This tutorial is going to be a crash course in going from a difference-equation representation of a filter to a filter using Pd's "raw" filters. The raw filters are [rzero~], [rpole~], [czero~], and [cpole~] (there's also [rzero-rev~] and company, but we won't be using them here). It won't go into detail on everything and will likely be overwhelming if you're new to filter design or the math involved, but hopefully it will get you started, help you understand some basic concepts needed for filter design, and/or at least clear up some things.

NOTATION

I'll first start with a simple first-order difference equation to explain the notation I'll be using and to give the basics of what you should be seeing in the equation:

$$y[n] = b_0 x[n] + b_1 x[n-1] + a_1 y[n-1]$$

$x[n]$ (read: "x of n") refers to input, and $y[n]$ refers to output. n is the current sample, $n-1$ is the previous, and so forth. b_m (read: "b sub m") are the coefficients of $x[n-m]$, and a_m are the coefficients of $y[n-m]$. Using b_m for $x[n-m]$ and a_m for $y[n-m]$ is the standard convention that you will typically see in academic papers.

So another way to read this equation is "the current output is b_0 times the current input plus b_1 times the previous input plus a_1 times the previous output". As such, a_m are commonly referred to as the feedback coefficients (since they multiply the output feeding back into the filter), and b_m are referred to as the feedforward coefficients.

One way to look at $x[n-1]$ is to think of it as the input delayed by one sample. Likewise, $y[n-2]$ would be the output delayed by two samples. So, as you can see, the m in b_m and a_m matches up with the number of delayed samples (i.e. $b_m x[n-m]$), and m is the delay in samples. The order of the filter is the maximum delay m. So if you had a difference equation with $x[n]$, $x[n-1]$, and $y[n-2]$, it would be a second-order filter, because the maximum delay is 2 samples.

Also, we're going to be discussing complex numbers here (numbers with a real and imaginary part). While you may have used $i = \sqrt{-1}$ in math classes, I will be using j instead. j is used in engineering instead of i, and since DSP is more of an engineering thing, you will come across j more often when researching this stuff.

RAW FILTERS

Okay, now that we have notation out of the way, let's quickly look at the difference equations for the raw filters.

[rzero~] and [czero~]: $y[n] = x[n] - b_1 x[b-1]$

[rpole~] and [cpole~]: $y[n] = x[n] + a_1 y[n-1]$

NOTE THAT THE SIGNS ARE DIFFERENT. If you don't remember this, it will bite you in the ass later. They are different in the difference equation, but they line up in the z-transform, which we will discuss in the next section. (Don't know why they're negative, though.)

The difference between [rzero~] and [czero~] is that [rzero~] uses only real numbers for b_1, and the input is also real. [czero~], on the other hand, uses complex numbers, meaning it has a real and imaginary part. The real and imaginary parts are represented as separate inputs. So for a complex number $x + jy$, x would go in the left inlet and y would go in the right (you don't have to calculate j...I mean, it isn't even real). The same is true with [rpole~] and [cpole~]. While most filters for audio are real filters, we will see later why the complex filters are important.

Z-TRANSFORM

The z-transform converts a difference equation in the time-domain to a transfer function in the z-domain. The usefulness of the z-domain is beyond the scope of this tutorial. But, you don't need to know everything about it for it to be useful here. Just accept that it works, just like you have to accept that imaginary numbers work and that infinity exists at least conceptually even though our brains are too small to comprehend, and you'll be fine. You'll be happy to know that doing the z-transform is much easier than wrapping your head around imaginary numbers, anyway. :-)

The basic representation of a generic filter in the z-domain is this:

$$H(z) = \frac{B(z)}{A(z)}$$

Again, $B(z)$ is read "B of z". The difference between square brackets [] and parenthesis () is that brackets represent signals with discrete points (like a digital sampling rate) and parenthesis represent continuous signals (like analogue). It's not a huge deal here, it's just another convention.

As you might have guessed, $B(z)$ is named because it represents the part of the difference equation with b_m coefficients, while $A(z)$ represents the part with the a_m coefficients. $H(z)$ is just another naming convention for filters. $h[n]$ is the impulse response in the time-domain, so $H(z)$ is the z-transform of the impulse response.

So, how do we get from the time-domain to the z-domain? Well, it's actually quite simple, but it's probably easiest to explain with an example. So let's use the biquad filter as an example. The difference equation of a conventional digital biquad is this:

$$y[n] = b_0 x[n] + b_1 x[n-1] + b_2 x[n-2] - a_1 y[n-1] - a_2 y[n-2]$$

(Sidenote: [biquad~] actually reverses the signs for the feedback coefficients)

Each of the samples in the equation gets replaced by an exponent of z, where the exponent is the delay length. Let's start by just figuring out $B(z)$. In this case, we get:

$$B(z) = b_0 + b_1 z^{-1} + b_2 z^{-2}$$

Pretty straight-forward. $x[n-m]$ becomes z^{-m}. ($z^0 = 1$, which is why b_0 stands alone here.)

Now for the feedback part: $A(z)$. With this part, $y[n-m]$ becomes $-z^{-m}$. Notice the signs get reversed. Also, a_0 is actually the number that multiplies $y[n]$, which is 1, and it's sign doesn't get reversed because it's on the left side of the difference equation. This gives us:

$$A(z) = 1 + a_1 z^{-1} + a_2 z^{-2}$$

So our final z-transform of the biquad filter is:

$$H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{1 + a_1 z^{-1} + a_2 z^{-2}}$$

So what we have is a second-order polynomial $B(z)$ over a second-order polynomial $A(z)$.

CONVERT TO FIRST-ORDER FILTERS IN SERIES

The biquad is a second-order filter. The raw filters in Pd are first-order filters. We need to represent this second-order filter using nothing but first-order filters. So, we need to break it down into first-order polynomials. To do this, we need to find the poles and zeros.

Well, let's back up a second. How can we even break this apart at all? As it happens, multiplication in the z-domain is the same thing as filters running in series in the time-domain. As a very simple example, look at it this way:

$$H(z) = \frac{B(z)}{A(z)} = \frac{B(z)}{1} \cdot \frac{1}{A(z)}$$

In this instance, $B(z)$ is one filter, and $1/A(z)$ is another. The signal goes through $B(z)$ first, then $1/A(z)$. **But it has exactly the same output as $B(z)/A(z)$.**

Since we're working with polynomials here, we need to factor them out into first-order polynomials to get first-order filters in series. To do this, we need to find the roots. The roots of $B(z)$ are the zeros, and the roots of $A(z)$ are the poles. It is generally less confusing to find them if we rewrite them as positive exponents of z instead of z^{-1}. We can do this by multiplying $H(z)$ by $z^2/z^2 = 1$:

$$H(z) = \frac{b_0 z^2 + b_1 z + b_2}{z^2 + a_1 z + a_2}$$

Next, we need to factor out b_0 from $B(z)$. It's not necessary for finding the roots, but it will be needed later for implementation purposes. This is simply a matter of dividing it by b_0, which we will rename g for gain. We'll rename b_m/b_0b_m/b_0 to β_m.

$$H(z) = g\left(\frac{z^2 + \beta_1 z + \beta_2}{z^2 + a_1 z + a_2}\right)$$

What we now have is a quadratic equation in both the numerator and the denominator. Finding the roots of those equations (i.e., the value of z that will make them equal zero) can simply be done using that quadratic formula we all thought was useless in high school:

For $y = ax^2 + bx + c$, the roots are

$$y = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

This translates for the zeros to:

$$q = \frac{-\beta_1 \pm \sqrt{\beta_1{}^2 - 4\beta_2}}{2}$$

and the poles to:

$$p = \frac{-a_1 \pm \sqrt{a_1^2 - 4a_2}}{2}$$

Note that there are two answers each because of the ±. Also note that if $b^2 - 4ac < 0$, you are going to end up with complex numbers as your zero or pole, and this is going to happen often. Once we find the zeros/poles, $H(z)$ becomes:

$$H(z) = g\left(\frac{(z-q_1)(z-q_2)}{(z-p_1)(z-p_2)}\right) = g\left(\frac{(1-q_1z^{-1})(1-q_2z^{-1})}{(1-p_1z^{-1})(1-p_2z^{-1})}\right)$$

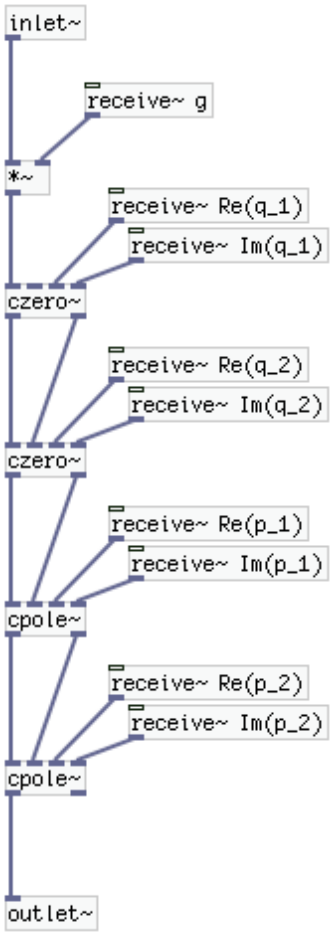
The last step just undoes the z^2/z^2 multiplication we did earlier to bring us back to negative powers of z. But the answers are the same. If you substitute q_1 for z, you will get 0 in both versions.

AND NOW FOR THE PD IMPLEMENTATION

As stated earlier, multiplication in the z-domain is the same thing as running filters in series. We now have four first-order polynomials and a gain. This is what we need to use the raw filters. Breaking it down into multiplications, we get:

$$H(z) = g(1-q_1z^{-1})(1-q_2z^{-1})\left(\frac{1}{1-p_1z^{-1}}\right)\left(\frac{1}{1-p_2z^{-1}}\right)$$

Assuming the poles and zeros are complex, this is the same thing as:



With a real filters, of course, you would just use the real versions and get rid of the imaginary patch cords. But with typical higher order audio filters, the zeros and poles will come in complex conjugate pairs, so you can expect to use the complex filters. Besides, a real number $x = x + j0$ anyway, so the complex ones will work in both situations.

With filter orders greater than 2, the quadratic formula obviously doesn't hold. However, it is quite common for high-order filters to be implemented as a series of biquads because they are easier to work with that way (Butterworth and Chebychev filters work nicely as biquad series). So the formula may still be useful. In addition, programs like Octave or Matlab have functions that can find the roots of arbitrary polynomials for you for those difficult cases.

PHEW!

That might be a lot to swallow in such a short space. For something more thorough, I recommend the online book The Scientist and Engineer's Guide to Digital Signal Processing by Steven W. Smith, which actually uses very

accessible language despite the title. It's also more than just filters. I should point out, however, that the author does go against convention and switches b_m and a_m, so watch out. But other than that, it's pretty great. Julius O. Smith III also has a good book on digital filters online. It's a little less accessible, but there's a lot there.

Hopefully that didn't suck?

.mmb