Yeah, But Which Way Is Down?

Practical digital notch filters for eliminating gravity from accelerometer readings

Abstract

Drastically slowing things down, you can still get half-decent results when designing simple digital filters with limited requirements if your goals aren't that ambitious. Like, trying to figure out where the middle of the Earth is if you're standing still on top of it. We analyze a few of the common approaches for this task found around the internet, and then finding them insufficient, figure out a few things that need to be taken into account to actually get pretty good results. For first order filters, a simple algorithm and MATLAB implementation is provided at the end.

Not to be a pedantic jerk or anything, but all these hobbyist drone pilots out there on the internet really need to stop reading Wikipedia out of context. What I mean is, if you poke around online forums to see how people are trying to use accelerometers with drones or robots, one question always comes up: unless my drone or robot is in freefall, how do I remove the gravity vector that's always captured by my accelerometer? Bad answers abound, and while I love Wikipedia: we can do better!

The naïve answer is that since gravity is a constant, you can just accumulate the derivative of your input signal over time, and the gravity constant gets differentiated down to zero. And that's certainly true if you can assume a continuous time signal coming from your accelerometer, or at least that you're getting readings sufficiently fast. To be fair, if you first start reading into this stuff, the applications are at least voice or music related, which start at sample rates of a few kilohertz (8 kHz and 44 kHz respectively, usually); and, bad filters used with incredibly high sample rates still get decent results. But that's not true for sensor data where you're getting accelerometer readings at like, 10 Hz, or maybe 100 Hz, but usually less than 10 Hz, and definitely not at regular intervals. This has huge implications for how you'll want to remove gravity.

The first implication is, if you go to Wikipedia and look up high-pass filters to filter out low frequency signals (gravity being a constant, or 0 Hz signal), they'll give you a circuit, and then derive a digital algorithm from the circuit using Laplace transforms. That's a problem because digital sensor readings are most certainly not continuous time analog signals. So, sure, it's a good way to get the point across if your audience is familiar with circuit theory, linear time-invariant systems, and complex analysis... but not so great if you just want to get a decent filter from a single line of code.

The second implication is—while still complaining about Wikipedia—they take these fancy circuits, translate them back into time signals and impulse responses, but then estimate derivates with backward finite differences to get you a causal response. However, there's no mention of stability or accuracy or other considerations on the efficacy of the filter. The statements they make about cutoff frequencies and time constants and all of that stuff applies quite literally to designing filters using circuit components, but when you convert these into their digital equivalent in the same way that they have, you get really bad filters. They even give you terrible advice in the Android documentation for using accelerometer sensor values[1].

Finally, the third one (you always need three things when listing things) is that it feels weird using, "standard," filters for such low frequency applications. Like, if you're designing a simple low-pass filter using windows or frequency sampling methods, the minimum number of previous samples you're going to need usually starts at 15. If you're sampling at 1 Hz... is the information from 15 seconds ago really that useful? If you're flying a drone or driving a robot, how much has changed in that timeframe, and how much have you been able to correct for?

[1] https://developer.android.com/reference/android/hardware/SensorEvent#sensor.type_accelerometer

So, let's see if we can do any better. For the rest of this, I'm going to assume you somehow have intimate familiarity with the difference between periodic continuous time signals (and the Continuous Time Fourier Series), periodic discrete time signals (and the Discrete Time Fourier Series), aperiodic continuous time signals (and the Fourier Transform and Laplace Transform), and aperiodic discrete time signals (and the Discrete Fourier Transform and Z-Transform).

To start off, let's investigate the idea of taking the derivative of accelerometer data to remove the constant force of gravity (on the face of the Earth). So, let's say we have a function or signal, ***y***, and we know it up to some time, t_0. To then find ***y*** at some future value, we can just integrate the derivative of ***y***, such that:

$$\mathbf{y}(t_1) = \mathbf{y}(t_0) + \int_{t_0}^{t_1} \mathbf{y}' \, \mathrm{d}t$$

Which seems pretty straightforward. Now, if ***y*** is the same as some other function, ***x***, except they differ by some constant, ***g***, like:

$$\mathbf{x} = \mathbf{y} + \mathbf{g}$$

It's again nice and straightforward that if we take the derivative of ***x***, the constant ***g***, can be eliminated:

$$\mathbf{x}' = \mathbf{y}' + \mathbf{g}' = \mathbf{y}'$$

And we can replace the derivative of ***y*** with the derivative of ***x*** in our previous integration step:

$$\mathbf{y}(t_1) = \mathbf{y}(t_0) + \int_{t_0}^{t_1} \mathbf{x}' \, \mathrm{d}t$$

Nothing fancy here. The reason we did this though, is because what if now, ***x***, is some input to a system, say a vector containing accelerometer data, but contains that pesky constant, ***g***, representing a constant gravity, really screwing up our system. Obviously, our vehicle or sensor or whatever isn't constantly accelerating by at least 9.81 m/s/s. But now that we've taken the derivative of that input signal, ***x***, theoretically there should be no more gravity. Now, following through with the integration, we can get:

$$\mathbf{y}(t_1) = \mathbf{y}(t_0) + \mathbf{x}(t_1) - \mathbf{x}(t_0)$$

Where t_1, is the current time step, where we've just received new data in ***x***, and the moment where we'd like to remove gravity from. Time t_0 is the previous time step, where we still have access to previous readings in ***x***, and we also have access to previously calculated values in ***y***. Putting things now into a discrete time notation, we have:

$$\mathbf{y}[n] = \mathbf{y}[n-1] + \mathbf{x}[n] - \mathbf{x}[n-1]$$

And then re-arranging so that all the signals are grouped together:

$$\mathbf{y}[n] - \mathbf{y}[n-1] = \mathbf{x}[n] - \mathbf{x}[n-1]$$

Since this is a digital signal, with an arbitrary input probably with some additive noise on top of it, we can analyze everything by taking the Z-Transform[2,3] of both sides:

$$(1 - z^{-1})Y(z) = (1 - z^{-1})X(z)$$

And then dividing one side by the other, we can find the impulse response or transfer function of this system such that:

$$H(z) = \frac{Y(z)}{X(z)} = \frac{(1 - z^{-1})}{(1 - z^{-1})}$$

Which is... almost truly meaningless. What does this even mean for a transfer function? I mean, if you follow through with the division, you get something like this when you then bring this, “filter,” back into the time domain:

$$\frac{(1 - z^{-1})}{(1 - z^{-1})} = 1 \Longleftrightarrow \delta[n]$$

What does that even mean? Does it mean that we’ve just created an all-pass filter? Well, we know that’s not quite true, because if we have a gravity signal, we can watch it disappear from our data. But is that always true? Let’s take a look some more. Perhaps we can learn some more from this simple filter if we expand it out to something like:

$$\frac{(1 - z^{-1})}{(1 - z^{-1})} = \frac{1}{1 - z^{-1}} - \frac{z^{-1}}{1 - z^{-1}}$$

Well, we haven’t done anything to specify a region of convergence (ROC) for this filter, so there’s still some ambiguity with how we can think about it. For example, if this filter has an ROC where the magnitude of *z* can extend out to infinity such that it’s causal, then the time domain function is:

$$\frac{1}{1 - z^{-1}} - \frac{z^{-1}}{1 - z^{-1}} \Longleftrightarrow u[n] - u[n-1]$$

But, if it has an ROC where the magnitude of *z* is the inverse such that it’s anti-causal, then the time domain function is:

$$\frac{1}{1 - z^{-1}} - \frac{z^{-1}}{1 - z^{-1}} \Longleftrightarrow u[-n-1] - u[-n]$$

But both of these functions are essentially the same, and really, are just the unit impulse as we recently saw. In order to continue the conversation, let’s assume for a minute that this actually does take the form of a high-pass filter by adding a small error term, w_0, to the denominator. That way, as w_0 goes to zero, we recover the, “filter,” created by our derivatives, and we get:

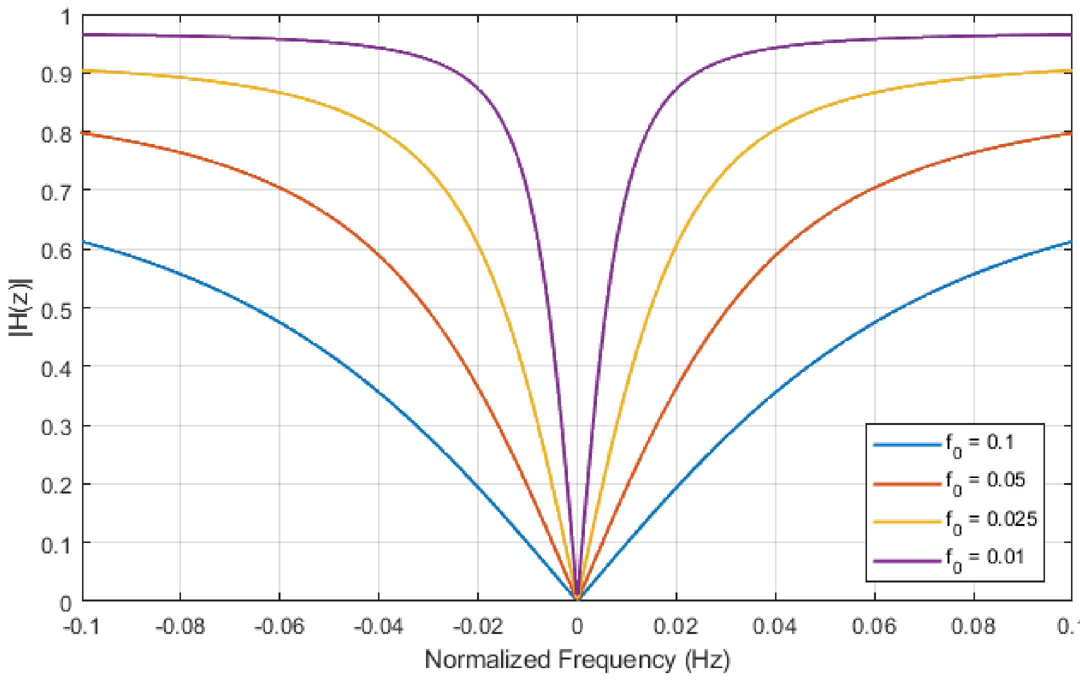
[2] Proakis & Manolakis, “*Digital Signal Processing,”* Chapter 3: The z-Transform and Its Applications... pp. 147. Pearson Education, New Jersey, USA, 2007.
[3] https://en.wikipedia.org/wiki/Z-transform

$$\frac{1 - z^{-1}}{1 - z^{-1}} = \lim_{\omega_0 \to 0} \frac{1 - z^{-1}}{1 + \omega_0 - z^{-1}}$$

Looking at this, we can definitely tell that this is a high-pass filter. There's an obvious zero on the real line when the magnitude of *z* is one, meaning when it has an angle of zero; and, a pole on the real line when the magnitude of z is 1 / (1 + w_0). It looks as though that w_0 even acts as some sort of cut-off frequency, reducing the gain of our filter below a certain point so that it more quickly gets to zero. To illustrate this better, we need to find the frequency response of this filter, which we do by finding the magnitude of the transfer function when *z* is on the unit circle. That is, the frequency response is given by:

$$\begin{aligned} |H(z \to e^{j\omega})| &= \left| \frac{1 - e^{-j\omega}}{1 + \omega_0 - e^{-j\omega}} \right| \\ &= \left(\frac{2(1 - \cos(\omega))}{1 + (1 + \omega_0)^2 - 2(1 + \omega_0)\cos(\omega)} \right)^{\frac{1}{2}} \end{aligned}$$

If we then plot this over the range +/- π, we can take a look at the frequency response for a couple of different cases where $w_0 = 2\pi f_0$ to get an intuitive sense of how this behaves as we approach the limit where w_0 is zero:



As a reminder, we have to use normalized frequency here as digital signal are limited by their sampling rate. Since we haven't directly specified a sampling rate, the normalized frequency sits in the range of +/- 0.5 of the sampling rate. That is, if we were to sample at 10 Hz, the Nyquist frequency—the highest frequency that we can filter before aliasing or, "folding,"—would be 5 Hz. Signals above 5 Hz would wrap around in this frequency domain to be modified by the filter's response on the other end of the spectrum. For example, a 6 Hz signal would experience the same gain and phase change as a –4 Hz signal; or, assuming a symmetric spectrum: a 4 Hz signal.

Anyway, the reason I go on about how this is a terrible filter is because adjusting our one free parameter, w_0, does not give us predictable results at first glance. Normally with this notation, we'd expect it to be a cut-off frequency whereby at that frequency signals are reduced by a 50%, or 3-dB. The 3-dB point for each of these filters doesn't seem to really correspond to w_0 at all. Also, you would expect for a high-pass filter that after the cut-off frequency, the gain would quickly get back up to one so that all of the higher frequency signals would... actually get passed. As is obvious from the above diagram, except for very small values of w_0, higher frequencies are always attenuated quite significantly. There is only a unity gain when the cut-off frequency is set to zero. While yes, there is a zero for DC signals (i.e. where gravity would be), the gain of these filters is not terribly predictable.

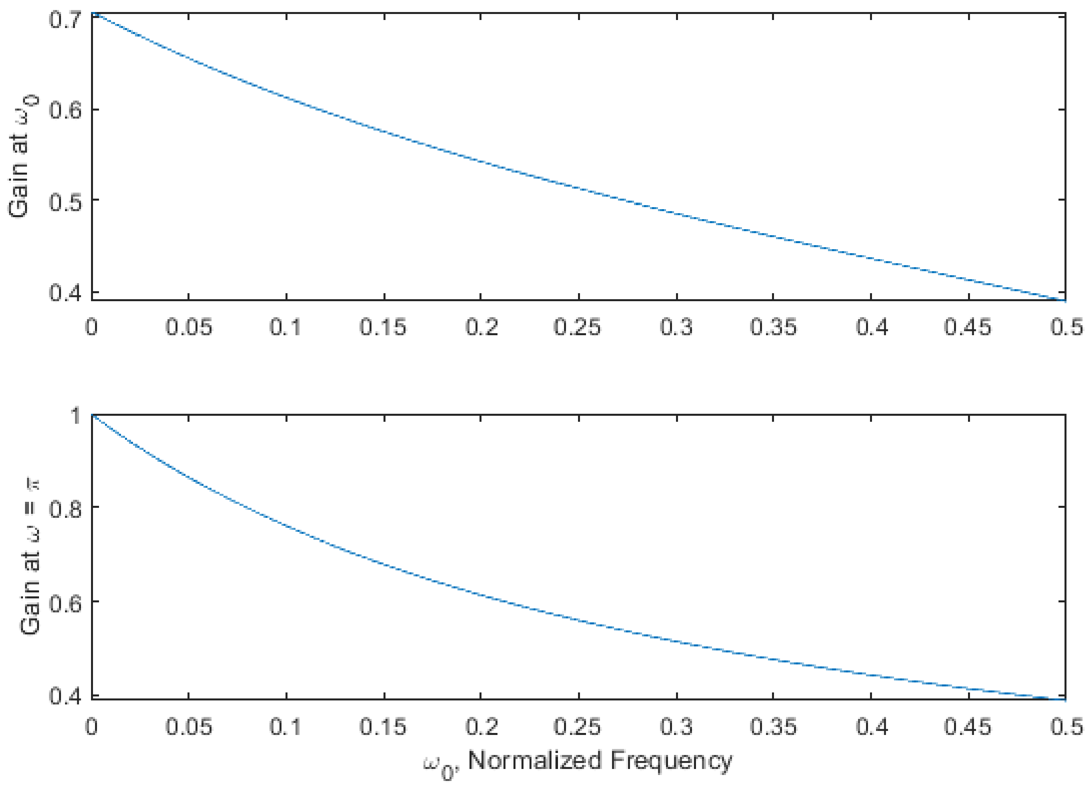
In fact, it gets even more interesting. Going back to our straight up, "derivative filter," by taking this in the limit that w_0 goes to zero, we can see that the gain when w_0 equals zero is... not zero:

$$\begin{aligned}\lim_{\omega_0 \to 0} |H(\mathrm{e}^{j\omega_0})| &= \left(\frac{2(1-\cos(\omega_0))}{1+(1+\omega_0)^2 - 2(1+\omega_0)\cos(\omega_0)} \right)^{\frac{1}{2}} \\ &= \frac{1}{\sqrt{2}}\end{aligned}$$

Which is actually what we would expect for a standard high-pass filter using an analog circuit, whereby the *power* (the square of the signal) is reduced by 50% at the cut-off frequency. But of course, this only approximates such a continuous-time filter in the limit where the cut-off frequency becomes zero, and our filter gain is technically undefined (even MATLAB gives NaN). I suppose that's to be expected when you're dividing zero by zero, but still, it's fun to see such contradictions come up in, "real life."

So, we conclude that the only way to filter out gravity using derivatives reasonably well is to have a signal that's pretty much continuous. Maybe that means 100 Hz is good enough, depending on how your drone or robot moves through physical human world space, but is 10 Hz or even 1 Hz good enough? I think we're gonna run into some problems...

To further nail home the point of this being a bad approach to filter design, take a look at graphs for both the gain at the cut-off frequency, w_0, as we change w_0; and also, the gain at w = 0.5 (or π) as we change w_0:



From this we can see that there's only one point that w_0 can be where it corresponds to the 3-dB point in a filter. And, for our purposes it's just much too high. Also, the only time the filter reaches unity gain at any point in the passband (i.e. at frequencies sufficiently beyond the cut-off) is when the cut-off frequency is zero! Both things considered together, it makes it difficult to use this approach for anything resembling some kind of design activity.

This brings us back to Wikipedia. If we go to the Wikipedia page for high-pass filters and scroll all the way down to the algorithmic implementation[4] of a high-pass filter, they give the following difference equation:

$$y[n] = \alpha y[n-1] + \alpha(x[n] - x[n-1])$$

Which is certainly a high-pass filter, but if you re-arrange things, take the Z-Transform, and then find the transfer function you get:

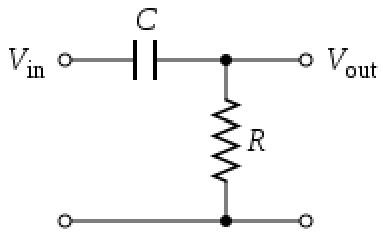
$$H(z) = \frac{1 - z^{-1}}{\frac{1}{\alpha} - z^{-1}}$$

Which is exactly the filter we were just complaining about above if alpha is less than one. Again, the reason it's such a poor filter is that for any value of alpha that's not one, the pass band will never have unity gain, and who knows what the gain at the cut-off frequency will be. To be fair, they got to this result by analyzing an analog circuit, finding the differential equation that represents the output of that circuit, discretizing it, and then building a filter algorithm that you can put into a computer. This is a completely legitimate way of taking

[4] https://en.wikipedia.org/wiki/High-pass_filter#Algorithmic_implementation

a known analog filter template then adjusting it to your needs. They even give it a name: "Infinite Impulse Response (IIR) Filter Design by Approximation of Derivatives."[5] However, you do have to understand what you're getting into when you do these things.

Let's try this another way by analyzing the same simple circuit from the Wikipedia page:



high-pass filter stolen from Wikipedia

In order to do so, we'll have to take everything into the complex, s-domain, by way of the Laplace transform. The input and output voltages we don't need to worry about, as all we need to do is find the transfer function as the ratio between output and input. First, the Laplace transform of a constant, real-valued resistor gives an impedance identical to that of its resistance, R. Second, the impedance of a capacitor with capacitance, C, is given by its Laplace transform of the integral equation it represents such that:

$$Z_c = \frac{1}{sC}$$

Which means that the current through the circuit is just the input voltage divided by the total impedance:

$$I = \frac{V_{in}}{R + \frac{1}{sC}}$$

And then the output voltage is given by the current through the resistor multiplied by the impedance of the resistor:

$$V_{out} = IR = \frac{V_{in}}{R + \frac{1}{sC}} R$$

Meaning that the transfer function of the circuit, output voltage divided by input voltage, is given by:

$$H(s) = \frac{V_{out}}{V_{in}} = \frac{R}{R + \frac{1}{sC}} = \frac{s}{s + \frac{1}{RC}}$$

Which looks like a standard high-pass filter transfer function to me, with a cut-off frequency of 1/RC rad/s that determines the 3-dB point for the *power* of the system. To help us continue the discussion, let's switch back to the variables that we used before, where our input signal is given by *x,* our output signal is given by *y*, and our cut-off frequency is given by w_0. This now means that we have:

[5] Proakis & Manolakis, "*Digital Signal Processing,"* Chapter 10.3.1: IIR Filter Design by Approximation of Derivatives, pp. 703. Pearson Education, New Jersey, USA, 2007.

$$H(s) = \frac{Y(s)}{X(s)} = \frac{s}{s+\omega_0}$$

If we cross multiply things out, then we get:

$$(s+\omega_0)Y(s) = sX(s)$$

If we then bring this back into the time domain we get:

$$y'(t) + \omega_0 y(t) = x'(t)$$

Where we've ignored the initial conditions for the sake of convenience—even the text books do it! If we then bring this into the digital domain by discretizing everything (replacing *t* with *n*) and using a first backward difference to estimate the derivatives we get:

$$\frac{1}{\Delta t}(y[n] - y[n-1]) + \omega_0 y[n] = \frac{1}{\Delta t}(x[n] - x[n-1])$$

If we then take the Z-Transform of this mess, and re-arrange to see what it looks like as a digital filter we get a transfer function that is:

$$H(z) = \frac{\frac{1}{\Delta t}(1 - z^{-1})}{\frac{1}{\Delta t}(1 - z^{-1}) + \omega_0} = \frac{1 - z^{-1}}{1 + \omega_0 \Delta t - z^{-1}}$$

Which is... more or less exactly where we started. Remember, our, "derivative filter," is exactly this in the limit where w_0 and Δt are taken to be zero. It didn't come up when we first started, but Δt is the period in between samples (or sampling interval), the inverse of which is our sampling rate. This means that this also determines the actual frequency characteristics of our filter, as opposed to the characteristics (magnitude and phase) in terms of normalized frequency. This also implies that what we've done, and what's been done on Wikipedia, has been to take the analog circuit high-pass filter, and make a substitution between the complex s-domain, and the complex z-domain such that:

$$s \rightarrow \frac{1}{\Delta t}(1 - z^{-1})$$

But as we know from our textbook[5], this maps the imaginary line in the s-domain (upon which the Fourier transform is defined), not to the unit circle in the z-domain (which is what the z-transform is supposed to), but to a circle centered at z = 0.5, with a radius of 0.5. This means that playing around with filters of this type is not only a bit riskier as it's easier to accidentally get anti-causal, unstable filters, it's harder to build design templates that are easy to configure as required. However, if we instead do something wacky, and transform from the s- to z-domain by way of the approximation:

$$s \rightarrow \frac{2}{\Delta t}\frac{1 - z^{-1}}{1 + z^{-1}}$$

… then we can get some interesting results. I don't want to work through the derivation, as it can be found again on Wikipedia[6] or in the textbook[7] but this is the result of using the trapezoidal rule for integration instead of differentiating by way of first backward differences. It is called the bilinear transform. The nice thing about this is it properly transforms the imaginary line in the s-domain to the full unit circle in the z-domain so that we don't have to worry about bounded input, bounded output (BIBO) stability complications with the previous backward differences transform. That is, if we take a prototype analog filter and transform it using the bilinear transform, we know that everything we do is causal and stable if all of our poles in the z-domain are within the unit circle (meaning that all of our poles in the s-domain are in the left half plane). Now, let's see how this approach changes our very simple high-pass filter:

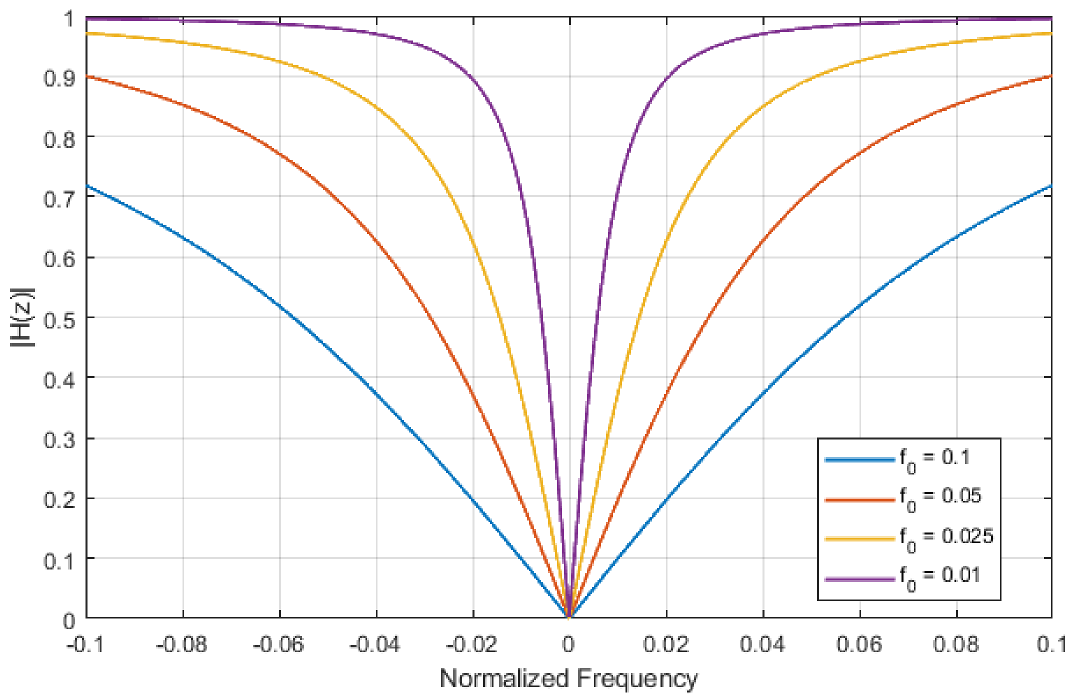
$$\frac{s}{s+\omega_0} \rightarrow \frac{\frac{2}{\Delta t}\frac{1-z^{-1}}{1+z^{-1}}}{\frac{2}{\Delta t}\frac{1-z^{-1}}{1+z^{-1}}+\omega_0} = \frac{1-z^{-1}}{(1+\omega_0\frac{\Delta t}{2})-(1-\omega_0\frac{\Delta t}{2})z^{-1}} = H(z)$$

From here on out, I'm going to go ahead and set *Δt* to be one. Not only does this make things slightly easier to type out in LaTeX, but it also matches up with what I'm dealing with in my particular scenario. However, the best reason is that when $w_0 = 2\pi f_0 \Delta t$, and Δt is one, then f_0 becomes a percentage of whatever your sampling rate actually is. Also, if your sensor readings come in at different times, it's a total pain to keep track of previous time steps and constantly updating your filter parameters. It's easier to just straight up filter a percentage of all frequencies. This still gives good results if your time between samples doesn't vary that much. For me personally, this also makes it easier to discuss and reason about these things; but really only in the context of a high-pass or low-pass filters. That way, if f_0 is for example 0.01, and we design our filter well, I can say that 99% of the frequencies processed by the filter are greater than the cut-off frequency, most of which experience unity gain. This also helps us reason a bit better about aliasing, as we can say that the vast majority of those 99% of frequencies will also experience unity gain—assuming we design our filter well.

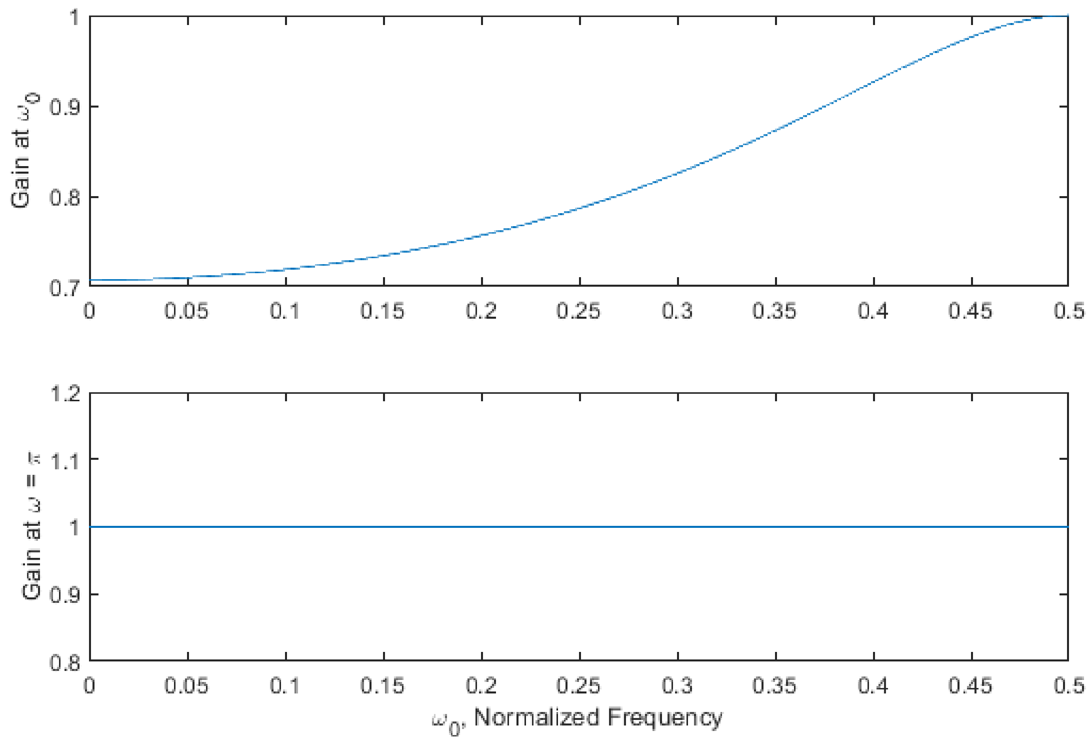
Now, is this a good way to design a filter? Let's take a look at the frequency of the magnitude response, by again evaluating the transfer function on the unit circle:

[6] https://en.wikipedia.org/wiki/Bilinear_transform

[7] Proakis & Manolakis, "*Digital Signal Processing,*" Chapter 10.3.3: IIR Filter Design by the Bilinear Transformation, pp. 703. Pearson Education, New Jersey, USA, 2007.



Which seems pretty good. At least there's unity gain in the passband way before our cut-off frequency, *w_0*, gets to zero. But what's the gain at the cut-off frequency itself? Is it predictable so that as we vary *w_0* we can make reasonable predictions about the characteristics of our filter? In order to do that, let's again plot the gain of this filter at the cut-off frequency as it's varied from zero to 0.5 (normalized frequency). We'll also take a look at the gain of the filter at the extreme end of the supposed passband, when *w* = 0.5 (or π) as we vary the cut-off frequency:



This is indeed promising. At least in the extreme end of the passband when *w* is π, this new filter always reaches unity gain. This is one consequence of the bilinear transform mapping the entire imaginary line in the s-domain to the unit circle in the z-domain, but I digress. Unfortunately, like our previous, "derivative," filter, the gain at the cut-off frequency itself is not constant. But we're still halfway there! What if we do something clever and tweak this just a little bit so that we can realize both of our goals: unity gain in the passband and predictable gain in at the cut-off frequency?

One way we might attempt to do this is by augmenting our equation with a multiplicative factor for w_0 that shifts things ever so slightly but in a predictable way. Let's do that; let's take our new transfer function, but multiply the cut-off frequency by some function, *b*, and see if we can get something useful:

$$H(z) = \frac{1 - z^{-1}}{(1 + \omega_0 \frac{b(\omega_0)}{2}) - (1 - \omega_0 \frac{b(\omega_0)}{2}) z^{-1}}$$

Now, the only thing we want to change is to make sure that at our cut-off frequency, the gain of the filter is 50%; though 50% of the signal strength, not the power of the signal. After all, what's the power of acceleration due to gravity...? To do this, we have to evaluate the magnitude of the frequency response at the cut-off frequency. This requires us of course to evaluate our transfer function on the unit circle, but at the cut-off frequency. We also evaluate the squared magnitude to make the math easier, such that:

$$|H(z \to \mathrm{e}^{j\omega_0})|^2 = \left| \frac{1 - \mathrm{e}^{-j\omega_0}}{(1 + \omega_0 \frac{b(\omega_0)}{2}) - (1 - \omega_0 \frac{b(\omega_0)}{2}) \mathrm{e}^{-j\omega_0}} \right|^2 = \left(\frac{1}{2}\right)^2$$

Now let's first divide this problem up a little bit by focusing first on the squared magnitude of the numerator:

$$\begin{aligned} |1 - \mathrm{e}^{-j\omega_0}|^2 &= (1 - \cos(\omega_0))^2 + \sin^2(\omega_0) \\ &= 2(1 - \cos(\omega_0)) \end{aligned}$$

And then of course, similarly for the denominator:

$$\begin{aligned} &\left| (1 + \omega_0 \frac{b(\omega_0)}{2}) - (1 - \omega_0 \frac{b(\omega_0)}{2}) \mathrm{e}^{-j\omega_0} \right|^2 \\ &= 2\left(1 + \left(\omega_0 \frac{b(\omega_0)}{2}\right)^2\right) - 2\left(1 - \left(\omega_0 \frac{b(\omega_0)}{2}\right)^2\right) \cos(\omega_0) \end{aligned}$$

We can then take these results, divide the numerator by the denominator again setting it equal to the square of one half, then cross multiplying we get:

$$4(1 - \cos(\omega_0)) = (1 - \cos(\omega_0)) + \left(\omega_0 \frac{b(\omega_0)}{2}\right)^2 (1 + \cos(\omega_0))$$

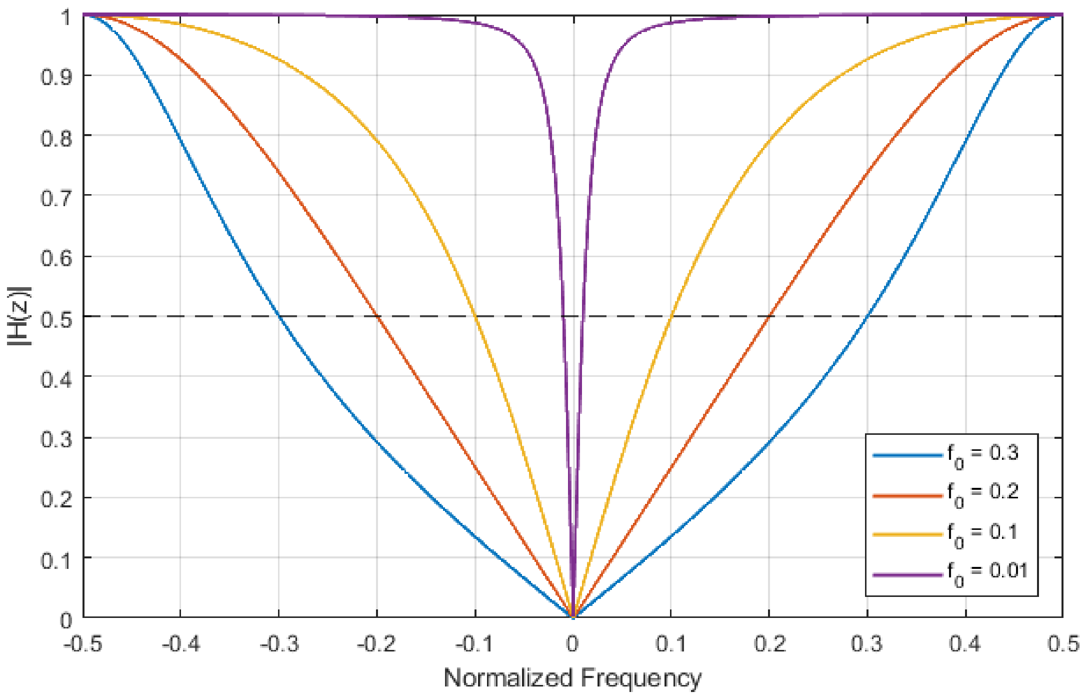
If we then expand out the RHS and re-organize everything, we can get all of the cosines on the RHS of the equation, with *b* on the LHS such that:

$$\left(\omega_0 \frac{b(\omega_0)}{2}\right)^2 = 4\frac{1-\cos(\omega_0)}{1+\cos(\omega_0)}$$

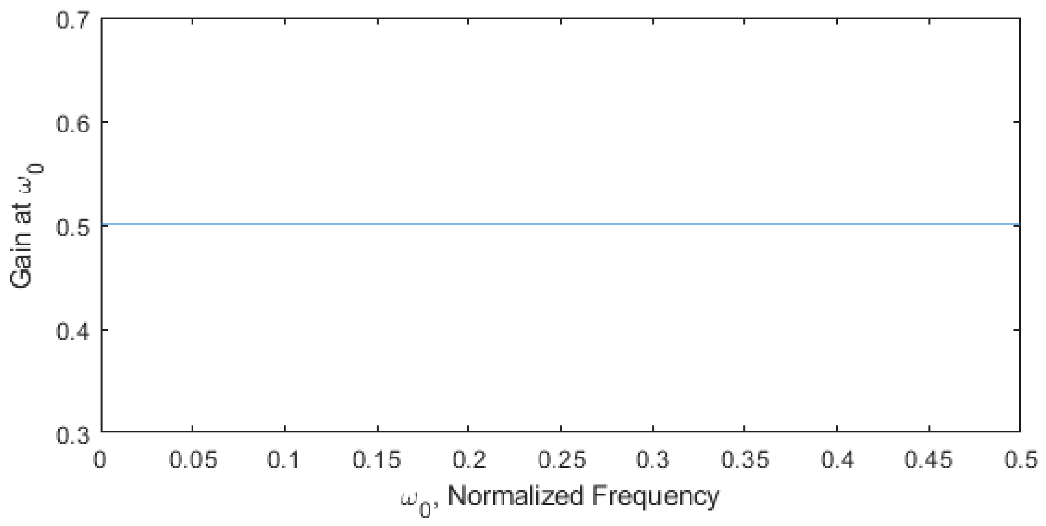
And then the final step is to quickly solve for *b*, so that we finally find:

$$b(\omega_0) = \left(\frac{12}{\omega_0^2}\frac{1-\cos(\omega_0)}{1+\cos(\omega_0)}\right)^{\frac{1}{2}}$$

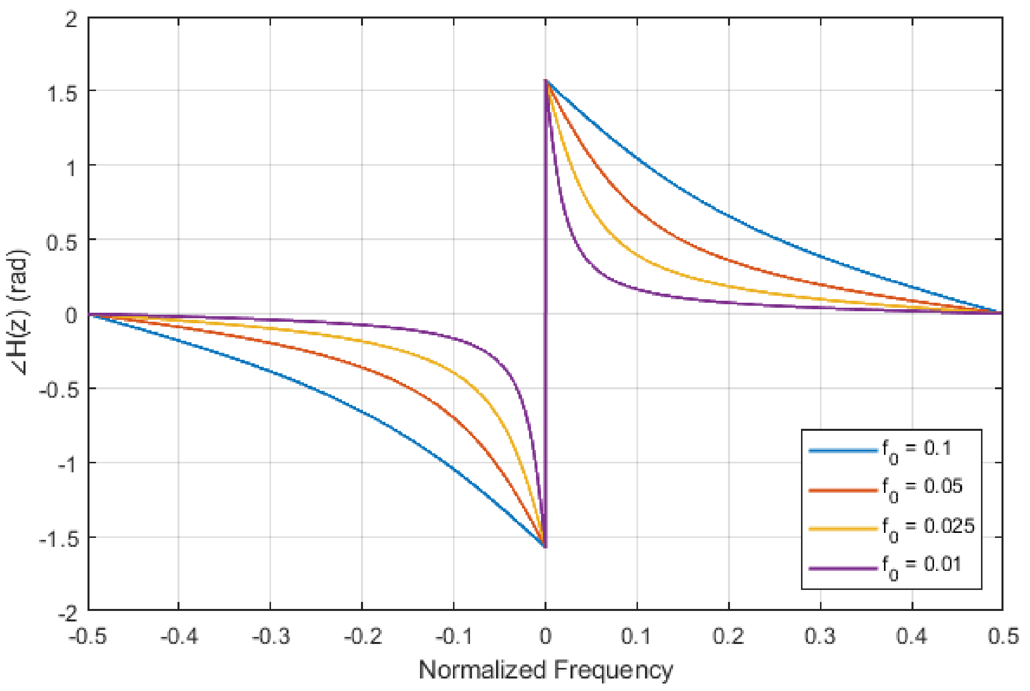
Where we ignore the negative result of the square root as that would move the pole of our filter outside of the unit circle, thus making things unstable or anti-causal. To see if we've gotten anywhere, let's take a look at the magnitude of the frequency response of this filter with a few different cut-off frequencies:



And that looks pretty good! For the first three cut-off frequencies that we picked, it's obvious that they all have a constant gain of 0.5 right at w_0 like we wanted, and everything definitely goes to unity gain as our frequencies increase towards half of our sampling rate. Alright. As it's not clear for lower frequencies; however, let's take a look at the full plot of the gain at the cut-off frequency as we vary the cut-off frequency:



And that looks pretty good too. But, before we call it mission accomplished, we would be doing ourselves a disservice if we didn't next look at the phase response of our new filter. So again, evaluating our transfer function on the unit circle, but instead of the magnitude we look at the angle:



Which is actually really good, *for low cut-off frequencies or high sample rates.* I say this is good, because for the most part in those circumstances, the phase change through the pass band—most of the filter—is literally zero. As far as high-pass filters go, that's really good. Especially so when we're dealing with low sampling rates—if we set our cut-off frequency to be low enough—we don't have to worry about aliasing so much, as most of the frequencies that get folded over will still see unity gain and zero phase shift.

Now, how do we actually *use* these filters? Mostly, we just do the opposite of what we started with: we turn our transfer function into a difference equation. So, what we're trying to do is get things into a form that we can easily put into our computer; something that looks like:

$$y[n] = \alpha y[n-1] + \beta(x[n] - x[n-1])$$

To get here, we have to take our transfer function:

$$H(z) = \frac{1 - z^{-1}}{(1 + \omega_0 \frac{b(\omega_0)}{2}) - (1 - \omega_0 \frac{b(\omega_0)}{2})z^{-1}}$$

We then put this back in terms of an input and output signal:

$$\left((1 + \omega_0 \frac{b(\omega_0)}{2}) - (1 - \omega_0 \frac{b(\omega_0)}{2})z^{-1}\right) Y(z) = (1 - z^{-1})X(z)$$

And then we take the inverse Z-Transform, which gets us:

$$(1 + \omega_0 \frac{b(\omega_0)}{2})y[n] - (1 - \omega_0 \frac{b(\omega_0)}{2})y[n-1] = x[n] - x[n-1]$$

And then re-arranging we get our difference equation above:

$$y[n] = \alpha y[n-1] + \beta(x[n] - x[n-1])$$

But with *α* and *β* given by:

$$\alpha = \frac{1 - \omega_0 \frac{b(\omega_0)}{2}}{1 + \omega_0 \frac{b(\omega_0)}{2}} \quad \beta = \frac{1}{1 + \omega_0 \frac{b(\omega_0)}{2}}$$

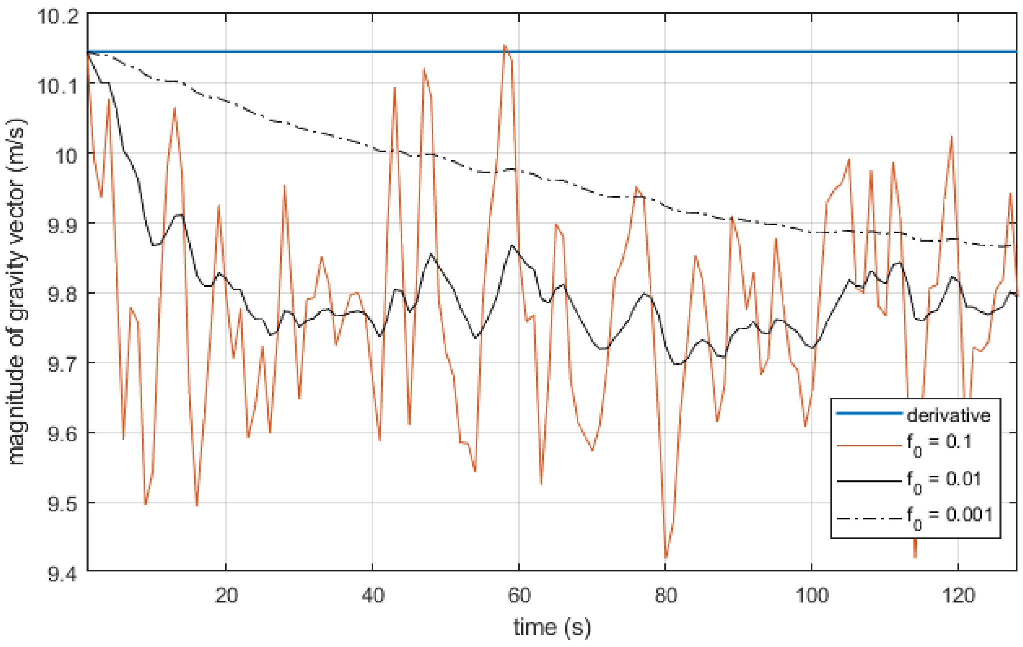
Where we have the function *b* given by:

$$b(\omega_0) = \left(\frac{12}{\omega_0^2} \frac{1 - \cos(\omega_0)}{1 + \cos(\omega_0)}\right)^{\frac{1}{2}}$$

Now, let's see how this filter actually works. In Appendix A, I've put together a MATLAB script that:

- Creates a vector of random acceleration data,
- Adds a constant gravity vector to it,
- Filters this data to remove gravity,
- Subtracts the original data from the filtered data,
  - This is now the best estimate of the gravity vector,
- Finds the magnitude of this new vector,
- Plots the magnitude; and then does this multiple times for four different cut-off frequencies: 0, 0.001, 0.01, and 0.1 (normalized frequency).

Because this script uses random data every time it's run, I ran it a couple of times to find a case that shows what happens when the random data skews the initial guess of gravity right at the start:



Looking at this graph, it's obvious what people mean when they talk about time constants for digital filters. At the risk of anthropomorphizing calculus, just look at the difference between each filter and how long it, "holds on to," past data versus blindly accepting new data. The basic derivative filter has a cut-off frequency of zero, meaning it has a time constant of infinity, meaning that the only data point it, "trusts," is the first one it ever sees. This is a problem if the first data point it ever sees is wrong! As we slowly increase this cut-off frequency, the filter responds faster and faster as it receives new data, until at some point it reacts too quickly, and seems to mostly just track the noise in the base signal. This means that for different filters and different applications there will be a trade-off between filter response time, and accuracy. Now *that's* a proper trade-off, and that's what engineering is all about.

If you want to go really crazy—since all of this uses the same data, and we're only talking first-order filters so it's very straightforward to implement—you can even combine different filters at different times to respond to outside influences. For example, if someone kicks your robot to screw up your accelerometer readings, you can switch to a filter that quickly adjusts back to reality. In order to go *really* crazy, you can even combine this with fancier filters that know how to take into account various types of noise (e.g. Kalman filters, unscented Kalman filters, particle filters), to get yourself a pretty damn good estimate of the true acceleration of your robot or drone.

To conclude, if you want to get a half-decent high-pass filter for removing gravity, you can follow the following algorithm. First, we're looking for numbers to input into a recurrence relation that's easy to then put into a computer. This will take the form:

$$y[n] = \alpha y[n-1] + \beta(x[n] - x[n-1])$$

Where *x* is an input signal representing accelerometer readings, and *y* is that same data but with the constant gravity vector removed. Because accelerometer data across the three main directions is uncorrelated, this works just as well for vector data streams as it does for scalar ones. Then in order to find *α* and *β* we go through the following steps:

1. Pick your cut-off frequency, with your given sample rate, such that your input to the filter would be:

$$\omega_0 = 2\pi \frac{f_0}{f_s} = 2\pi f_0 \Delta t$$

If you want your cut-off frequency to represent a percentage of all frequencies present in the signal, you can just set your sample rate or sampling time to be one.

2. Use this to calculate an adjustment factor, *b,* using:

$$b(\omega_0) = \left(\frac{12}{\omega_0^2} \frac{1-\cos(\omega_0)}{1+\cos(\omega_0)} \right)^{\frac{1}{2}}$$

3. Then find your *α* and *β* with:

$$\alpha = \frac{1 - \omega_0 \frac{b(\omega_0)}{2}}{1 + \omega_0 \frac{b(\omega_0)}{2}} \quad \beta = \frac{1}{1 + \omega_0 \frac{b(\omega_0)}{2}}$$

And... that's it. Simple, right?

Lastly, we can talk a little bit about higher-order filters. Because this is such a simple filter as we're only trying to do our best to remove all DC, or constant, signals, making a higher order filter is as simple as taking the transfer function and raising it to a higher power. However, when you do this, you'll have to re-derive the adjustment factor, *b*, and while it's a pain to do so, it's certainly doable for the first few higher orders. If you don't want to re-derive b, you can leave it as is, but the error in where the 3-dB cut-off is, it seems at first glance, to minimize at about 5%, beyond third order. For a second order filter, you're looking at less than 10% error, which is ok I guess; though for our purposes, accuracy in cut-off frequency—as long as it's reasonably predictable—isn't *that* big a deal.

Why would you want a higher order filter though? Because they tend to be flatter in the stop-band, with a narrower transition band. So, if you want to more effectively remove low frequencies adjacent to your DC signals, that's probably one way to go about doing it. However, please keep in mind that the phase response of higher order filters as they move through the transition band is significantly less flat. This means that if you're sampling slowly and expect a lot of aliasing, you can get less than optimal results, introducing error into your accelerometer data.

Afterward

Thinking about this some more, one thing needs to be stressed: such notch filters will remove *all* DC signals from your data, including other sources of constant acceleration such as... accelerating. They will of course also remove all frequencies that are integer multiples of your sampling rate. For example, if you're sampling at 1 Hz, then 2 Hz, and 3, and 4, and so on... will also be filtered out completely... yet another fun consequence of aliasing.

Also, thinking more about correcting the cut-off frequency, for small cut-offs, we can simplify our, "correction," by realizing that:

$$\frac{12}{\omega_0^2}\frac{1-\cos(\omega_0)}{1+\cos(\omega_0)} \approx \frac{12}{\omega_0^2}\frac{\frac{\omega_0^2}{2}}{2+\frac{\omega_0^2}{2}} = 6\frac{1}{2+\frac{\omega_0^2}{2}} \approx 3$$

Where there is a less than 1% error from the true value when w_0 is less than 0.1, and less than 5% error when w_0 is less than 0.34. Since we're using this to filter out extremely small frequencies, and these values are in normalized frequency, we can for all practical purposes just skip the, "omega correction," step entirely. Also, I can't find the references anymore, but while there is some discussion about frequency warping when converting analog filters to digital ones, it doesn't seem to be that big a deal for practical purposes.

Speaking of which, if we stick to being practical and instead try to design our filters where the "power" of the signal is reduced by half, then just by doing nothing—and for smaller values of w_0—our default approach with no correction is a very good representation of the analog filter. So, my new method of correcting cut-off frequencies for removing constant, or DC, signals from your system is: don't worry about it.

Now Butterworth Filters

Originally, I wasn't going to talk much about Butterworth filters, but then I got bored and was reading up on them on Wikipedia[8] and decided it wasn't that much more complicated than what we've done so far; and, you can still get decent results with lower order filters. This means that they can be quite practical at very slow sampling rates when trying to only remove gravity or other DC signals. Also, I don't have a good reference for this, but apparently if you have an Apple phone, they use proper analog Butterworth filters in order to report, "user acceleration," or, "linear acceleration," to the operating system. I'm told Samsung does something similar with newer phones.

Butterworth filters are the higher order opposites of notch filters in that they're low pass filters whose transfer function in the s-domain only has poles such that, for unity gain:

$$H(s) = \prod_{n=1}^{m} \frac{\omega_0}{s - \omega_0 p_n}$$

Where *m* is the order of the filter, or literally the number of poles that multiply together; w_0 is the cut-off frequency of the filter, and p_n is the phase of the *n*th pole. Now, what gives the Butterworth filter its interesting properties (perfectly flat passband, easy to design to spec) is in the *s*-domain the poles lie evenly spaced in the left-hand plane along a circle centered at $s = 0$, with a radius of w_0, where the pole is located at an angle on the circle given by:

$$p_n = e^{j\frac{\pi}{2m}(2n+m-1)}$$

Where again *m* is the total order of the filter, and *n* is just the number of the current pole you're staring at. So, for example if $m = 1$, then you'll have a single pole at 180°; if $m = 2$ you'll have poles at ±150°; if $m = 3$ you'll have poles at 180° and ±135°; and so on... Now, because we don't want to build any circuits or anything, we have to transform this into the *z*-domain so that we're better able to type it into our computer. As discussed above, the easiest way to do this well is via the bilinear transform:

$$s \Rightarrow \frac{2}{\Delta t}\frac{1 - z^{-1}}{1 + z^{-1}}$$

If we then substitute this into our transfer function above, we get our new one such that:

$$H(z) = \prod_{n=1}^{m} \frac{\omega_0 \Delta t}{2} \frac{1 + z^{-1}}{(1 - \frac{\omega_0 \Delta t}{2} p_n) - (1 + \frac{\omega_0 \Delta t}{2} p_n) z^{-1}}$$

Which is actually pretty interesting if you stare at it long enough.

First off, this filter has zeros! The prototype high-pass filter we used to design a notch filter had both a pole and a zero so it made sense that there was both a pole and a zero, but what gives here? Well, the short answer is that in order to guarantee stability, every pole in a transfer function needs at least one zero to balance it out. In the *s*-domain that's easy to guarantee, as if you have at least one pole, there will always be one zero in the limit where *s* goes to infinity. However, since the entire imaginary axis gets mapped to the unit circle in

[8] https://en.wikipedia.org/wiki/Butterworth_filter

the *z*-domain, all those points in and around infinity get mapped to the unit circle itself, so we have to be explicit about where the zeros are.

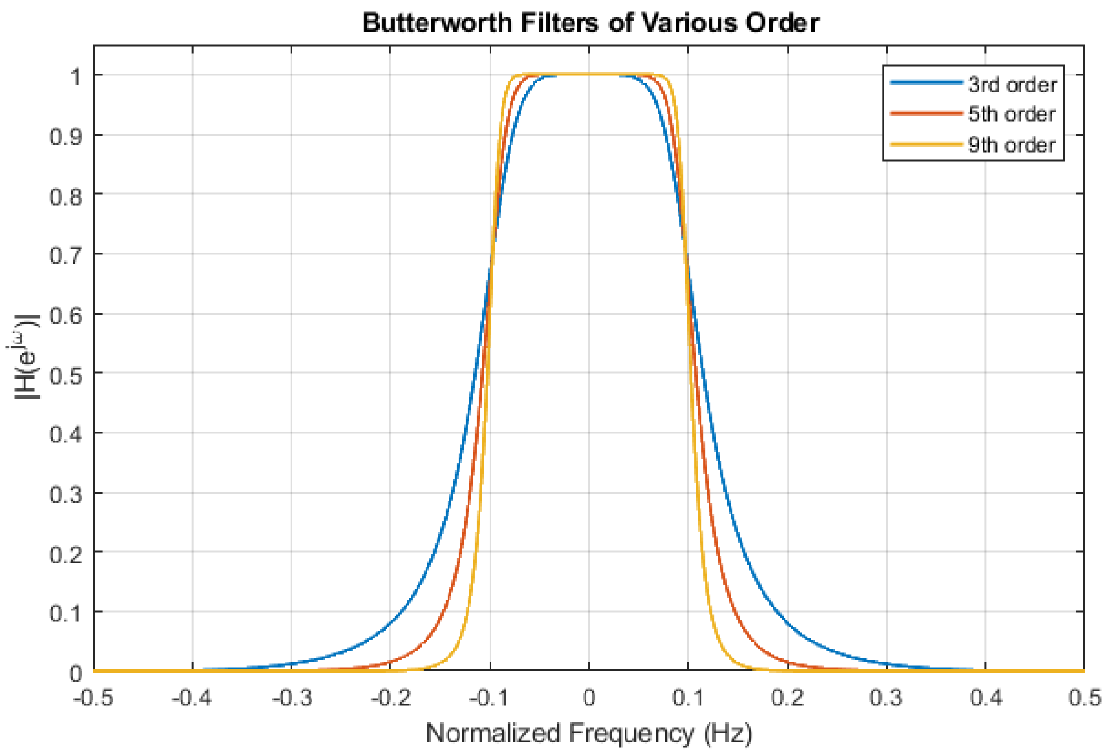
Second, all of the zeros are at the same place, *z* = -1, and all of the poles are inside the unit circle, but no longer at the same angle as they were in the *s*-domain—with the exception of any pole that falls along the real axis. Thankfully, all of the poles that don't lie on the real axis come in complex conjugate pairs so that when we convert everything back into the digital time domain, our filter will always have real coefficients. Stated another way, for every *m* that is even, you'll have *m* / 2 complex conjugate pairs of poles; and, for every *m* that is odd, you'll have one pole that is real and positive, while all other poles come in complex conjugate pairs. Thinking about this some more, if you squint really hard, you can tell that all the poles actually lie in the right-hand side of the unit circle, clustered close to the real axis as the cut-off frequency or sample rate approach zero.

Third, this digital filter is technically more complex than the equivalent circuit. While an *m*th order Butterworth filter will usually have *m* stages, in the digital world we have to deal with *m* zeros and *m* poles explicitly. However, for our purposes we can make a simple approximation: for a small enough cut-off frequency (e.g. for use in DC signal filtering) or sample rate, we can assume that there aren't any zeros (since they all have a real value of negative one), though we would have to adjust the gain of the filter to compensate. We would also have to accept that the phase of the filter would be significantly impacted since there will be poles that aren't strictly accounted for. I don't have a proof for this, but to get something more or less similar, the gain would have to be adjusted such that:

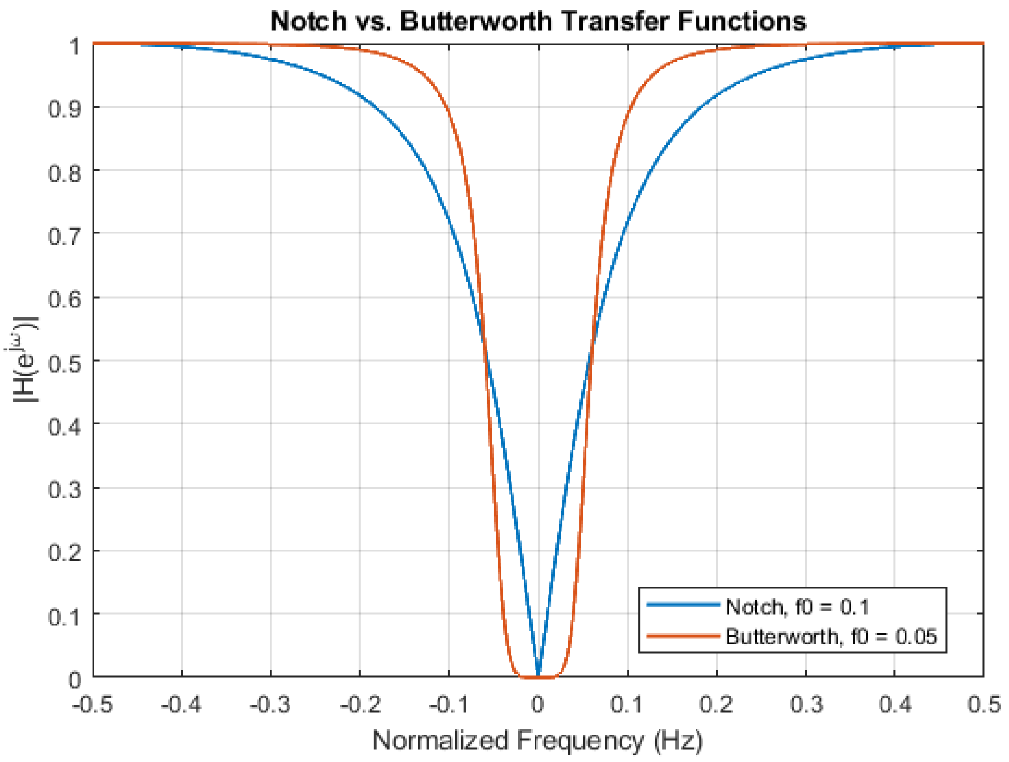
$$H(z) = \left(\sum_{z=1} \left(\frac{\omega_0 \Delta t}{2} (1 + z^{-1}) \right)^m \right) \prod_{n=1}^{m} \frac{1}{(1 - \frac{\omega \Delta t}{2} p_n) - (1 + \frac{\omega \Delta t}{2} p_n) z^{-1}}$$

Which is a bit of a mouthful; however, when you multiply the summand out to convert that into a full polynomial (of order *m*), all it means is you just add up the coefficients of that resulting polynomial, then multiply the poles of your filter by that constant gain. So, while you still have an IIR filter, when you actually use it digitally, you don't have to do as many multiplications and additions as you would if there were still *m* zeros to take into account.

But I digress. Let's take a look at what a Butterworth filter actually looks like for a cut-off frequency (normalized) of 0.1 for a few different orders, *m*:

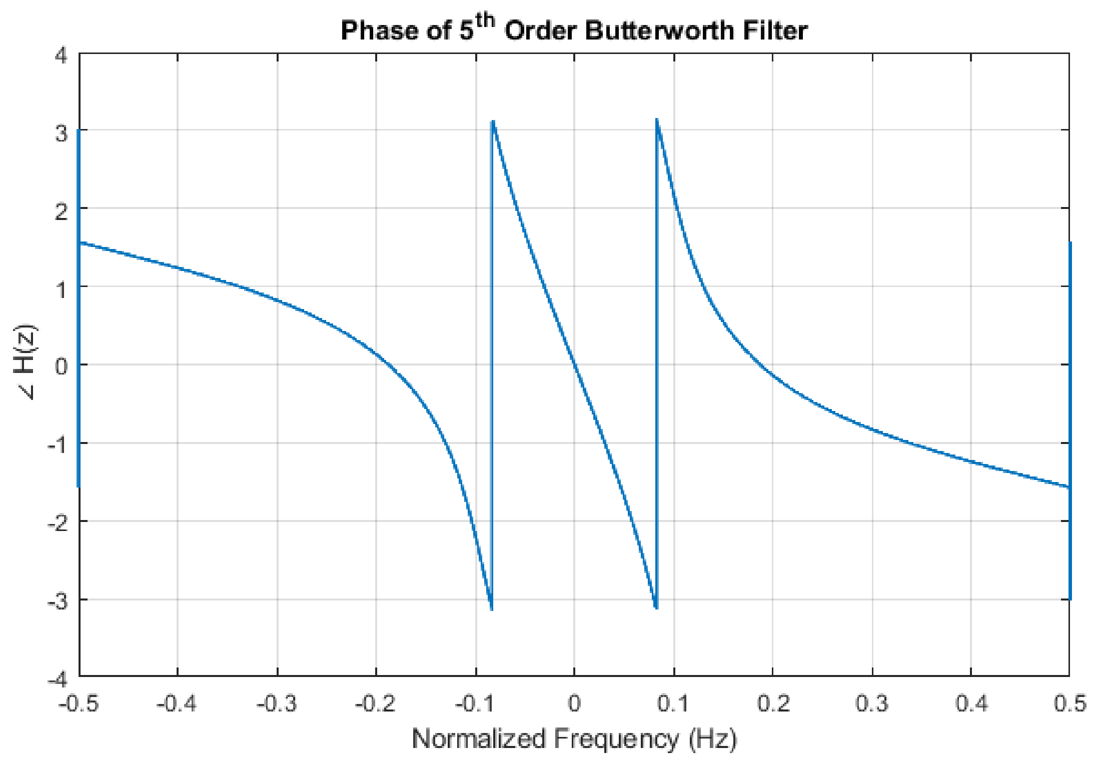


Looking at this it's almost obvious why everyone loves Butterworth filters: the frequency response is almost perfectly flat across the passband; this is a good thing. In the limit as the filter order approaches infinity, it gets as close to a perfect rectangular low pass filter as it gets. However, getting close to ideal does get harder and harder as order increases, meaning that we get to suffer through another round of fun trade-offs. Then again, for our purposes of small cut-off frequency, we can still get good results with lower orders. Of course, because this is a low pass filter, it doesn't filter out a DC signal like gravity, but filters out all other frequencies. In order to use it, we would then take the filtered result, and subtract it from our original measurements to get a signal without gravity. To compare that kind of transfer function with our notch filter above, we can take the Butterworth and flip it upside down:

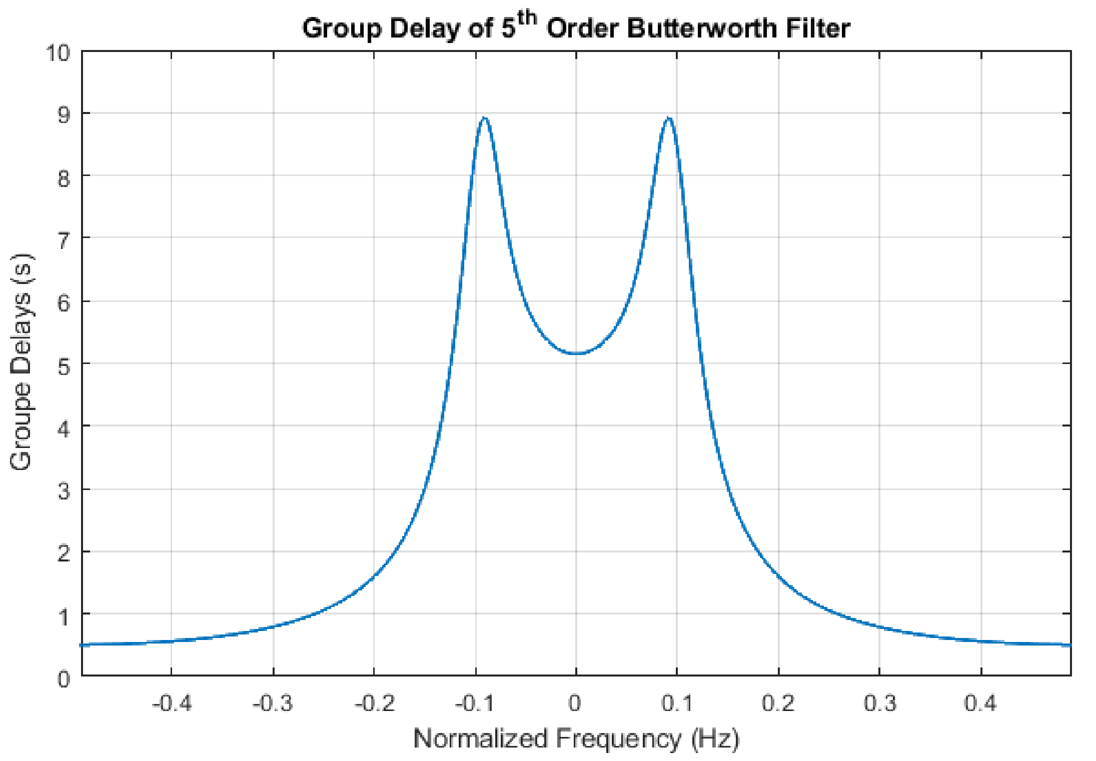


Where the upside-down Butterworth filter has an order of five, cut-off frequency of 0.05, the notch filter has a cut-off frequency of 0.1, and to make it easier to plot, both cut-off frequencies are at the half-power point. It's just an illustrative coincidence that they align at the half-signal point. For a fifth-order Butterworth filter like this, we can see that it has a much thinner transition band, and a wide but flat passband compared to that of the notch filter. So, if you want to get rid of signals close to DC (like a slow-moving gravity vector), and you're willing to accept what accounts to a quintupling of computational complexity (for fifth order), then the Butterworth filter is decent.

Now, finally, we do have to take a quick look at the phase of our Butterworth filter; e.g. for fifth order with a cut-off frequency of 0.1:



Which is pretty wacky compared to the notch filter! Although, truthfully, if you had a higher order notch filter, you'd see something similar. However, because the really wacky part is in the passband, and since this represents the potential gravity signals that we'll be getting rid of anyway, it's not such a big deal. However, if we take a look at the group delay of the same:



There is one thing that we have to think about: in the stop band there is a constant group delay for all signals which may affect the non-gravity measurements that you're interested in. Another trade-off that we have to think about.

Finally, in Appendix C there is some sample code that generates digital filter coefficients that can be used to filter data, such as with the built in MATLAB function `filter`.

Appendix A: Example Code Removing Gravity from Random Acceleration

The following is a script and its supporting functions that demonstrates how to use this high-pass filter template to remove gravity from random acceleration data. The script first creates random acceleration data, adds to it a constant gravity vector, and then filters it. To demonstrate the response of various filters, the two vectors are subtracted, and the magnitude of what remains (the filtered gravity vector) is plotted.

```
clear; close all;

x = rand(3, 128)-1/2 + [0; 0; 9.80665];
y = zeros(size(x));

% derivative filter:
for i = 2:size(x, 2)
    y(:,i) = y(:,i-1) + (x(:,i) - x(:,i-1));
end

g = sqrt(sum((x - y).^2));
figure; plot(g, 'LineWidth', 1.5);
grid on; hold on;

% improved high-pass filter, cut-off at 10%:
[alpha, beta] = getfilter(0.1, 1);
for i = 2:size(x, 2)
    y(:,i) = alpha*y(:,i-1) + beta*(x(:,i) - x(:,i-1));
end

g = sqrt(sum((x - y).^2));
plot(g);

% improved high-pass filter, cut-off at 1%:
[alpha, beta] = getfilter(0.01, 1);
for i = 2:size(x, 2)
    y(:,i) = alpha*y(:,i-1) + beta*(x(:,i) - x(:,i-1));
end

g = sqrt(sum((x - y).^2));
plot(g, 'k');

% improved high-pass filter, cut-off at 0.1%:
[alpha, beta] = getfilter(0.001, 1);
for i = 2:size(x, 2)
    y(:,i) = alpha*y(:,i-1) + beta*(x(:,i) - x(:,i-1));
end

g = sqrt(sum((x - y).^2));
plot(g, 'k-.');

xlabel('time (s)');
ylabel('magnitude of gravity vector (m/s)');
legend('derivative', 'f_0 = 0.1', 'f_0 = 0.01', 'f_0 = 0.001');
```

```
function [alpha, beta] = getfilter(f0, dt)

    if nargin < 2
        dt = 1;
    end

    if f0 * dt > 0.5
        error(['You''re trying to filter beyond the folding frequency ' ...
               'for your given sampling interval.']);
    end

    w0 = 2*pi*f0*dt;
    w0 = w0 * correct_omega(w0);

    alpha = (1 - w0/2) / (1 + w0/2);
    beta = 1 / (1 + w0/2);
end

function b = correct_omega(w)
    b = sqrt(12 * (1 - cos(w))./(1 + cos(w)) ./ w.^2);
end
```

Appendix B: Example Plotting Frequency Response of Z-Transforms

This isn't referenced in the above, but someone asked me to make it more obvious, with code, how I plot the magnitude and phase response of a Z-Transformed transfer function, so here is one example in the MATLAB script below:

```
clear; close all;

% we need to evaluate the Z-Transform around the unit circle, and there are only 2
% pi angles we get to choose from, so we set up our domain as:
w = linspace(-pi, pi, 2^12);

% to evaluate along the unit circle in the complex domain, we need our z to be
% set to complex numbers representing the unit circle.  Or rather, we set our
% inverse, zi, to be the inverse of that:
zi = exp(-1i*w);

% these are just the initial parameters:
f0 = 0.05;
dt = 1;
w0 = 2 * pi * f0 * dt;

% instead of defining a separate function somewhere, we can set up our adjustment
% factor, b, as an inline, or anonymous, function:
b = @(w) sqrt( 12 ./ w.^2 * (1 - cos(w)) ./ (1 + cos(w)));

% this is just the transfer function itself, exactly as written down.  The only
% caveat is that the inverse z, zi, we defined above to be points along the unit
% circle:
H = (1 - zi) ./ ((1 + b(w0)*w0/2) - (1 - b(w0)*w0/2) * zi);

% now we just gotta plot everything:
figure;

subplot(2, 1, 1);
plot(w / 2 / pi, abs(H), 'LineWidth', 1.5);
title('Magnitude and Phase of Transfer Function H(z)');
ylabel('|H(e^{j\omega})|');
grid on;

subplot(2, 1, 2);
plot(w / 2 / pi, angle(H), 'LineWidth', 1.5);
ylabel('\angleH(e^{j\omega})');
xlabel('Normalized Frequency');
grid on;
```

Appendix C: Calculating Butterworth Coefficients for a Digital Filter

The following is a short function that returns two vectors of coefficients that you can use to filter data. One set, alpha, multiplies previous values of the filtered signal; the other, beta, multiplies current and previous values of the measurements. This will return the DC-ish signals, so to filter out gravity in accelerometer measurements you'd have to subtract this from your original measurements.

```
%   To implement a filter with alpha and beta, you can use something like:
%
%   y(n) = sum(x(n:-1:n-m-1).*beta) - sum(y(n-1:-1:n-m-1).*alpha(2:end));
%
function [alpha, beta] = getbutter(f0, dt, m)
    if nargin < 2
        dt = 1;
    end

    if nargin < 3
        m = 3;
    end

    p = exp(1i*(2*(1:m).' + m - 1) * pi / 2 / m);

    w0 = 2*pi*dt*f0;

    alpha = [(1 - w0/2*p(1)); -(1 + w0/2*p(1))];
    for i = 2:m
        alpha = conv(alpha, [(1 - w0/2*p(i)); -(1 + w0/2*p(i))]);
    end
    alpha = real(alpha);

    beta = w0/2*[1; 1];
    for i = 2:m
        beta = conv(beta, w0/2*[1; 1]);
    end

    beta = beta / alpha(1);
    alpha = alpha / alpha(1);
end
```