The Filter Wizard
issue 28: Analyze FIR Filters Using High-School Algebra
Kendall Castor-Perry

In this first part of a two-parter, the Filter Wizard looks at some fundamental facts about FIR filter responses, using some basic math you might even remember from school. In the second part, he turns these facts into a neat technique for assembling an FIR filter 'from scratch' so that it has a useful frequency response.

Why use FIR filters, anyway? Now that we've introduced good IIR (infinite impulse response) filters into PSoC Creator's Filter tool, I've spent a lot of time telling people the ways in which IIR filters can be a superior choice. They often require significantly fewer processing cycles to deliver comparable filtering. There are tradeoffs between FIR and IIR forms, of course. But here's the reason that drove the choice for a recent project: FIR filters have a finite impulse response!

Now this might appear to be just a banal restatement of the acronym for the filter type – of course FIR filters have a Finite Impulse Response, duh. But there are times when this finiteness is just what you need. A filter with an impulse response that's strictly finite in length has a limited amount of 'memory'. Changes in the input signal that occurred at a time earlier than ("now" minus the duration of the impulse response) cannot possibly contribute to the latest output sample. If the input stabilizes to a constant value, the output will become exactly constant after a period equal to the duration of the impulse response. In other words, the settling time is finite, and known exactly.

In contrast, the output of an IIR filter is, in principle at least, affected by conditions in the input signal arbitrarily far back in the past. The output can also continue to change for an indefinitely long period of time after the input has stabilized. In a standard quantized digital version it's actually quite difficult to say for certain when the output will stop varying. In fact, sometimes it never does completely settle down, but fusses around for ever in a pattern called a limit cycle. One of those tradeoffs to examine one day. But back to the topic at hand.

Recently, colleagues requested a filter implementation for strongly suppressing the effect of both 50Hz and 60Hz AC line (plus their 2nd harmonics) pickup on some tiny sensor signals. We also needed to be able to strictly bound the settling time of the filter. For me, this settled (whoops, sorry about that) the IIR/FIR choice firmly in the FIR camp. Either form is possible on the Digital Filter Block hardware on the PSoC3 and PSoC5 devices I'm working with these days. To fit the required number of filter channels concurrently into the hardware, I knew that these filters would need to be quite 'small', and that I couldn't waste implementation resources through the use of an inefficient design methodology. I needed to make every filter coefficient count towards my response goal! The actual method will be revealed in part 2, and to make sense of it, we first need to contemplate the meaning of FIR filters and how we represent them.

How is an FIR filter defined?

An FIR filter is completely defined by an ordered set of values that multiply (or 'weight') progressively more delayed versions of the input signal. Delaying samples of a signal is easy, of course. Just store the values in a memory somewhere, and read them out later. It doesn't have to be a digital memory; an early form of FIR filter called a transversal filter didn't digitize the signal at all, it just stored the voltage directly on a small capacitor, which was looked at later. The charge-coupled devices that today are mostly found in image sensors were once used to create sampled analogue voltage memories strung together into delay lines – so the early FIR filters were in fact analogue! But I digress.

All these weighted, delayed versions of the input signal are summed together, and this gives the desired output signal. If the input signal to this process is an impulse that has a non-zero value at only one sample time, the resulting output from the filter – its impulse response – has an identical form to the sequence of coefficients. Figure 1 shows an example set of filter coefficients (a 15-tap FIR designed by the PSoC Creator tool), together with the gain and the impulse responses.

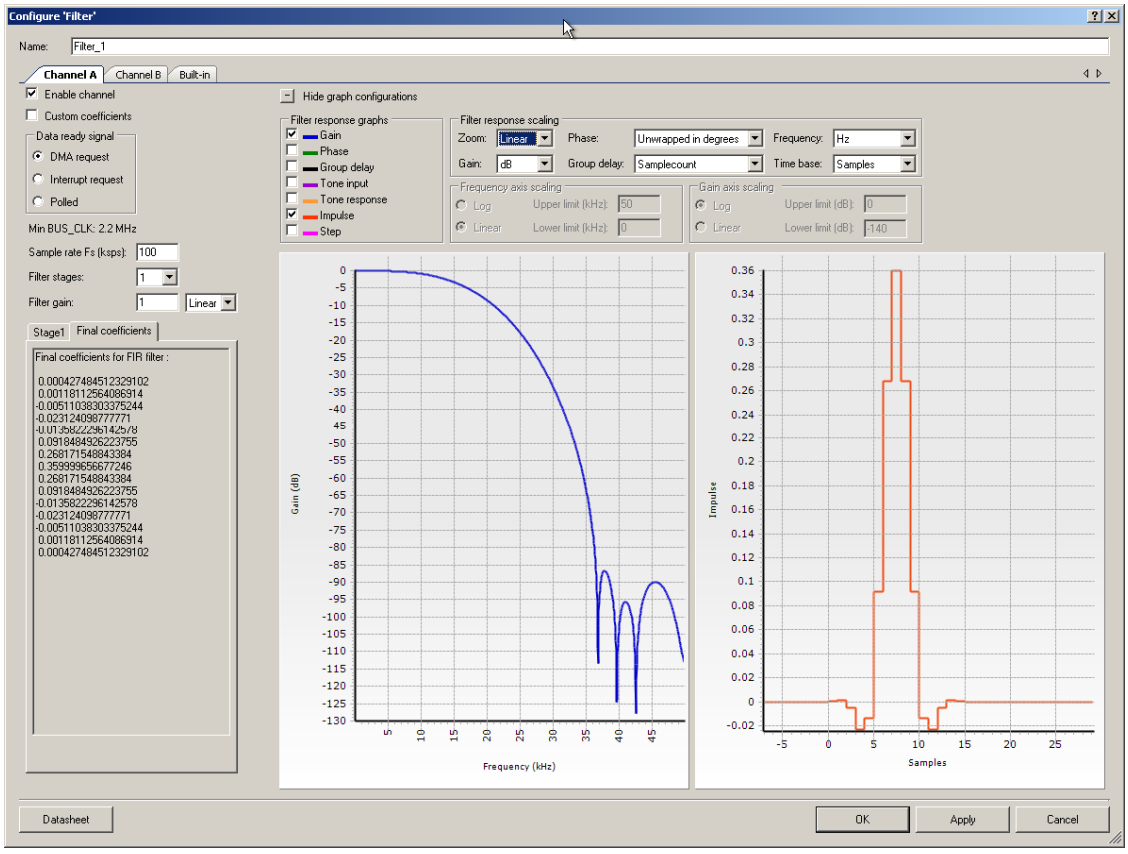

Figure 1: An example 15-tap FIR filter – coefficients, frequency and impulse responses.

Of course, many methods are available to calculate coefficients that deliver some desired filtering behaviour. Now, filter design is rather like cooking. You can get something partly or completely prepared by someone else and just pop it in the microwave, job done.

Or you can put a little more effort in and make something up from more basic ingredients, using more fundamental tools. You do get more appreciation for food if you have some feeling for the processes employed to deliver the sensations that you crave. And so it is for electronic design, especially for filters. Regular readers will recognize a repeating theme of mine here – roll up your sleeves and get "under the hood" occasionally. Sometimes you'll create something unusual that's just not available on the shelves of your local store. Other times, the main value is that you'll learn how to "call BS" (I think that's the correct American vernacular) on someone who's trying to pull some wool over your eyes (or your signals).

The good news is that you don't need to be a filter expert or a math nerd to make progress here. For example, as a chef (to push that cooking metaphor), you can exploit the Maillard reaction to make tasty dishes, without needing to understand its chemistry. And you can play around with some algebra to create cool filters even if you're not entirely comfortable with what it really means (yet <g>). And that's just what we're going to do right now!

Thinking about FIR filters as polynomials

So, here's the mental connection I want you to make: think of that ordered sequence of values, representing the coefficients of our FIR filter, as a **polynomial** in some variable, we'll call it z. A polynomial is just the sum of successive powers of our variable, each one multiplied by some coefficient. That's like we just defined our FIR filter to be.

When working with sampled signals and the systems that process them, we make heavy use of this variable z, and also its inverse, z^-1. It doesn't have a simple physical meaning, but it's intimately associated with time. z^-1 is particularly connected with the act of delaying a signal by a single sample period (mathematically, it's referred to as an operator). In a causal system, the output can only be influenced by things that have already happened, in other words delayed versions of an input signal. The consequence is that you'll mostly find **negative** powers of z turning up in filter equations. We can just multiply through by some large power of z to make the polynomials have a much more familiar form. You don't need to understand the derivation and use of the so-called z-transform in order to do practical things with polynomials in z and z^-1.

Here's the polynomial form of the filter shown in figure 1. To make the equation more compact, I've trimmed the number of significant digits in the coefficients. It's shown in both the negative power [1] and positive power [2] forms.

$$F(z) = 0.000427485z^{0} + 0.001181126z^{-1} - 0.005110383z^{-2} - 0.023124099z^{-3}$$
$$- 0.01358223z^{-4} + 0.091848493z^{-5} + 0.268171549z^{-6} + 0.359999657z^{-7}$$
$$+ 0.268171549z^{-8} + 0.091848493z^{-9} - 0.01358223z^{-10} - 0.023124099z^{-11}$$
$$- 0.005110383z^{-12} + 0.001181126z^{-13} + 0.000427485z^{-14} \qquad [1]$$

$$F(z) = 0.000427485z^{14} + 0.001181126z^{13} - 0.005110383z^{12} - 0.023124099z^{11}$$
$$- 0.01358223z^{10} + 0.091848493z^{9} + 0.268171549z^{8} + 0.359999657z^{7}$$
$$+ 0.268171549z^{6} + 0.091848493z^{5} - 0.01358223z^{4} - 0.023124099z^{3}$$
$$- 0.005110383z^{2} + 0.001181126z^{1} + 0.000427485z^{0} \qquad [2]$$

This might not seem like a big deal – isn't it just another way of expressing what we already know about that filter? Well, it permits us to use some powerful fundamental tools of algebra to analyze, and eventually to **synthesize** polynomials with useful properties. The key thing is that we can always **factorize** any such polynomial, rendering it as a product of individual terms that each have either linear or quadratic form. Does this yet bring back memories of those algebra classes at school?

To factorize such a polynomial, we need to find its **roots**. These are the values of the variable that cause the value of the polynomial to vanish. Most math manipulation tools will have functions that give you the roots of a polynomial. The roots will either be real (giving a linear term) or in complex pairs, which each multiply up into a quadratic term. The overall polynomial is equal to the product of all these quadratic and linear terms.

Let's take our example filter, find the roots and therefore the individual factors. I used an Excel root finder that seemed to work well.

real part	imaginary part
0.303342	0.046139423
0.303342	-0.046139423
-0.367424	0
-0.950358	0
-0.674441	0.738328724
-0.674441	-0.738328724
-0.891708	0.452610945
-0.891708	-0.452610945
-0.794906	0.60673263
-0.794906	-0.60673263
-1.052235	0
3.222062	0.490086903
3.222062	-0.490086903
-2.721648	0

Table 1: All the roots of the polynomial in equation [2].

There are fourteen roots – which is good, because it was a fourteenth order polynomial (the fifteenth tap is the constant term, i.e. it multiplies the zeroth power of z) – of which four are purely real and the rest come in complex conjugate pairs. Remember that classic formula for the solutions of a quadratic equation? When the expression inside the square root is negative, that's where the imaginary part of the root comes from, and the plus-or-minus sign tells you there are two of them, with opposite signs of the imaginary part.

Now we can write down the factors by multiplying up all the terms of the form (z - root). We bundle up those complex pairs into quadratic factors, which will have nice **real** coefficients, as we can show by multiplying out a conjugate pair of roots, x+jy and x-jy:

$$\begin{aligned} Q(z) &= \left(z-\left(x+j\cdot y\right)\right)\cdot\left(z-\left(x-j\cdot y\right)\right) \\ &= z^2 - x\cdot z + j\cdot y\cdot z - x\cdot z - j\cdot y\cdot z + \left(x^2+y^2\right) \\ &= z^2 - 2\cdot x\cdot z + \left(x^2+y^2\right) \end{aligned} \qquad [3]$$

Doing this on all the root or root pairs (pick the two that have the same real part) in table 1, we get equation 4:

$$\begin{aligned} F(z) &= \left(z^2 - 0.606685z + 0.0941454\right)\cdot\left(z^2 + 1.34888z + 1\right) \\ &\cdot\left(z^2 + 1.783416z + 1\right)\cdot\left(z^2 + 1.589812z + 1\right)\cdot\left(z^2 - 6.444124z + 10.62187\right) \\ &\cdot\left(z + 0.367424\right)\cdot\left(z + 0.950358\right)\cdot\left(z + 1.052235\right)\cdot\left(z + 2.721648\right) \end{aligned} \qquad [4]$$

Just to make sure of my work, I multiplied [4] out again, (well, Excel did), and figure 2 is the proof that we get back the same filter. By the way, this response was obtained by using Excel's FFT function directly on the impulse response. Yet another useful way of calculating the frequency response if you don't have a simulator to hand. This filter only has 15 interesting time points in its impulse response, of course; I padded it out to 1024 points with extra zero values, to get an FFT plot with nice close frequency spacing to give a smooth plot.

But you may be getting restive again – what have we learned by doing all this? Well, here's the useful thing. It turns out that the **product** of these factors represents the behaviour of a filter made up of a **series connection** of blocks, each one of which implements a small filter with a polynomial given by that factor. So by factorizing the FIR filter's big polynomial, we have developed a set of small filters (each with either two or three tap weights) that could be connected in series to give the same filter behaviour as the original filter.

Now, IIR filters are almost always designed as series connection of second-order (i.e. quadratic) pieces. But it's rare that people bother to look at the equivalent factorized decomposition of an FIR filter. That's because it generally doesn't confer any sort of advantage since FIR filters are already so simple to implement. But it's a great tool to **analyze** the coefficient sets that you get from your chosen FIR design software.

My main goal is not to get you breaking down someone else's FIR filter design using these tools. What's really interesting is to look at **the behaviour that the factors themselves exhibit**. We can then separately manipulate each factor to potentially make it do something we want. If we create some factors from scratch, each of which does something useful and interesting, and then multiply them all up to get a polynomial, then

this polynomial's coefficients are those of **an FIR filter that does all those interesting things at once**.

Figure 2: Response of the polynomial you get by multiplying equation [4] back up.

Just to tempt you, have a think about this. Our FIR filter has **three** nulls in the stopband (figures 1 and 2). And our factorization of the polynomial [4] shows that it has **three** factors that have a constant term (the coefficient of z^0) of unity. This is **not** a coincidence <g>. So next time, we'll work out how to drive this process, creating factors that contribute the particular stopband behaviour we want. This'll show you that there's more than one root (ouch) to good filter design! best / Kendall