noise varies between operational modes, so these are clarified in Table 4.1. Note that the pressure and temperature are both included here - the altitude calculation will rely on them both, and so the noise in each reading should be accounted for. Also note that the measurement noise can vary quite significantly, and so maybe the values should be kept in an enumeration so that when the sensors are configured at startup, their noises are auto-fetched. Alternatively, there could be a portion of setup that averages the measurements over a period of time and determines the noise on the signal.

The RMS noise is the square root of the variance in the case of no offset, and this value is also presented in the table.

Now using this information, we can construct the measurement noise covariance matrix with the knowledge that the two sensors are completely independent, meaning that the matrix will be diagonal. Using the values from Table 4.1, we end up with the following matrix:

$$R = \begin{bmatrix} \sigma_a^2 & 0 \\ 0 & \sigma_s^2 \end{bmatrix} = \begin{bmatrix} 0.01024 & 0 \\ 0 & \sigma_s^2 \end{bmatrix} \tag{4.4}$$

The Kalman Filter

Looking back to Equation 2.3, we have already described our basic principle of estimating the next state based on our current state. And we can define our state vector as:

$$x = \begin{bmatrix} a_\mathrm{z} \\ v_\mathrm{z} \\ s_\mathrm{z} \end{bmatrix} \tag{4.5}$$

which is to say that we are interested in the z-direction acceleration, velocity, and position. We defined these relations in Equations 4.1, 4.2, and 4.3, which we can use to create our F matrix. This allows us to define a state space description of the system, based on Equation 2.5:

$$\begin{bmatrix} a_\mathrm{z} \\ v_\mathrm{z} \\ s_\mathrm{z} \end{bmatrix}_{k+1} = \begin{bmatrix} 1 & 0 & 0 \\ \Delta t & 1 & 0 \\ \Delta t^2/2 & \Delta t & 1 \end{bmatrix} \begin{bmatrix} a_\mathrm{z} \\ v_\mathrm{z} \\ s_\mathrm{z} \end{bmatrix}_{k} + v_\mathrm{k} \tag{4.6}$$

And then in our measurement space, we can simply define our H matrix as a mapping between our measurements (acceleration and position) and our states (acceleration, velocity, and position). This form is based on Equation 2.6.

$$\begin{bmatrix} a_\mathrm{z} \\ s_\mathrm{z} \end{bmatrix}_{k} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a_\mathrm{z} \\ v_\mathrm{z} \\ s_\mathrm{z} \end{bmatrix}_{k} + w_\mathrm{k} \tag{4.7}$$

With this information, we can define the 'predict' and 'update' steps as we outlined in Equations 2.7 and 2.8, with the note that we are ignoring the thrust input, and so $u_k = 0$. Remember that K is a gain matrix calculated at initialisation through a recursive process detailed in Equation 2.10, and using the dimensions of the rest of the matrices, we can see that K must be a 3×2 matrix.

Predict:

$$\widehat{\begin{bmatrix} a_{\mathrm{z}} \\ v_{\mathrm{z}} \\ s_{\mathrm{z}} \end{bmatrix}}_{k+1} = \begin{bmatrix} 1 & 0 & 0 \\ \Delta t & 1 & 0 \\ \Delta t^2/2 & \Delta t & 1 \end{bmatrix} \widehat{\begin{bmatrix} a_{\mathrm{z}} \\ v_{\mathrm{z}} \\ s_{\mathrm{z}} \end{bmatrix}}_{k} \tag{4.8}$$

Update:

$$\widehat{\begin{bmatrix} a_{\mathrm{z}} \\ v_{\mathrm{z}} \\ s_{\mathrm{z}} \end{bmatrix}}_{k} = \widehat{\begin{bmatrix} a_{\mathrm{z}} \\ v_{\mathrm{z}} \\ s_{\mathrm{z}} \end{bmatrix}}_{k-1} + K_{\mathrm{k}} \left[\begin{bmatrix} a_{\mathrm{z}} \\ s_{\mathrm{z}} \end{bmatrix}_{k} - \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \widehat{\begin{bmatrix} a_{\mathrm{z}} \\ v_{\mathrm{z}} \\ s_{\mathrm{z}} \end{bmatrix}}_{k-1} \right] \tag{4.9}$$