Further Notes on Anti-Gravity Filtering

Combining DSP and state estimation techniques to remove gravity from accelerometer readings.

Abstract

For slow moving systems, removing gravity by just straight up filtering out low-frequency components of an accelerometer signal works surprisingly well, but also introduces some defects that—without correction—make acceleration data useless for real-time applications. Starting out with some basic filters, we then correct for defects using straightforward state estimation via an Extended Kalman Filter. Examples via MATLAB code included in the appendices.

When walking along the surface of the Earth, how fast does the force due to gravity change? What about when you're driving? What about when you're standing still, but waving your phone around in front of your face because you're playing a mobile game? What if you're using a mobile phone to measure acceleration while driving a car, but the acceleration due to gravity measured by the phone does not accurately reflect the movement of the car? What if you can't use standard attitude and heading reference system (AHRS) techniques[1] to figure out which way gravity is pointing (down), so that you can't easily correct for it?

At first glance, most people would say that gravity just doesn't change that quickly, and that on the surface of the earth it might mostly be constant. Therefore, to correct for it, you can just filter out all DC components from any acceleration measurements. However, while this is technically correct, it's not at all true. It turns out, accelerometers are not measuring gravity, they are measuring the force reacting to gravity by whatever they're placed on or attached to. What this means, then, is that when you're trying to use an accelerometer to figure out where gravity is pointing, you have to take into consideration that the force due to gravity is changing as fast as your accelerometer is moving. Which is different when driving in cars compared to standing still and waving a phone around.

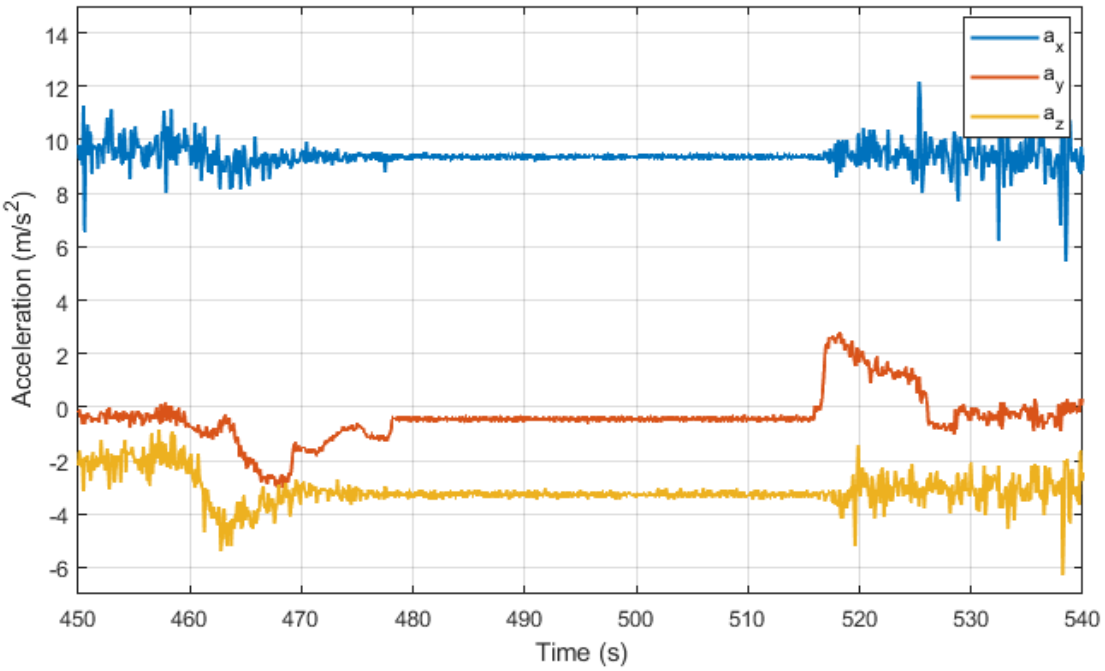
What *this* means, is that if you can't use AHRS techniques to figure out the orientation of your sensors in order to remove gravity, you have to get creative—and specific—about what to do. So, let's be specific:

1. I'm trying to measure the movement of a car using a smartphone,
2. Smartphones are not rigidly attached to cars,
3. Smartphones have built in GPS that measure position, speed, & heading,
4. Smartphones have built in inertial measurement units, IMUs, that measure acceleration, angular velocity, and the Earth's magnetic field,
5. I have to be careful how I use GPS, and can't use the magnetic field measurements,
6. Everything is very low frequency; ability to calibrate / initialize measurements is limited.

Numbers two and six above really limit how we can approach getting good use of this data. For example, if the smartphone is bouncing around in the car while it's moving, you're going to pick up a lot of spurious signals that don't really represent the motion of the car. Then, if you're sampling these signals at a relatively slow rate like 10 Hz, you can't exactly go around using 80-tap finite impulse response (FIR) low pass filters and get a linear phase response. Of course, numbers three to five limit how you can use fancier kinds of filters.

Let's see what we can do with some basic low-order infinite impulse response (IIR) filters that can be used to remove gravity. Since we'll assume that gravity is a very slow moving, low-frequency signal, there are two basic approaches that we can take: 1) high-pass filter the data so that extremely low frequency components are removed directly from it; 2) low-pass filter the data to get a copy of just extremely low frequency components to subtract from our signal as required. To get an idea of what extremely low frequency means, let's take a look at a few seconds' worth of data:

[1] https://ahrs.readthedocs.io/ is an excellent resource, including both theory and code!



This is ninety seconds worth of data from the accelerometer on my smartphone while I drive around in my car. What we're looking at is the car slowly coming to a stop over the course of 20 seconds from 460 to 480, staying stopped at a red light for about 35 seconds from 480 to 515, and then slowly getting up to speed from 515 to 525. The majority of this linear acceleration is taking place along the y-component of the measured acceleration.

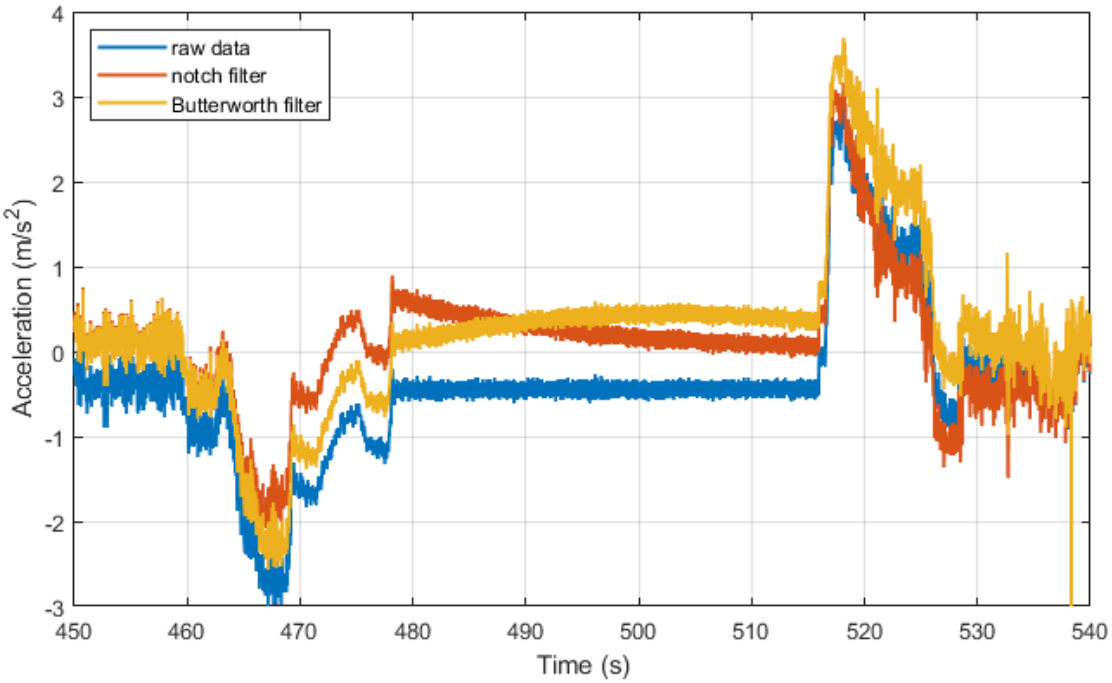
Most smartphones align their sensors so that the x-coordinate is aligned along the short axis of the phone, y aligned along the long axis, and z is normal to the plane of the phone. So, looking at this, you can tell that my phone is lying on its side, with the long axis of the phone mostly aligned along the driving axis of my car. This means that the x-axis is pointing mostly up, so as you can see most of gravity is captured in that direction (that is, the car pushing against the phone to resist gravity), while the y-axis captures linear acceleration (coming to a stop, then starting up again after half a minute), with the z-axis capturing some rotation between the 460 and the 470 second mark.

As you can see, each component of this acceleration has at least a little bit of a DC offset, which actually does add up to about the force of gravity near me, if just a little bit higher. Ironically, DC biases that aren't gravity, but sensor error, aren't an issue for us as we want to remove all extremely low-frequency signals anyway. So let's try two naïve approaches to see what we can get:

1. Use a basic notch filter with a notch frequency of 0 Hz and cut-off frequency of 0.01 Hz to remove low-frequency components from our signal,
2. Use a third-order Butterworth filter with a cut-off frequency of 0.01 Hz to select the low-frequency components of our signal, and then subtract them from our raw signal.

All of these values will need to be, "tuned," for your specific use case, and I picked these because they seemed about right for my use case, and then it just ended up working.

Focusing on the acceleration data just for the y-component of the acceleration—as that's where most of the true linear acceleration of the car / smartphone seems to be—so that we can also see how these filters distort a real signal, we get the following results:



Just to be clear, the blue line is the y-component of the acceleration data from the previous figure; the red line is the blue line after filtering with a basic notch filter; and, the yellow line is the blue line after filtering with a Butterworth filter, subtracting that result from the blue line. These results are pretty good! First, you can tell that the notch filter is great at removing DC signals, so much so that it leaves in very slow-moving sinusoidal signals that result in some distortion. That is, notice how the red is adjusted too far upwards, making some of what we know to be deceleration appear to be positive acceleration (as the car comes to a stop). Similarly, as the car starts to accelerate, it can't do so fast enough, showing around the 530 second mark some acceleration as deceleration. However, when the car is stopped from 475 seconds to 515 seconds, the acceleration data is allowed to, "relax," to a true value of zero m/s/s.

The usefulness of also removing slightly different very low frequency sinusoidal signals is demonstrated by the Butterworth filter approach. Looking at where the car comes to a stop, and then starts back up again, you can tell that it better aligns with the raw data, but shifted up by a constant amount. However, because there are some slow-moving sinusoids in our filtered gravity signal, when we subtract the result of the Butterworth filter from the raw data, we add them back in. What this means is that it's possible for acceleration readings to be non-zero when there should be exactly zero acceleration.

Regardless of which way you choose to filter this data, neither are truly suitable for direct use in any real-time scheme if you want to understand, say, linear speed along the axis of the car. Both of these methods still show acceleration where there should be none. However, because we're using smartphones and their sensors to measure things, we can take advantage of the additional information that we have access to via GPS: scalar speed, scalar distance, or total distance travelled. Since we can know when the car is not moving via its speed, we should be able to adjust our acceleration data so that it better matches reality. Because we'd like to compare vector acceleration with scalar speed, the easiest way to do that is with an extended Kalman filter.

As a quick reminder, a Kalman filter is an optimal way to improve the measurements of a system using knowledge of how it evolves through time, and the relative uncertainties of your measurements compared to the uncertainty introduced by this time evolution. So if we have measurements of acceleration and speed, we can clean them up by using the fact that acceleration is just the time rate of change of speed. We can do this

through the three phase Kalman filter approach of: 1) predict; 2) weigh; 3) estimate. First, we predict the current value of our state with:

$$\mathbf{x}_p = f(\mathbf{x}')$$

Where $\mathbf{x}_p$ is the predicted state, **x**' is the state estimate at a previous time step, and *f* is a function that uses some kind of physical process model that we use to make our predictions. We then linearize this with a Jacobian to get a state transition matrix:

$$\mathbf{F} = \frac{\mathrm{d}f(\mathbf{x}')}{\mathrm{d}\mathbf{x}'}$$

Which we can then use to predict the uncertainty in our predictions:

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

Where **P**' is the uncertainty of the final estimates in the previous time step, and since we have no control inputs (we're just measuring things with a smartphone sensor), $\mathbf{Q}_a$ captures all of the uncertainty in our time evolution model, which in our case will be a constant acceleration model. We can then compare the measurements, **z**, with the predictions:

$$\mathbf{y} = \mathbf{z} - h(\mathbf{x}_p)$$

Where *h* is a function that translates our predictions into the domain of the measurements so that they can be directly compared. This function is then linearized, again with a Jacobian:

$$\mathbf{H} = \frac{\mathrm{d}h(\mathbf{x}_p)}{\mathrm{d}\mathbf{x}_p}$$

So that we can use it to translate the state uncertainty predictions in the domain of measurement, and properly weighed by finding the Kalman gain, **K**:

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

Where **R** represents the uncertainty in the measurements. From this, we can then find mostly-optimal estimates of our state, and its uncertainty, at our current point in time:

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

$$\mathbf{P}_e = (\mathbf{I} - \mathbf{K}\,\mathbf{H})\,\mathbf{P}_p\,(\mathbf{I} - \mathbf{K}\,\mathbf{H})^{\mathrm{T}} + \mathbf{K}\,\mathbf{R}\,\mathbf{K}^{\mathrm{T}}$$

To start off, we're just going to keep it simple: we'll estimate a vector acceleration and velocity, comparing them to vector acceleration measurements and a scalar speed. All uncertainties will be set to one (so predictions and measurements will have equal weighting), except for the uncertainty in acceleration, which will depend on speed. This means that our prediction step will look like:

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

Which shows a simple constant acceleration model, where velocity vectors are accumulated from acceleration. Linearizing this with a Jacobian gets us:

$$\mathbf{F} = \begin{bmatrix} \mathbf{I}_3 & \Delta t\, \mathbf{I}_3 \\ \mathbf{0}_3 & \mathbf{I}_3 \end{bmatrix}$$

Where, since we're using a constant acceleration model, we use a model uncertainty of:

$$\mathbf{Q}_a = \begin{bmatrix} \mathbf{0}_3 & \mathbf{0}_3 \\ \mathbf{0}_3 & \sigma_a^2\, \mathbf{I}_3 \end{bmatrix}$$

Then, since we're measuring speed and vector acceleration, our measurement vector, **z**, is given by:

$$\mathbf{z} = \begin{bmatrix} v_z \\ \mathbf{a}_z \end{bmatrix}$$

Where v_z is the speed reading of our GPS, and $\mathbf{a}_z$ is the *filtered* accelerometer data; that is, the data we get after we low-pass filter acceleration data, subtracting the result from the raw data. Our measurement prediction function, *h*, is then simply:

$$h(\mathbf{x}_p) = \begin{bmatrix} |\mathbf{v}_p| \\ \mathbf{a}_p \end{bmatrix}$$

Which can be linearized in a straightforward way with another Jacobian to be:

$$\mathbf{H} = \begin{bmatrix} \frac{\mathbf{v}_p^{\mathrm{T}}}{|\mathbf{v}_p|} & \mathbf{0}_{1\mathrm{x}3} \\ \mathbf{0}_3 & \mathbf{I}_3 \end{bmatrix}$$

Also, since all of our measurements are independent of one another (GPS and accelerometers are different devices, the 3 axes of an accelerometer are orthogonal), our measurement uncertainty can be given as:

$$\mathbf{R} = \begin{bmatrix} \sigma_{vz}^2 & 0 \\ \mathbf{0}_3 & \sigma_{az}^2\, \mathbf{I}_3 \end{bmatrix}$$

Where the relative uncertainty in the process model (constant acceleration), and speed measurements are the same, one:

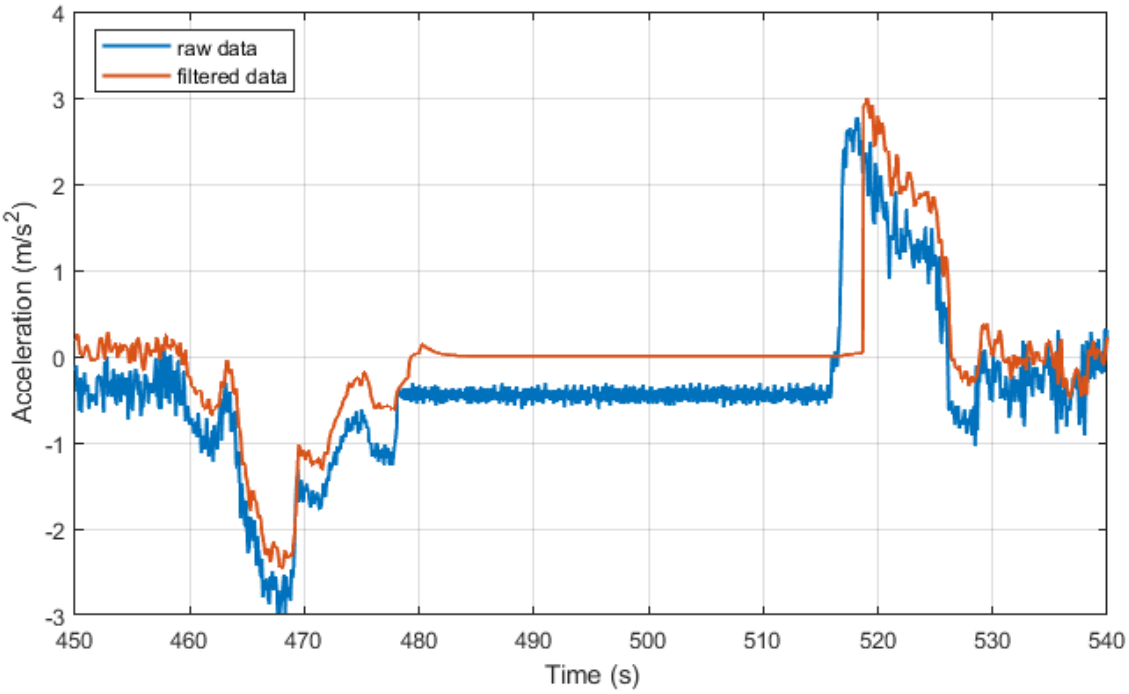
$$\sigma_a^2 = 1$$
$$\sigma_{vz}^2 = 1$$

To get the results we want though, the uncertainty in acceleration measurements will be dependent on the speed measured in such a way that we effectively ignore the acceleration measurements when travelling at a slow speed:

$$\sigma_{az}^2 = \begin{cases} 1 & |v_z| > 1 \text{ m/s} \\ 10,000 & |v_z| \leq 1 \text{ m/s} \end{cases}$$

It seems more complicated than it actually is, I promise. All we need to know is that the way things are set up here, it means that all measurements (speed and acceleration) and processes (constant acceleration, linear velocity) will have the same weight, except when speed is equal to or below 1 m/s/s, at which point the uncertainty in the acceleration measurements will become so high that they're effectively ignored. The inputs to this filter will be the current speed, as well as filtered acceleration readings, and the output will be a velocity vector and further filtered acceleration vector. Code for this specific filter can be found in the `ekf_va.m` file located somewhere near where this document is stored, but also in Appendix A.

After filtering the accelerometer data and throwing it through this filter, we get an output that looks like:



Which is actually pretty good! One thing we didn't mention before is that our sensor readings are coming in at different times, and at different rates. So, e.g., while our accelerometer data comes in at more or less 10 Hz, GPS tends to be capped at 1 Hz, meaning our speed measurements stay constant over one second intervals as we update our estimates using the filters every time we get a new accelerometer reading. This also means that when speed is low, we ignore accelerometer measurements, even if reality is different in real time. And that's exactly what we see just before the 520 second mark, where the initial acceleration is truncated for a couple of seconds before speed clears the 1 m/s threshold. We also managed to low-pass filter the acceleration data a little bit, simply because for estimates that have direct measurements, Kalman filters act as low-pass filters for that data. Oh, for some reason I decided to give this one a name, and call this a clamping filter, because it strongly clamps inaccurate measurements to follow more accurate ones.

Now, is there a way we can improve upon this?

I've recently become convinced by a very old paper[2] that Kalman filters are just generalized, adaptive complementary filters. And while discrete-time Kalman filtering can only every let you reproduce first order filters (e.g. basic low-pass / high-pass), time-varying noise models let you adjust filter parameters in real time to let you deal with all sorts of defects. So, without using any pre-filtering, can we design a simple extended Kalman filter that removes gravity, compensates for low-pass filter defects, and perhaps even cleans up our signal a bit?

The first step is to design a process model that takes into account that gravity is mostly constant, and is an additive defect to the linear acceleration of the car. That's actually relatively straightforward, it's the standard kinematics acceleration accumulation, but we estimate a separate gravity vector, subtracting that from our estimated acceleration to find a velocity vector:

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

As an aside, the estimated acceleration vector here is an estimate that matches the accelerometer measurements meaning that it still includes gravity. My first instinct was to estimate the acceleration vector to be free of gravity, combining an estimated gravity vector to compare to measurements; however, that doesn't allow you to use your velocity estimate (and thus speed measurement) to help constrain the gravity vector, and you get wacky results.

From here, we can get a state transition matrix:

$$\mathbf{F} = \begin{bmatrix} \mathbf{I}_3 & \Delta t\,\mathbf{I}_3 & -\Delta t\,\mathbf{I}_3 \\ \mathbf{0}_3 & \mathbf{I}_3 & \mathbf{0} \\ \mathbf{0}_3 & \mathbf{0}_3 & \mathbf{I}_3 \end{bmatrix}$$

And since we're using a constant acceleration—and constant gravity—model, our model uncertainty (and thus process uncertainty) is defined by:

$$\mathbf{Q}_a = \begin{bmatrix} \mathbf{0}_3 & \mathbf{0}_3 & \mathbf{0}_3 \\ \mathbf{0}_3 & \sigma_a^2\mathbf{I}_3 & \mathbf{0}_3 \\ \mathbf{0}_3 & \mathbf{0}_3 & \sigma_g^2\mathbf{I}_3 \end{bmatrix}$$

Now, the one thing I don't see discussed much in the literature is how we can inject our knowledge of a system into Kalman filters by way of measurements. In this particular case the magnitude of the gravity vector on the surface of the Earth has a known value (more or less). So what we'll do, is we'll augment our measurement vector above, to also include the magnitude of gravity, of approximately 9.80665 m/s/s, meaning our measurement vector is:

[2] Higgins, Walter. "A Comparison of Complementary and Kalman Filtering." IEEE Transactions on Aerospace and Electronic Systems, Vol. AES-11, No. 3, May 1975.

$$\mathbf{z} = \begin{bmatrix} v_z \\ g_0 \\ \mathbf{a}_z \end{bmatrix}$$

And then our measurement prediction function is similar to the previous one, but now taking into account the magnitude of gravity:

$$h(\mathbf{x}_p) = \begin{bmatrix} |\mathbf{v}_p| \\ |\mathbf{g}_p| \\ \mathbf{a}_p \end{bmatrix}$$

Meaning that our measurement observation matrix is similarly:

$$\mathbf{H} = \begin{bmatrix} \frac{\mathbf{v}_p^{\mathrm{T}}}{|\mathbf{v}_p|} & \mathbf{0}_{1\mathrm{x}3} & \mathbf{0}_{1\mathrm{x}3} \\ \mathbf{0}_{1\mathrm{x}3} & \mathbf{0}_{1\mathrm{x}3} & \frac{\mathbf{g}_p^{\mathrm{T}}}{|\mathbf{g}_p|} \\ \mathbf{0}_3 & \mathbf{I}_3 & \mathbf{0}_3 \end{bmatrix}$$

And we also need to add uncertainty into our measurement of gravity, so the measurement uncertainty is:

$$\mathbf{R} = \begin{bmatrix} \sigma_{vz}^2 & 0 & \mathbf{0}_{1\mathrm{x}3} \\ 0 & \sigma_{g_0}^2 & \mathbf{0}_{1\mathrm{x}3} \\ \mathbf{0}_{3\mathrm{x}1} & \mathbf{0}_{3\mathrm{x}1} & \sigma_{az}^2 \mathrm{I}_3 \end{bmatrix}$$

Now we just have to figure out how we want to handle the uncertainties in process and measurement. I find from personal experience when treating Kalman filters as complementary filters, that it's easier to tune the process model uncertainty than it is to tune the measurement uncertainties. So, when it comes to comparing measurements, we'll make sure they're all treated the same:

$$\sigma_{vz}^2 = 1$$
$$\sigma_{g_0}^2 = 1$$
$$\sigma_{az}^2 = 1$$

In order to clean up the acceleration estimate to be less noisy than the measurements, we want our Kalman filter to act as a low pass filter for acceleration data. This means that we want our constant acceleration model to be relatively more certain than the measurements. After all, if your measurements are jumping around at all frequencies (the power spectral density of additive white Gaussian noise is constant across frequencies), you can quiet them down by increasing the certainty that they don't change at all. After playing around with the parameters a bit for this specific scenario, we get good results with a constant acceleration model uncertainty of:

$$\sigma_a^2 = 0.05$$

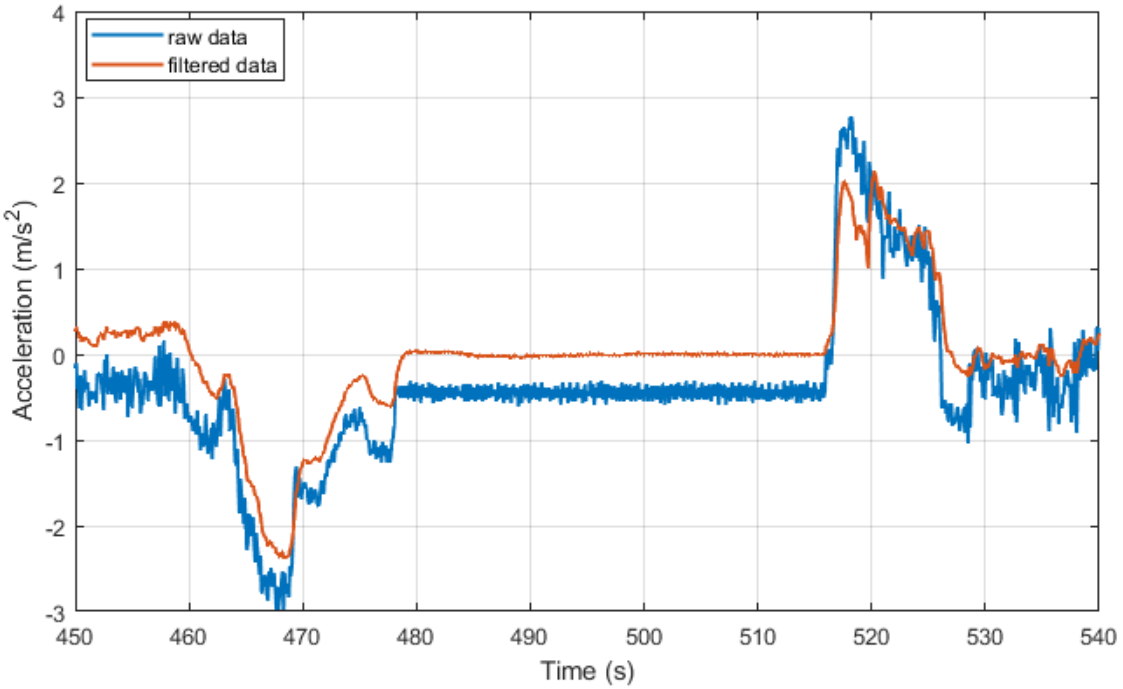
To finally estimate gravity, we have to encapsulate how we want a gravity vector to behave—again, in this specific scenario—into our filter by way of the process model uncertainty. We know we want it to be mostly constant, meaning the constant gravity model needs to be very certain relative to the measurements and other processes. We also know that we want it to become less certain as speed decreases. That is, at high speeds we want gravity to be more constant, i.e. change more slowly, than at low speeds. To do this, we can design a simple process that looks like:

$$\sigma_g^2 = \alpha \left(1 + \frac{\beta}{1 + |\mathbf{v}_p|^2} \right)$$

Where we use the square of the predicted speed to make the constant gravity model adjust relatively quickly. To make sure that gravity is quite constant, we set alpha to be very small, but want to limit beta to be about the same size as the maximum speed that can happen, so we choose:

$$\alpha = 10^{-6}$$
$$\beta = 2500$$

And truthfully, the way to find these parameters is to fiddle around with them until you get the best match that you want. I've got a few tricks for validating parameters like this, but that's outside the scope of this discussion. Anyway, putting this all together, we get the following results:



Which is also pretty good! There's still significant error as the car starts to accelerate around the 515 second mark, but our new model significantly reduces it. Because we named the last filter, let's call this one a simple gravity EKF. Code for this filter can be found in the `ekf_vga.m` file located somewhere near where this document is stored, but also in Appendix B. Code to load in test data, run the filters, and generate the above figures can be found in the `test_gravity_EKFs.m` file, or in Appendix C, with all test data in the `EKFGravityTestData.mat` file.

Now, is this added complexity worth the trouble? That really depends on your use case, and the estimates that matter most to you. For me, all I really cared about was linear and centripetal acceleration that I could validate using speed measurements, so I ended up using a simple clamping filter. But the gravity EKF gives you more parameters to fiddle around with.

Appendix A: Clamping Extended Kalman Filter

```
function [x_est, P_est] = ekf_va(z, dt, x_est, P_est)

F = state_transition(x_est, dt);

Qa = model_uncertainty(x_est, dt);
Q = F * Qa * F.';
R = measurement_uncertainty(z, dt);

x_prd = state_prediction(x_est, dt);
P_prd = F * P_est * F.' + Q;

y = z - measurement_prediction(x_prd);

H = measurement_observation(x_prd);
S = H * P_prd * H.' + R;
K = P_prd * H.' * (S \ speye(size(S)));

x_est = x_prd + K * y;
Fk = speye(size(P_prd)) - K * H;
P_est = Fk* P_prd * Fk.' + K * R * K.';

    % predicting the current state:
    function fx = state_prediction(x_est, dt)

        a = x_est(4:6);
        v = x_est(1:3) + a * dt;

        fx = [v; a];
    end

    % predicting the uncertainty of the current state:
    function F = state_transition(x_est, dt)
        F = zeros(length(x_est));

        F(1:3, 1:3) = eye(3);
        F(1:3, 4:6) = eye(3) * dt;

        F(4:6, 4:6) = eye(3);
    end

    % turning predictions into measurements:
    function hx = measurement_prediction(x_prd)
        v = x_prd(1:3);
        v0 = sqrt(sum(v.^2));
        a = x_prd(4:6);

        hx = [v0; a];
    end

    % Determine covariant uncertainty of predictions and measurements:
    function H = measurement_observation(x_prd)

        H = zeros(4, 6);
        v = x_prd(1:3);
        v0 = sqrt(sum(v.^2));

        if v0 ~= 0
            H(1, 1:3) = [v(1)/v0, v(2)/v0, v(3)/v0];
        else
            H(1, 1:3) = [1, 1, 1];
        end

        H(2:4, 4:6) = eye(3);
    end
```

```
    % Determine the uncertainty of the state prediction process:
    function Qa = model_uncertainty(x_est, ~)
        Qa = eye(length(x_est));
    end

    % Determine the uncertainty of the measurements:
    function R = measurement_uncertainty(z, ~)
        R = eye(length(z));

        v0 = z(1);
        if v0 <= 1
            R(2:4, 2:4) = eye(3) * 10000;
        end
    end
end
```

Appendix B: Gravity Extended Kalman Filter

```
function [x_est, P_est] = ekf_vga(z, dt, x_est, P_est)

F = state_transition(x_est, dt);

x_prd = state_prediction(x_est, dt);
Qa = model_uncertainty(x_prd, dt);
Q = F * Qa * F.';
R = measurement_uncertainty(z, dt);
P_prd = F * P_est * F.' + Q;

y = z - measurement_prediction(x_prd);

H = measurement_observation(x_prd);
S = H * P_prd * H.' + R;
K = P_prd * H.' * (S \ speye(size(S)));

x_est = x_prd + K * y;
Fk = speye(size(P_prd)) - K * H;
P_est = Fk* P_prd * Fk.' + K * R * K.';

    % predicting the current state:
    function fx = state_prediction(x_est, dt)

        a = x_est(4:6);
        g = x_est(7:9);
        v = x_est(1:3) + (a - g) * dt;

        fx = [v; a; g];

    end

    % predicting the uncertainty of the current state:
    function F = state_transition(x_est, dt)
        F = zeros(length(x_est));

        F(1:3,1:3) = eye(3);
        F(1:3,4:6) = eye(3) * dt;
        F(1:3,7:9) = -eye(3) * dt;

        F(4:6,4:6) = eye(3);
        F(7:9,7:9) = eye(3);
    end

    % turning predictions into measurements:
    function hx = measurement_prediction(x_prd)
        v = x_prd(1:3);
        v0 = sqrt(sum(v.^2));

        g = x_prd(7:9);
        g0 = sqrt(sum(g.^2));

        a = x_prd(4:6);

        hx = [v0; g0; a];
    end

    % Determine covariant uncertainty of predictions and measurements:
    function H = measurement_observation(x_prd)

        H = zeros(5, length(x_prd));

        v = x_prd(1:3);
        v0 = sqrt(sum(v.^2));
        if v0 ~= 0
            H(1,1:3) = [v(1)/v0, v(2)/v0, v(3)/v0];
```

```
        else
            H(1,1:3) = [1, 1, 1];
        end

        g = x_prd(7:9);
        g0 = sqrt(sum(g.^2));
        if g0 ~= 0
            H(2,7:9) = [g(1), g(2), g(3)] / g0;
        else
            H(2,7:9) = [1, 1, 1];
        end

        H(3:5,4:6) = eye(3);
    end

    % Determine the uncertainty of the state prediction process:
    function Qa = model_uncertainty(x_prd, ~)
        Qa = zeros(length(x_prd));
        Qa(4:6,4:6) = eye(3) * 0.05;

        v = sqrt(sum(x_prd(1:3).^2));
        Qa(7:9,7:9) = eye(3) * 1e-6 * (1 + 2500/(1+v^2));
    end

    % Determine the uncertainty of the measurements:
    function R = measurement_uncertainty(z, ~)
        R = eye(length(z));
    end
end
```

Appendix C: Testing the Gravity Filters

```
clear;

% contains ax, ay, az, acctime, gpsspeed, gpstime:
load('EKFGravityTestData.mat');

clampingfilter = false;
gravityfilter = ~clampingfilter;

%#ok<*UNRCH>
if clampingfilter
    nstate = 6;
elseif gravityfilter
    nstate = 9;
end
x0 = zeros(nstate, 1);
P0 = eye(nstate) * 10;
x_est = zeros(length(x0), length(acctime));

gpsidx = 1;
accidx = 1;
accelerometer_event = 1;
gps_event = 2;

v = 0;
a = [ax(1); ay(1); az(1)];
dt = acctime(2) - acctime(1);
fs = 1 / mean(diff(acctime));

m = 3;
[alpha, beta] = getbutter(0.01, 1 / fs, m);
alpha = alpha.';
beta = beta.';
a_raw = zeros(3, m+1);
a_filtered = zeros(3, m);
g = zeros(3, 1);

while true
    [~, sensor_event] = ...
        min([acctime(accidx), gpstime(gpsidx)]);

    switch sensor_event
        case accelerometer_event
            a_now = [ax(accidx); ay(accidx); az(accidx)];

            if clampingfilter
                a_raw = [a_now, a_raw(:,1:end-1)];
                g = sum(a_raw .* beta, 2) - sum(a_filtered .* alpha(2:end), 2);
                a_filtered = [g, a_filtered(:,1:end-1)];
                a_now = a_now - g;

                z = [v; a_now];
                [x0, P0] = ekf_va(z, dt, x0, P0);
                x_est(:, accidx) = x0;

            elseif gravityfilter
                z = [v; 9.80665; a_now];
                [x0, P0] = ekf_vga(z, dt, x0, P0);
                x_est(:, accidx) = x0;
            end

            accidx = accidx + 1;
            dt = acctime(accidx) - acctime(accidx - 1);

        case gps_event
            v = gpsspeed(gpsidx);
```

```
            gpsidx = gpsidx + 1;

    end

    if gpsidx == length(gpstime) || accidx == length(acctime)
        break;
    end
end

results_fig = figure;
if clampingfilter
    plot(acctime, ay, acctime, x_est(5,:), 'LineWidth', 1.5);

elseif gravityfilter
    plot(acctime, ay, acctime, x_est(5,:) - x_est(8,:), 'LineWidth', 1.5);
end
axis([450 540 -3 4]); grid on;
xlabel('Time (s)'); ylabel('Acceleration (m/s^2)');
legend('raw data', 'filtered');
results_fig.Position = [200 500 750 420];
```