DEMYSTIFYING DIGITAL FILTERS

CYCLING ’74

PUBLISHED BY CYCLING ’74

https://cycling74.com/

First published, March 2021

Contents

Part One: Key Concepts of Digital Filters

In this section, we will introduce the key concepts we use when we talk about digital filters.

- systems and signals
- linearity and time invariance
- time and signal domains for filtering
- impulse responses
- convolution versus recursion

What is a filter anyway?

A filter is a system that produces an output signal based on an input signal. As DSP guru Julius O. Smith III puts it,

> Any medium through which the music signal passes, whatever its form, can be regarded as a filter. However, we do not usually think of something as a filter unless it can modify the sound in some way. For example, speaker wire is not considered a filter, but the speaker is (unfortunately). The different vowel sounds in speech are produced primarily by changing the shape of the mouth cavity, which changes the resonances and hence the filtering characteristics of the vocal tract. The tone control circuit in an ordinary car radio is a filter, as are the bass, midrange, and treble boosts in a stereo preamplifier. Graphic equalizers, reverberators, echo devices, phase shifters, and speaker crossover networks are further examples of useful filters in audio. There are also examples of undesirable filtering, such as the uneven reinforcement of certain frequencies in a room with "bad acoustics." A well-known signal processing wizard is said to have remarked, "When you think about it, everything is a filter."

When we talk about filters as musicians and Max users, we're usually referring to systems that modify the frequency content of incoming signals. That's not the case for all filters — for example, an allpass filter only changes the phase of its inputs rather than its frequency.

As a Max user, you're already familiar with the idea of a signal as a stream of floating point numbers. A system *transforms* that signal — you can think of a system as a function or an algorithm.

In DSP textbooks, you'll usually see a diagram that shows a system as a box in between two waveforms, an inlet signal x and an outlet signal y.

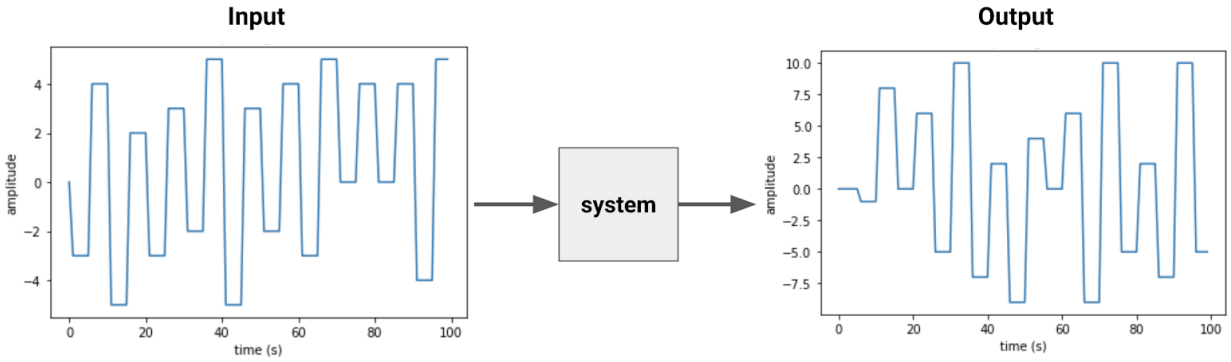

Figure 1: A system acts on an input and returns an output.

Time and frequency

If you're the user of a filter, it shouldn't matter much to you what happens to a signal in the time domain — you're more interested in what happens to the signal in the frequency domain. When patching or programming, though, you're more likely to be working in the time domain. With that in mind, it's a good idea to make sure you know how to travel between the two domains.

To do that, you should know how to use the **Fourier transform**. Chances are you've heard of the FFT (Fast Fourier Transform), which is just what it sounds like – a faster way to do a Fourier transform. If you're unfamiliar with FFTs and would like a better intuition for them, you can check out this video by Grant Sanderson of 3Blue1Brown and this interactive website by Jack Schaedler. To see more on the practical details as a Max user, head over to the tutorials on the `fft~` and `ifft~` objects in the Cycling '74 MSP analysis tutorials as well as the Advanced Max FFT video series.

In short, though, the FFT is our ticket to translate between time-based and frequency-based representations of the same signal. Keep this in mind because it will come in handy soon!

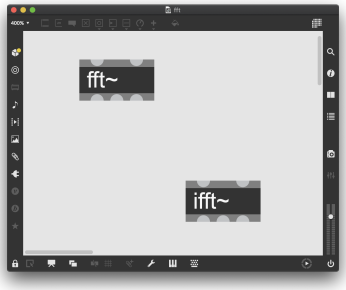

Figure 2: The `fft~` and `ifft~` objects in Max are used to accomplish Fourier Transforms with the FFT.

Linear Time-Invariant (LTI) systems are our friends

When talking about systems, the properties of **linearity** and **time-invariance** are useful in calculations. Fortunately, many real-world systems (and filters) have both of these properties and make the math

surrounding them much easier to handle. Specifically, LTI systems allow us to work easily with Fourier analysis, one of the most useful pieces in a DSP toolkit.

Linearity

For a system to be **linear**, it has to satisfy two properties: **scaling** and **superposition**.

1. Scaling

The output can only be multiplied by a scalar. This means that you can only multiply the input by a constant value, like a function $f(x) = -2x$ or $g(x) = 0.1x$. A function like $h(x, y) = xy$ or $p(x) = sin(x)$ does not satisfy this property.

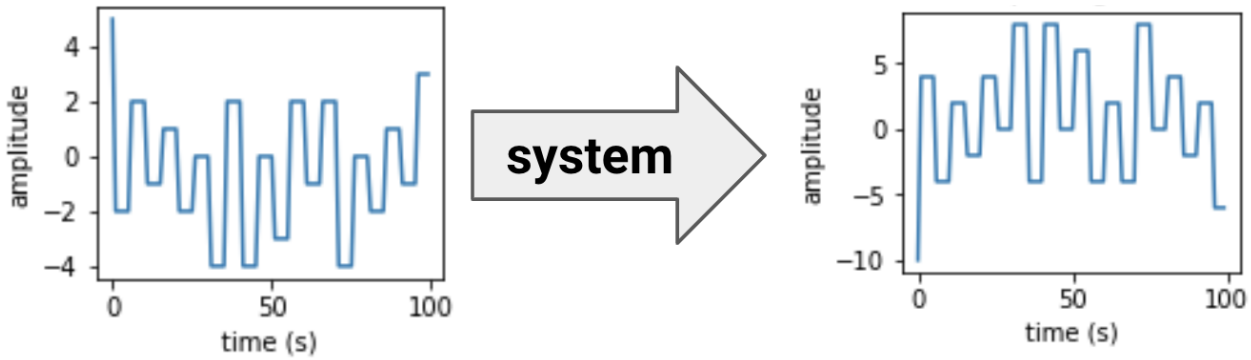

Figure 3: The scaling property.

2. Superposition

It should not matter if you add two signals before or after the system. The final result should be the same.

The nice thing about linearity is that it makes it easy to break apart the pieces of an input and do computations separately on those pieces before we bring them back together at the end.

Time-invariance

The property of **time-invariance** might sound confusing at first. How can anything interesting happen to a signal if nothing changes over time? Careful! We're only making sure that the system (our function or algorithm or black-box, depending on how you like to think about it) doesn't change over time, not the actual data going through it.

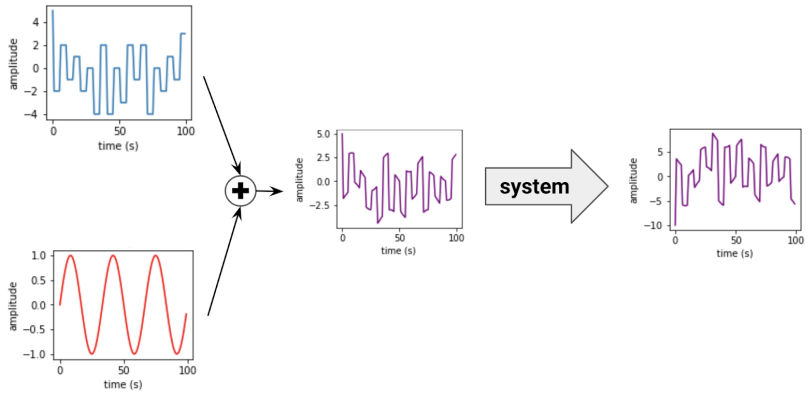

Figure 4: Superposition: Adding inputs before the system.

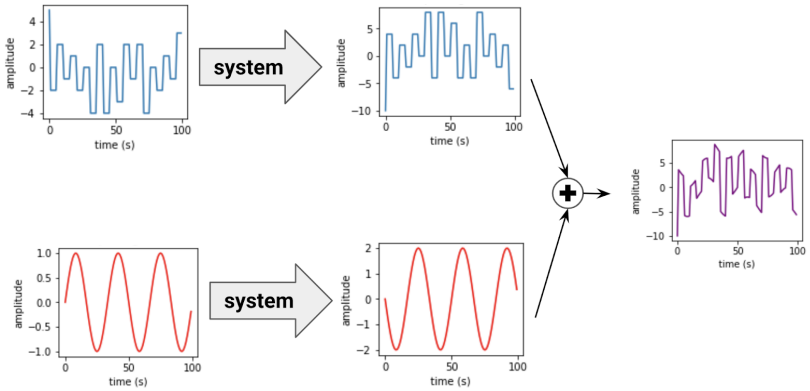

Figure 5: Superposition: Adding inputs after the system.

Time-invariance is sometimes called **shift invariance** because it means you can shift where a signal is in time, and it won't change how the system treats it.

Impulse, frequency, and step responses

When we talk about a filter's performance, we often look for its **impulse response**. In other words, we want to describe what the output signal of a filter looks like when it's sent a single, extremely short spike.

The reason an impulse response makes a good "test signal" is that an infinitely short pulse in the time domain is infinitely wide in the spectral domain. This means an impulse contains every possible frequency.

To be technically correct, since our actual impulse has to take up at least one sample, it cannot be infinitely narrow and thus doesn't actually contain every frequency. However, it is close enough for real-world applications.

The usefulness of an impulse response becomes clearer when we're talking about our pal the LTI system. In an LTI system, the impulse response contains all the information about the filter.

From the impulse response, we can take the Fourier transform and

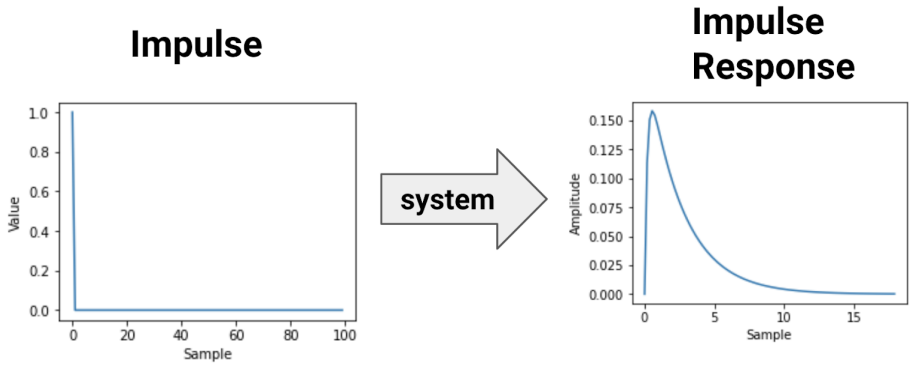

Figure 6: Finding an impulse response.

find the **frequency response** of the filter, which you may be familiar with.

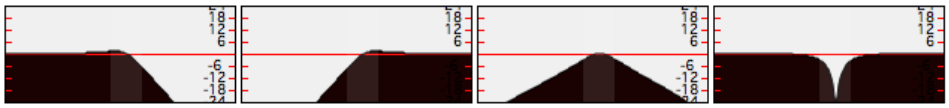

Figure 7: Image from the Max tutorials filters chapter.

If you take the integral of an impulse response (or more accurately in the world of digital signals, take a running sum), you can also find the **step response** which shows how the system reacts to sharp transients. You can imagine this as feeding in many `0`'s to the system then abruptly switching to all `1`'s and seeing what the output looks like.

As mentioned earlier, all of these representations contain the same information — they're just *presented* differently. A comparison of these representations is shown in Figure 8.

With all of these different representations, we can find out how well our filter does the job we want it to do. For example, if we want a filter that has a steep roll-off, we can check how well it does that by looking at the frequency response. As we learn more about filters, we'll see that there is no one filter that works for every job, but every job can have a filter tailored for it.

How do you make a digital filter?

You may or may not be surprised to learn that all a filter does is take a weighted average of the buffer of samples moving through it! There are two ways to do this: through **convolution** or through **recursion**.

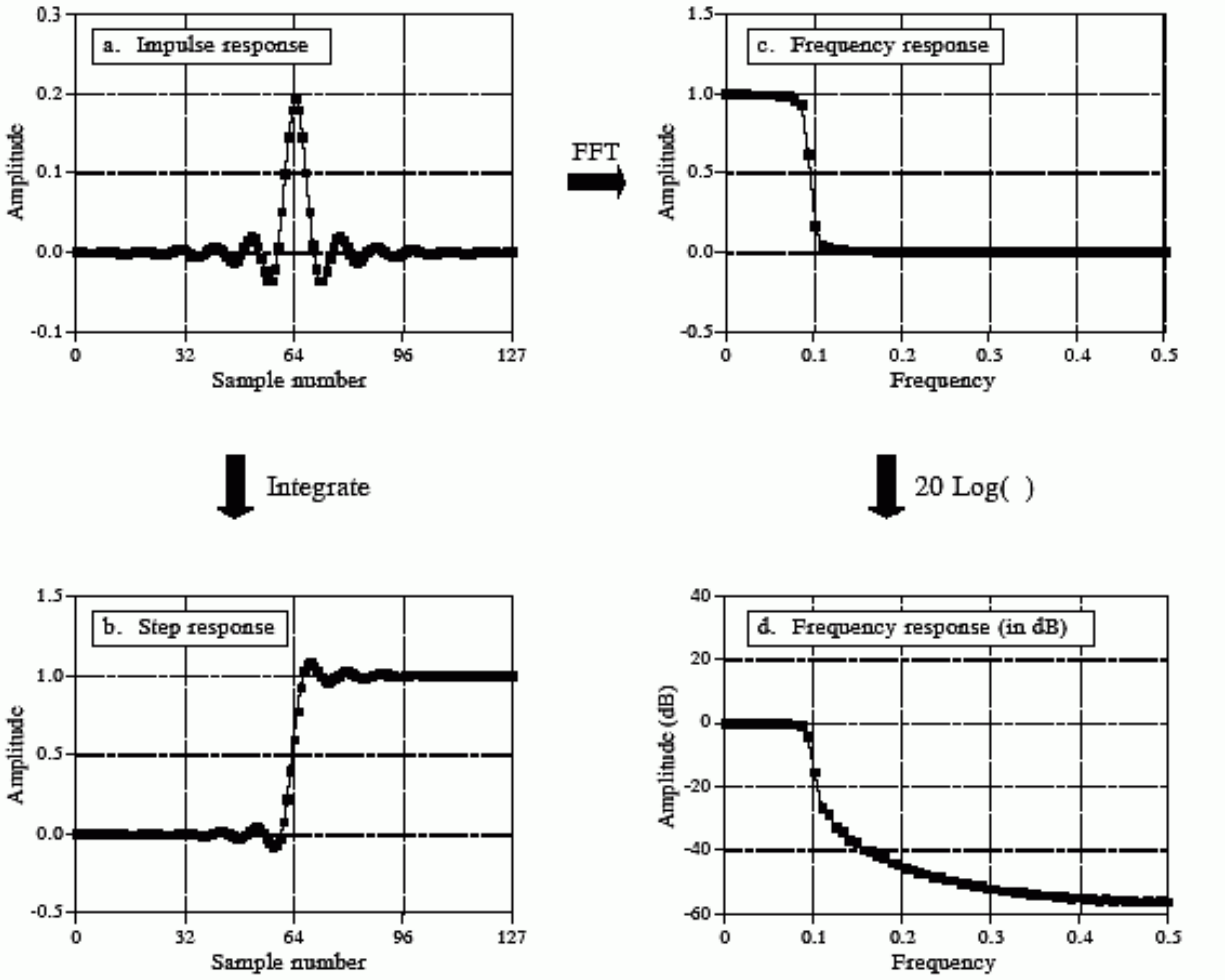

FIGURE 14-1
Filter parameters. Every linear filter has an impulse response, a step response, and a frequency response. The step response, (b), can be found by discrete integration of the impulse response, (a). The frequency response can be found from the impulse response by using the Fast Fourier Transform (FFT), and can be displayed either on a linear scale, (c), or in decibels, (d).

Figure 8: Source: The Scientist and Engineer's Guide to Digital Signal Processing, copyright ©1997-1998 by Steven W. Smith. For more information visit the book's website at: www.DSPguide.com

Convolution

We say that the input signal x is convolved by a **kernel** associated with the system to get the output y.

This kernel is the way we weight the input samples in our weighted average.

In the example illustrated by Figure 9, the blue boxes are samples whose values are known, and white boxes are samples that aren't known. For each output sample in y, the kernel slides along, lining up its rightmost box with the output sample to calculate. Then, for each matching pair of kernel values and input values, these are multiplied together and then added to get the output value. Once this is complete, the kernel moves to the right another sample and does the calculation again.

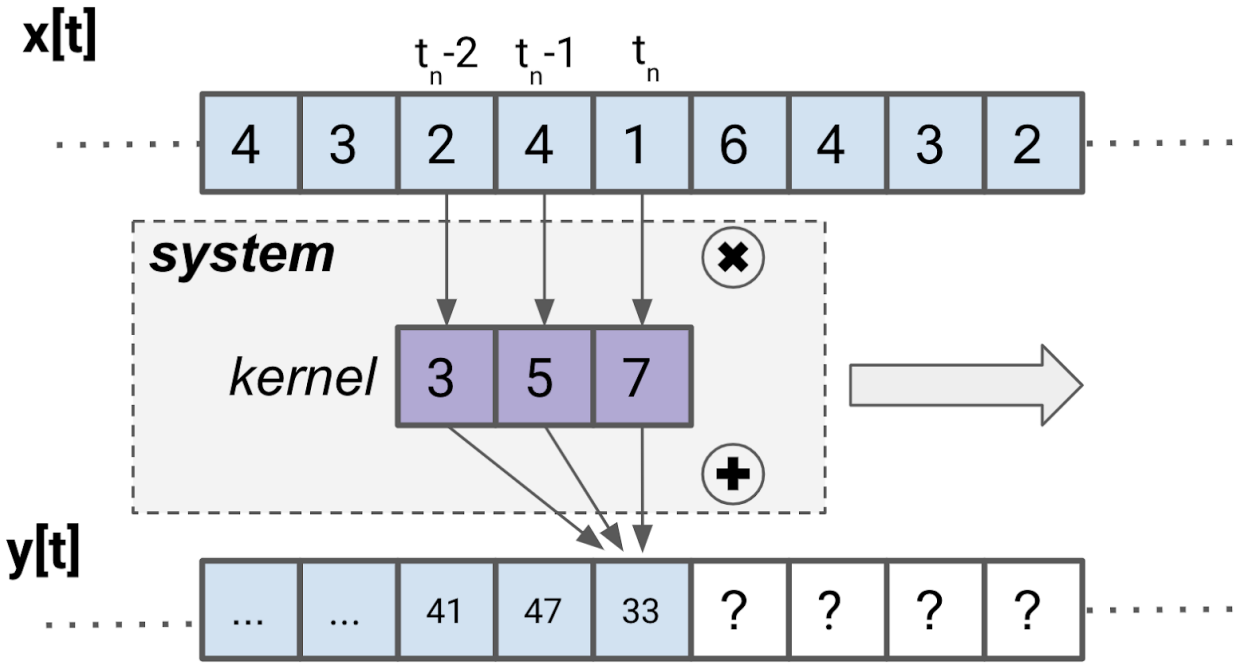

Figure 9: Convolution, illustrated.

Some questions to ponder about this illustration to check your understanding:

1. Where do the values of 41 and 47 come from?

2. What would the next value in the output signal be once we shifted the kernel to the right one sample?

3. Why does the illustration not show the leftmost two output boxes' values? What information would you need in order to calculate them?

4. The values of the output are much larger than the input's. What change would you make to the kernel to make the values more similar in size? (Hint: think about percentages.)

You may be interested to see the corresponding block diagram for this convolution example (shown in Figure 10).

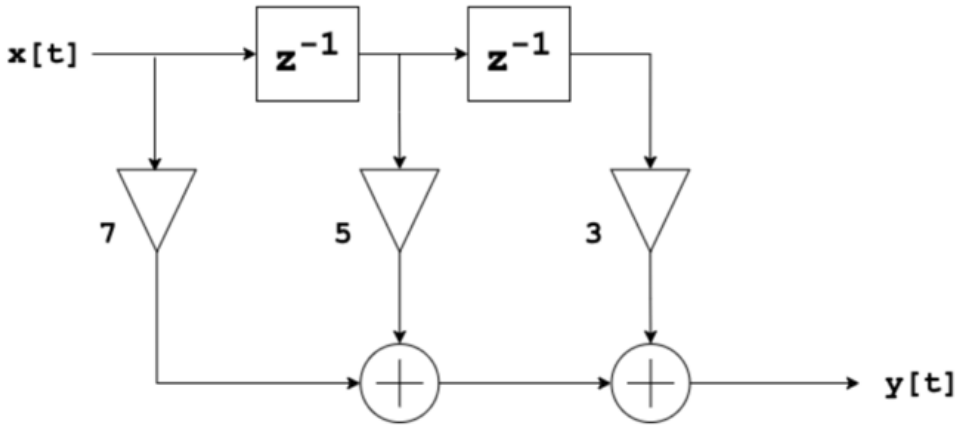

Figure 10: A block diagram for the convolution example in Figure 9.

Now that we've looked at one concrete example, let's try to generalize the convolution algorithm into an equation.

First, start with the additions and multiplications.

$$y[t_n] = 7 \cdot 1 + 5 \cdot 4 + 3 \cdot 2 = 33$$

To generalize, you can substitute in variables for the values of x.

$$y[t_n] = 7 \cdot x[t_n] + 5 \cdot x[t_n - 1] + 3 \cdot x[t_n - 2]$$

Then do the same thing for the kernel h's values.

$$y[t_n] = h[0]x[t_n] + h[1]x[t_n - 1] + h[2]x[t_n - 2]$$

If you wanted, you could stop here. However, to make this generalized equation more compact, you can use summation notation. Summation notation is useful shorthand for writing out long, repetitive sums.

In our example, the sum is short enough to write out, but there are times when you're adding hundreds of numbers that follow a pattern and you wouldn't want to write that out fully. Since long sums show up everywhere in DSP, it's worthwhile to take a short detour to learn how they work. (If you're already familiar with summations, feel free to skip down to the final form of our equation.)

Notice that there is a pattern in the equation we just wrote: the first term uses the first value of h (at index 0) and the rightmost value of x (at index t_n), the second term uses the second value of h (at index 1), and the second to rightmost value of x (at index $t_n - 1$), and so on. If you assign a variable n to the index, then you'll see in our example that n ranges from 0 to 2. Written in summation notation, this is:

$$y[t_n] = \sum_{n=0}^{2} h[n]x[t_n - n]$$

In this notation, the index is defined as n and initialized to 0 and written below the sigma (Σ) sign. The end of the range is written above the sigma.

To finish reaching a general equation for convolution, let's write what the sum would look like if the kernel size was also a variable. Let's call it N. Since we're describing any t, not just the specific t_n of our example, let's drop the subscript, too.

$$y[t] = \sum_{n=0}^{N-1} h[n]x[t - n]$$

Because convolution is such a useful operation, you can also use the "star" ($*$) operator to write this equation in an even shorter way.

$$x[t] * h[t] = \sum_{n=0}^{N-1} h[n]x[t - n]$$

There is a difference between seeing a star between two functions and two variables or values. In the first case, that means convolution, but in the second case, it means multiplication. Confusing, yes, but it's what mathematicians have stuck with. Usually you won't see the star being used to mean multiplication in DSP papers/books/articles/etc., though, so the possible confusion isn't as likely as you might fear.

The reason for looking at these kinds of generalized equations is because it's an easy way to describe filters in a mathematical way without drawing any diagrams. These kinds of equations describing filters are called **difference equations**. (You'll see why they're called "difference" equations rather than "summation" equations later on in this tutorial when we talk about IIR filters.)

Impulse responses of convolution filters

With notation out of the way, let's return to the filter we've made and illustrated. To see how the filter performs, remember that impulse responses are the best way to learn about a system. Let's calculate a few samples of the impulse response by hand.

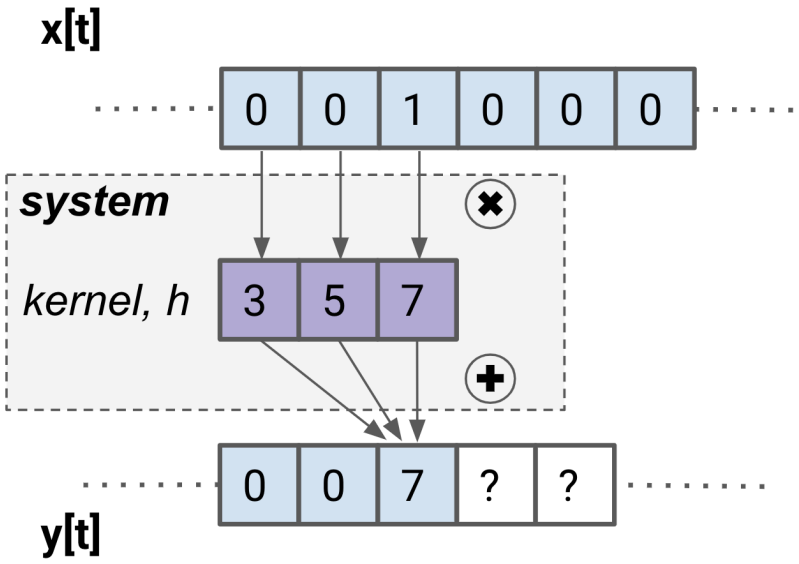

Figure 11: Computing an impulse response, step 1.

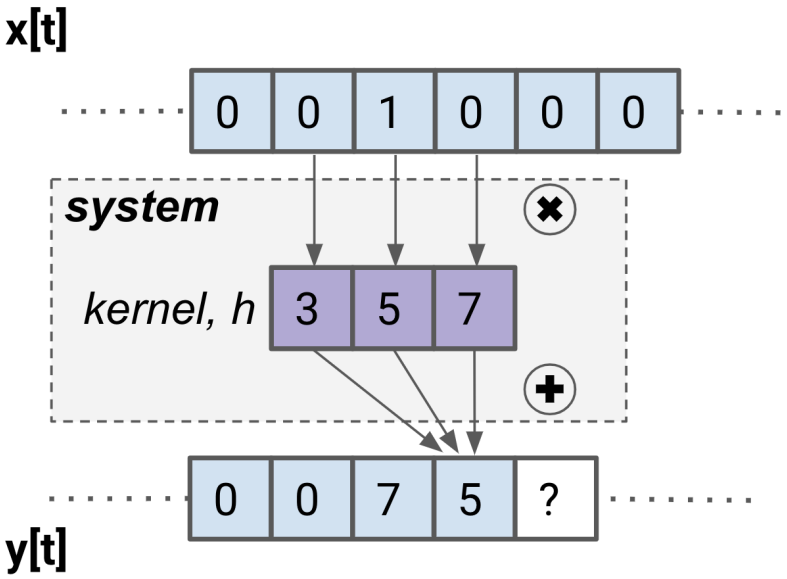

Figure 12: Computing an impulse response, step 2.

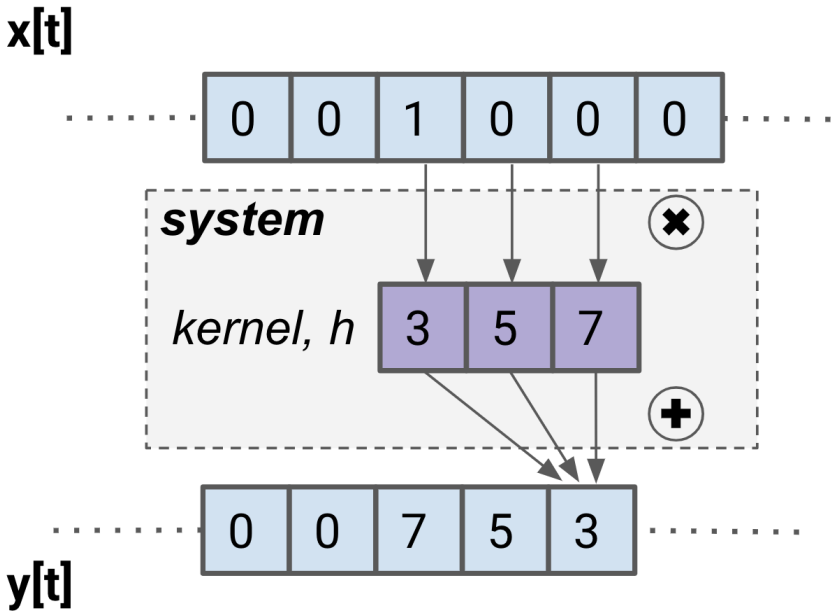

Figure 13: Computing an impulse response, step 3.

From this exercise, you can see that by convolving an impulse with the kernel, you get a backwards version of the kernel as the output! You may recognize that this isn't just luck – for any LTI system, the kernel comes directly from the impulse response.

Armed with the knowledge you've gained so far, you can start to think about the ways we can design a filter that uses convolution. In our first example, we tried playing around with values of the kernel and looked at the outputs we could get. Our other option is to go in the other direction – start from the output signal we want and compute the kernel.

To do that, remember that you can use the Fourier transform to find the frequency response from the impulse response. Often you know the frequency response you want, so it should be possible to find the filter kernel from there. The steps are as follows:

1. Create the desired frequency response curve.
2. Compute the impulse response using the FFT on the frequency response.
3. Use the impulse response as a convolution kernel.

This kind of filter (one that uses convolution) is called a **Finite Impulse Response (FIR)** filter. The easy way to remember this is that an impulse response has to be a finite length to do convolution with

it. The corollary of this is often used in textbook definitions of FIR filters–impulse responses always decay to zero in finite time when using FIR filters.

We will go more in depth on FIR filters in Part 2, but until then, let's take a brief look at the other kind of filters, **recursive** filters.

Recursion

Readers with a sharp eye may have noticed something about the convolution algorithm and the FIR block diagram – the output only uses the input samples. This means that we're only using feedforward loops, never feedback.

If you were to use both feedback and feedforward, you would be making a recursive filter. Notice that for recursive filters, we use the terms **feedback coefficients** and **feedforward coefficients** instead of **kernel**.

This difference in terminology is tied to convolution calling its set of coefficients a kernel.

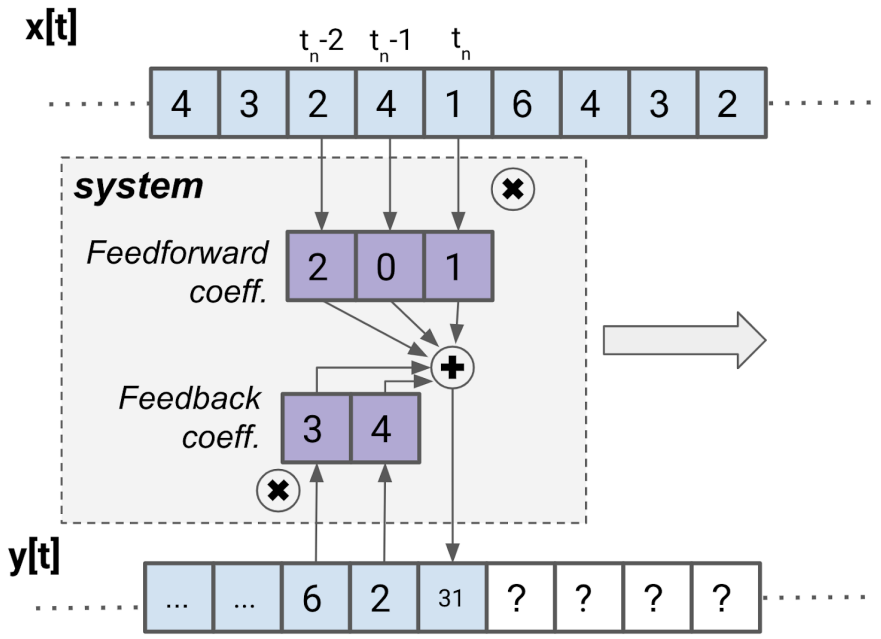

Figure 14: Recursion, illustrated.

You've likely seen block diagrams for recursive filters before if you've looked at biquad filters, such as the one in Figure 15.

Once feedback loops come into play, there's no guarantee that the impulse response will decay to zero in finite time any longer. These filters with infinitely decaying impulse responses are called **Infinite Impulse Response (IIR)** filters. When doing your own research, you may also hear them called recursive or feedback filters. We will learn much more about IIR designs in a later section.

This isn't always the case, though. There are some situations where recursion can be used to create FIR filters, but that's a topic for another time.

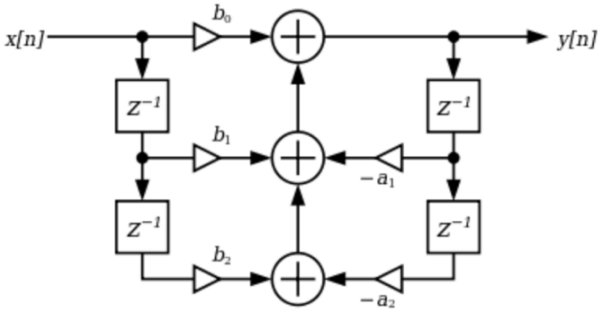

Figure 15: A biquad filter block diagram.

Part Two: FIR Filters

In the first part, you learned that filters are systems that act on signals. If the system is linear and time-invariant, you can use impulse responses to entirely characterize the system, too. You saw that convolution and recursion are two algorithms that give us two varieties of filters: **FIR** (finite impulse response) and **IIR** (infinite impulse response).

This next part will explore FIR filters in more depth. It will cover:

- The advantages and disadvantages of FIR filters
- Design and implementation of FIR filters
- The windowing method
- A brief look at the optimum-design method
- Resources for FIR filtering in Max

FIR filter advantages and disadvantages

Before you dive into making an FIR filter, it's worth knowing what they do well and what they do poorly.

Advantages

- It's easy to design an FIR filter for linear phase response
- The stability of an FIR filter is guaranteed (there are no feedback loops!)

Disadvantages

- They're computationally intensive (convolution is a computationally expensive operation)
- It's harder to change the filter parameters in real time

Designing and implementing FIR filters

In the last part of this tutorial, you were introduced to the general process for designing an FIR filter:

1. You create a desired frequency response — that involves drawing out the frequency response curve, sampling it, and put that result into a vector of values

2. Compute the FFT of the frequency response to get the impulse response

3. Use the impulse response as the filter kernel and convolve the input with the impulse response you found in step 2

The three-step procedure described above is theoretically sound, but when you start getting into the implementation, there's a problem that arises — for nearly all situations, the impulse response you will compute in step 2 will be infinite, rather than finite.

Why does this happen, and what do we do about it?

A brick-wall lowpass filter

To see this issue firsthand, let's consider an example.

Suppose you want to create a brick-wall lowpass filter (in other words, a filter with a perfectly flat passband, infinitely narrow transition band, and perfectly flat stopband).

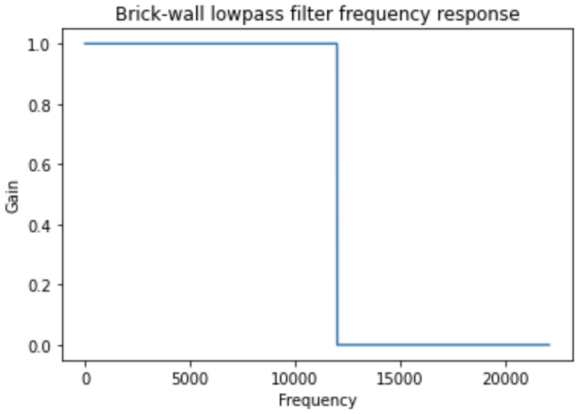

Figure 16: A brick-wall lowpass filter.

To get the impulse response, you'd use the FFT. (Since the details of the FFT are not the focus of this tutorial, we'll skip straight to the results.)

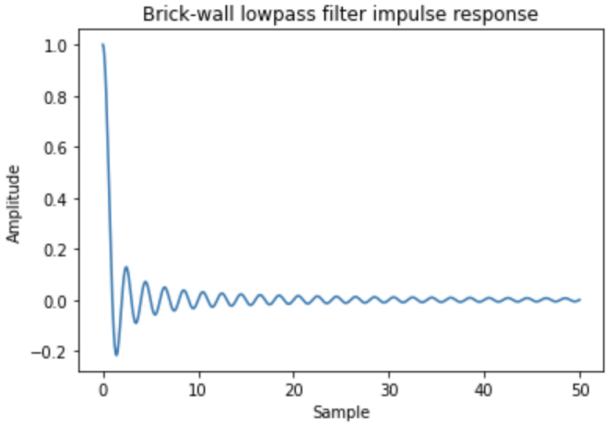

Figure 17: The impulse response of a brickwall lowpass filter.

This particular function appears so often in Fourier analysis that it has its own name: the **sinc** function. This function shows up all over in DSP texts, so it's worth knowing. The equation for the sinc function is

$$\text{sinc}(x) = \frac{\sin(x)}{x}$$

There are two important characteristics of the sinc function to consider.

1. It has an infinite domain. In other words, it outputs a value for every x.

You might notice that if $x = 0$, it appears that you'll run into a division by zero. Mathematically, this is no issue, and it can be proved with a little calculus that the result is actually 1, not undefined.

Note that a computer will not have the insight that $x = 0$ is a valid input, so if you're coding up your own sinc function, you'll need to handle this special case.

2. Related to the point above, the sinc function is "non-causal," meaning it has non-zero values for negative values of x.

A non-causal impulse response of a system indicates that it is using feedback loops (and thus depends on samples in the future). This means the system is recursive.

Knowing these two characteristics of sinc (and thus the characteristics of the impulse response of our brick-wall filter), you should now see how you might end up with a filter kernel that is unsuited for an FIR filter.

This is just one example, though, and it is healthy to remain skeptical of how likely you'll end up in this situation! To quell those concerns, you can research more about the duality property of the

Fourier transform. To summarize the property, finite signals in one domain (like the frequency response we designed above) become infinite in another (like the sinc function) and vice-versa.

Now that you have an idea of why the issue will arise, let's explore one of the most common solutions - windowing.

Windowing method

You can choose to "zero out" parts of the impulse response by multiplying the impulse response by a **window function**, a signal full of zeros except for a section of interest. By multiplying a signal with a window function, the result will look like a finite version of the original signal.

As an illustration, take a look at Figure 18 with an arbitrary sinusoidal function (an infinite signal), a rectangular window, and the result of multiplying the two together.

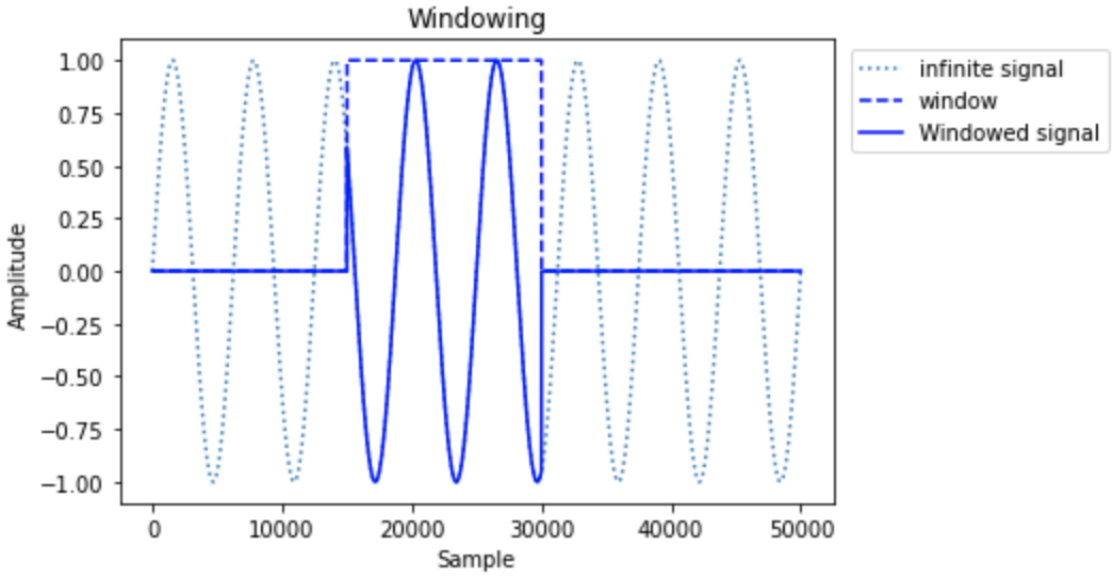

Figure 18: An example of windowing an infinite signal.

You may wonder why you need to bother with the idea of a window function at all when you could just simply "crop" the infinite signal. The reasoning goes back to a property of linear time-invariant systems — multiplication in the time domain is convolution in the frequency domain, and vice versa. This means when you crop a sinc impulse response by multiplying with a rectangular window, you have just convolved the frequency response with the same rectangular window! In other words, *by windowing the impulse response, you're*

going to change the original frequency response, too.

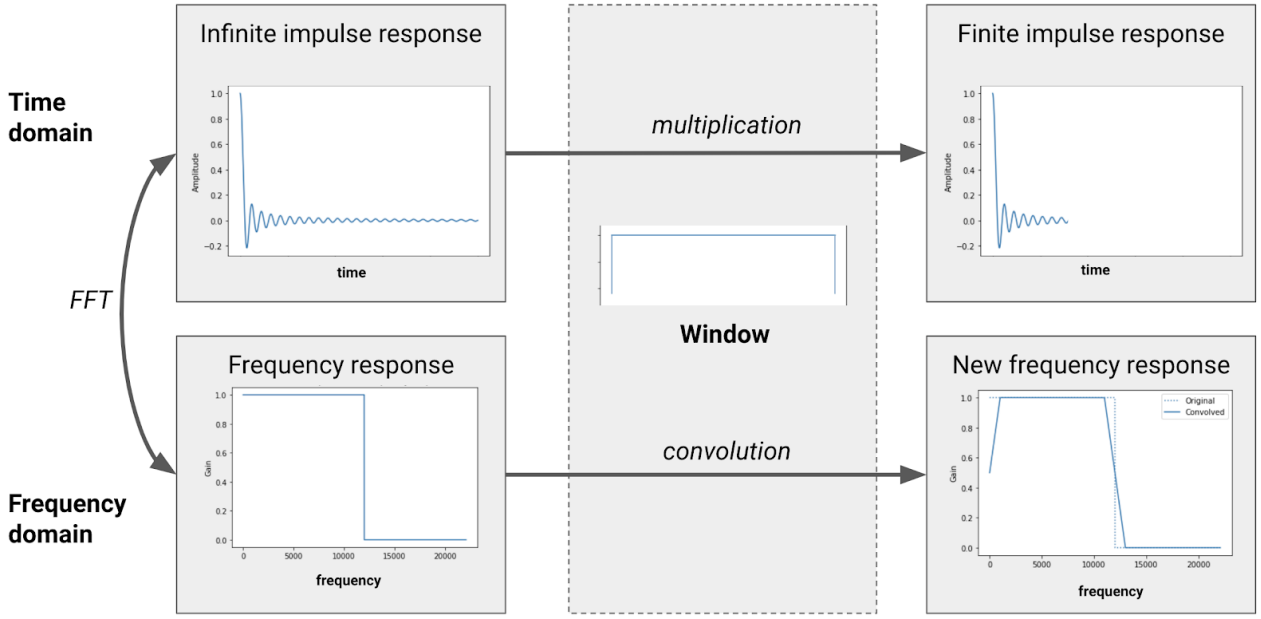

Figure 19: Changing the impulse response by windowing affects the frequency response.

All is not lost, though! If you are careful about selecting a window function, you can minimize the effects on the new frequency response and keep your original design.

There are many window functions to choose from, each one with their own optimizations.

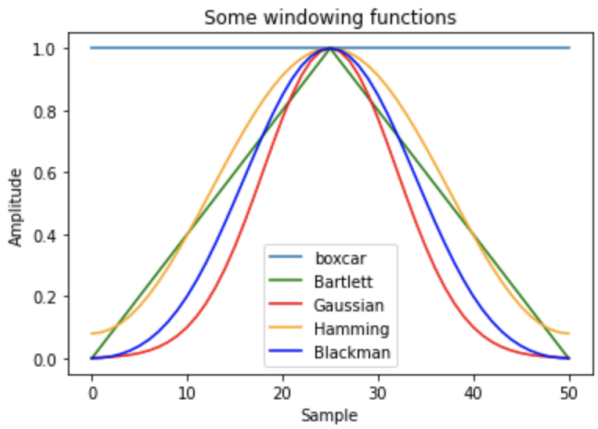

Figure 20: Some window functions.

You should notice that a common characteristic of window functions is their tapered edges. Much of window design comes down to just how this taper occurs, and sometimes window functions are

simply called taper functions.

To help pick from all of those window functions, you can use the same tools you've learned about already — impulse and frequency response analysis. Let's analyze the responses of two popular windows, Hamming and Blackman.

Pro-tip: If you're interested in understanding why tapering is so crucial, you'll want to read about the Gibbs phenomenon.

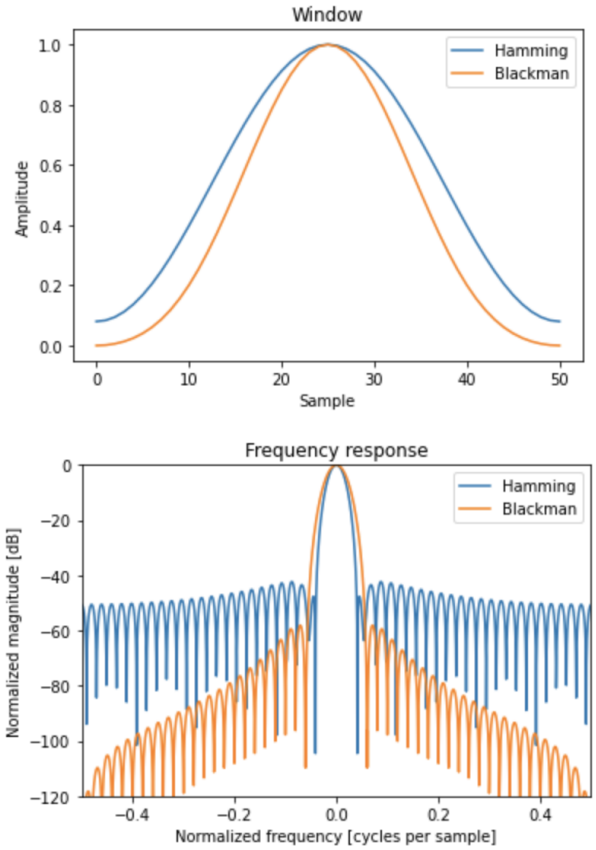

Figure 21: The Hamming and Blackman windows.

First, notice that the Blackman window has better stopband attenuation — at frequencies further from the "main lobe," the gain drops off more quickly. The tradeoff for this is that the frequency response of the Blackman window will have a slower roll-off than the Hamming window as shown in Figure 22.

Another factor to keep in mind is the window length. The longer of a window you use, the more of the original signal will remain (so you'll be using more of that impulse response you worked for earlier!), which will give you more accuracy.

The tradeoff for this is computation speed and delay — the larger your window, the larger your filter kernel will be and the longer you'll have to run expensive convolution computations and the longer of a delay line you'll need.

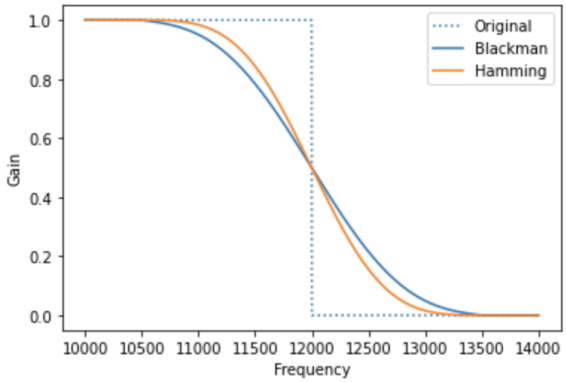

Figure 22: Comparing the frequency response of an ideal brick-wall lowpass filter with the Hamming and Blackman windowing results.

Let's say you decide that you're comfortable with a slightly slower rolloff in exchange for better stopband attenuation. With that in mind, you choose the Blackman window.

Now you're ready to apply the window to the sinc impulse response from earlier.

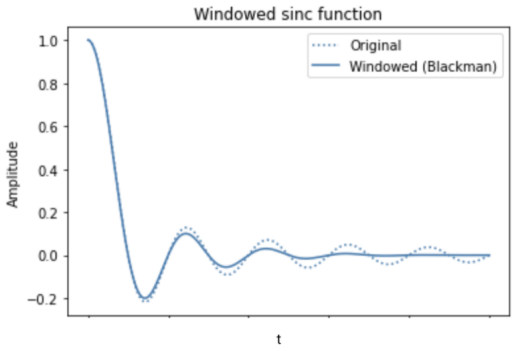

Figure 23: The windowed sinc function.

With that, you now have a finite impulse response ready to act as a filter kernel!

Optimum design methods

While the windowing method is straightforward, there are other ways to design FIR filter kernels. The main alternative to the windowing method is known as **optimum design**.

We're not referring to the method as "optimal" because it is necessarily better than other methods. Instead, the name comes from the mathematical problem of optimization.

So far, you've seen that designing FIR filters with the windowing method involves some trial and error. You must:

- Design your filter response.
- Find the impulse response.
- Try out some windows.
- Examine the results.

What if you wanted to specify a parameter like, say, the maximum amount of ripple in the passband and not go through these computations over and over? The answer, you may have guessed, is with optimum design.

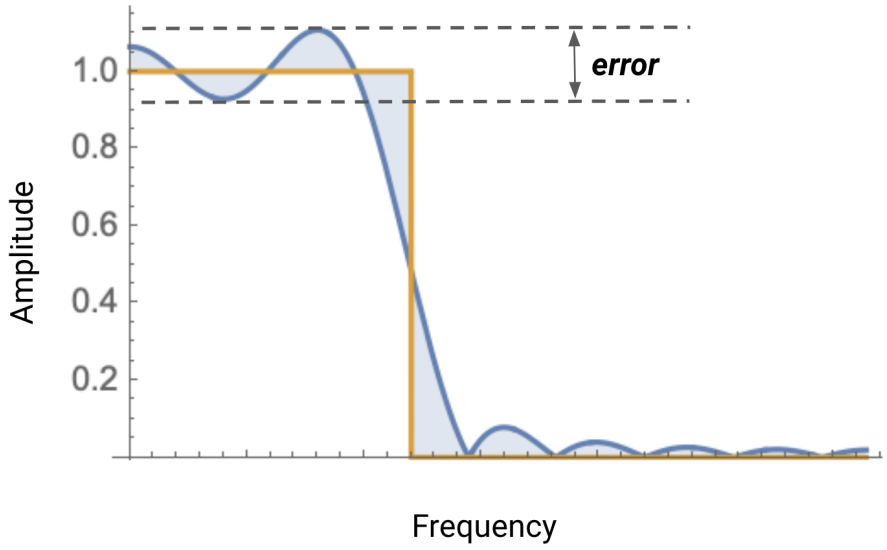

Figure 24: The optimum design method is used to minimize error like the amount of ripple in the passband.

The idea is that if you can describe the error of your resulting frequency response (which is shown in blue in Figure 24) compared to the ideal case (which is shown in orange), you can minimize that function for some amount of error you are willing to accept and use the resulting parameters.

That amount of error is sometimes referred to as the tolerance because it represents how much you will tolerate the result deviating from the ideal case.

A deeper discussion on the topic of optimum design is out of the scope of this tutorial, but if you are interested in going further, check out the freely-available optimal FIR filters section of *Spectral Audio Signal Processing* by Julius O. Smith III or Chapter 7 of the classic textbook *Digital Signal Processing* by Alan V. Oppenheim and Ronald W. Schafer.

FIR resources in Max

To work with FIR filters in Max, there are a number of objects and examples at your disposal.

The `buffir~` object takes a reference to a `buffer~` containing an impulse response and uses it as the filter kernel. For help, see:

- `buffir~ .maxhelp`
- `buffir-eq.maxpat`

Patching in `gen~`, which gives you single-sample control, is a natural fit for designing impulse responses. For an example, check out:

- `gen~.buffir.maxpat`

The HISSTools Impulse Response Toolbox package contains externals and abstractions that are helpful for building FIR filters.

- `bufconvolve~` for offline (non-real time) convolution and deconvolution of buffers
- `iruser~` to create IRs from a specified frequency response and output phase

Part Three: IIR Filter Concepts

We will now lay the conceptual groundwork required for designing and analyzing IIR filters, the counterpart of FIR filters. This section will cover:

- Filter kernels vs. filter coefficients
- Difference equations
- Transfer functions
- Filter order
- Poles, zeros, and stability analysis

IIR vs. FIR

Recall from part one that FIR and IIR filters are quite similar in implementation — they are simply weighted average machines. The difference is that FIR filters require only feedforward loops (relying only on input samples) whereas IIR filters use both feedforward and feedback loops (relying both on input and output samples).

Why might you use an IIR implementation over an FIR one?

Advantages

- Lower latency, fewer coefficients needed for the same specifications as an FIR filter
- Easier to model analog filters
- Easier to tweak parameters on the fly

Disadvantages

- Less numerically stable (feedback loops can cause unbounded growth)
- Nonlinear phase response

Despite their disadvantages, IIR filters are a common choice because they are less memory-intensive and many applications do not require linear phase responses.

Kernels, coefficients, and the difference equation

Remember that filters are just weighted average machines, and the work of filter design is to define the weights used in that machine. In FIR filters, we referred to these weights as the filter kernel, but they can also be called filter coefficients. Since the term kernel is associated with convolution, IIR filters — which don't use convolution — stick to calling them coefficients.

The word "coefficient" makes sense when you think about the relationship they have with the generalized equation for the output y of an FIR filter:

$$y[t] = \sum_{n=0}^{N-1} h[n]x[t-n]$$

Notice that once you expand out the summation, the kernel values $h[n]$ act as coefficients for each of the terms from input signal x. For example, a kernel with three values expands to:

$$y[t_n] = h[0]x[t_n] + h[1]x[t_n - 1] + h[2]x[t_n - 2]$$

Recall that another word for the generalized formula for computing outputs of a filter is a difference equation. You know what difference equations look like for FIR filters, but what about IIR filters?

Let's try an example, illustrated in Figure 25.

The summing and multiplying to find $y[t_n]$ is:

$$1 \cdot 1 + 4 \cdot 0 + 2 \cdot 2 + 2 \cdot 4 + 3 \cdot 6 = 31$$

With IIR filters, we distinguish between feedback and feedforward coefficients. By convention, feedback coefficients, given the letter a, are written with negative signs in front. Feedforward coefficients are given the letter b. The subscript of the coefficient is associated with the delay (in units of samples).

$$b_0 = 1, b_1 = 0, b_2 = 2, a_1 = -4, a_2 = -3$$

These coefficient naming conventions are not always followed. You may see cases where feedback coefficients are b and feedforward coefficients are a. It is important to check the context of the equation you are looking at.

Replacing values with variables gives:

$$y[t_n] = b_0x[t_n] + b_1x[t_n - 1] + b_2x[t_n - 2] - a_1y[t_n - 1] - a_2y[t_n - 1]$$

Gathering up terms into sums then yields:

$$y[t] = \sum_{i=0}^{2} b_i x[t_n - i] - \sum_{j=1}^{2} a_j y[t_n - j]$$

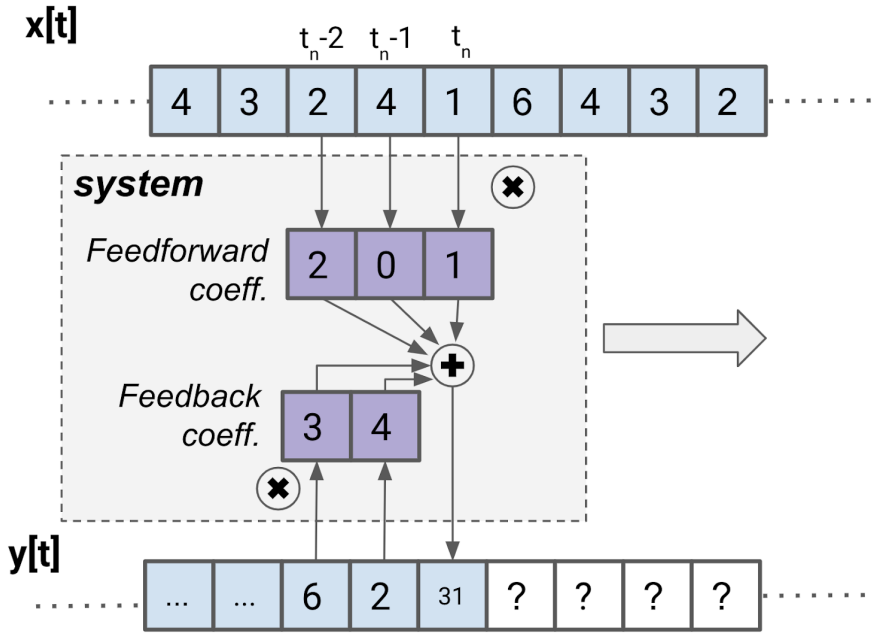

Figure 25: An example of an IIR filter computation step.

Finally, for M feedforward coefficients and N feedback coefficients, you can write more generally:

$$y[t] = \sum_{i=0}^{M} b_i x[t_n - i] - \sum_{j=1}^{N} a_j y[t_n - j]$$

Seeing the prominent subtraction of the two summations should now explain why the equation is called a "difference" equation!

Another new representation

So far we've discussed a number of interrelated ways to represent a given filter. If you know any one of the following, you can compute any of the others, assuming the filter is LTI:

- Impulse response
- Frequency response
- Step response
- Filter kernels/filter coefficients
- Difference equation

Besides the representations you've learned so far, there is yet another one called the **transfer function**. The transfer function is

the key to analyzing filter properties you may have heard of before, including:

- Filter order
- Poles and zeros of a filter
- Filter stability

Why wait until now to look at transfer functions? Well, since FIR filters don't run into the same stability problems as IIR filters, transfer functions are much more interesting to analyze for IIR filters. Furthermore, poles and zeros are used in IIR design and not so much in FIR design.

Before discussing these filter properties, let's first look at how you can find the transfer function from the difference equation using a very close relative of the Fourier transform — the **Z-transform**.

The Z-transform

Remember that the Fourier transform is a way to "translate" between a frequency and time representation of a signal. To do this, the Fourier transform uses sinusoids as a building block by mixing together sine waves of different frequencies and amplitudes to represent any other signal. In the world of math, this "mixing" is called a linear combination and sine functions are considered the basis. What if, instead of sine waves, you could use another kind of function as a basis?

The idea of the Z-transform is to use a complex exponential function, represented by the letter "z".

$$z = Ae^{i\phi} = A \cdot (\cos\phi + i \sin\phi)$$

In this equation, the A represents some amplitude, i is the imaginary number, and ϕ is an angle.

Complex exponentials may be intimidating at first, but they're incredibly useful for two practical reasons:

- It's often easier to do operations on exponential functions compared to trigonometric ones.
- Complex numbers wrap up two pieces of information into one — the amplitude and the phase. (You could think of complex numbers as being two-dimensional versus an integer which is one-dimensional.)

Note for completeness' sake: There are two similar forms of the Z-transform: the bilateral transform and the unilateral transform. The form above is the bilateral one, but the differences in the two forms won't make a difference for the scope of our analysis.

The Z-transform is defined as follows:

$$\mathcal{Z}\{x[n]\} = \sum_{n=-\infty}^{\infty} x[n]z^{-n}$$

The left hand side of the equation is saying that the Z-transform operates on a signal x that is indexed by integers n.

On the right hand side, the transform is defined as a sum over values of n from negative to positive infinity. This just means that every value of x is considered; if you had 12 samples in x, for example, you'd have a sum of 12 terms. Each of the terms in the sum is a value of input x (at index n) multiplied by the complex exponential z raised to the negative n power.

From this you can see what makes the Z-transform so similar to the Fourier transform — it represents an input signal as a sum of many sinusoids (only this time, those sinusoids are wrapped up in a complex exponential function). In fact, if you set $z = e^{i\phi}$ in the Z-transform, you would get back the definition of the complex discrete Fourier transform. The difference between the two is just that pesky amplitude A in the Z-transform!

This means the Fourier transform domain is in the unit circle of the z-plane.

What does the Z-transform of a simple example signal look like? Let's use a signal x that is two samples long. (Note the new shorthand for the Z-transform, too. A capital letter with an argument of z represents the Z-transform.)

$$\mathcal{Z}\{x[n]\} = X(z) = x[0]z^0 + x[1]z^{-1}$$

You can drop away z^0 since that just equals 1, and then you'll find:

$$X(z) = x[0] + x[1]z^{-1}$$

Now you see why block diagrams show a single sample delay with z^{-1} notation! This is related to what is known as either the **shift-invariance** or **time-delay** property of Z-transforms, which you can learn more about (along with other properties of the Z-transform) here.

If you ever want to avoid doing the work of performing a Z-transform by hand, you can try using Wolfram Alpha's Z-transform calculator.

Finding the transfer function

The basic form of a transfer function H(z) is a ratio between the Z-transform of an output signal versus an input signal.

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

To see how you might find a real-world transfer function given a difference equation, let's walk through an example of an IIR filter's difference equation.

$$y[n] = 2y[n-1] + \frac{1}{3}x[n] + x[n-1]$$

First, gather the output signal terms on the left and the input signal terms on the right.

$$y[n] - 2y[n-1] = \frac{1}{3}x[n] + x[n-1]$$

Next, write each term as its Z-transform.

$$Y(z) - 2Y(z)z^{-1} = \frac{1}{3}X(z) + X(z)z^{-1}$$

Do a little rearranging, taking advantage of the linearity property of the Z-transform so that X(z) and Y(z) can be factored out.

$$(1 - 2z^{-1})Y(z) = \left(\frac{1}{3} + z^{-1}\right) X(z)$$

One more step, and you're there...

$$H(z) = \frac{Y(z)}{X(z)} = \frac{\frac{1}{3} + z^{-1}}{(1 - 2z^{-1})}$$

Congratulations! You've found your very first transfer function.

Filter order

Transfer functions are often represented in a factored polynomial form. In other words, the "shape" of a factored transfer function will look something like:

$$H(z) = g\frac{(1 - q_1 z^{-1})(1 - q_2 z^{-1}) \cdots (1 - q_{M_1} z^{-1})}{(1 - p_1 z^{-1})(1 - p_2 z^{-1}) \cdots (1 - p_{M_2} z^{-1})}$$

There will be some constant gain factor in front and the numerator and denominator will be a bunch of terms multiplied together where each term is 1 minus some factor times z^{-1}.

Your first transfer function, for instance, slightly rearranged to be in factored form is:

$$H(z) = \frac{1}{3}\frac{\left(1 + 3z^{-1}\right)}{(1 - 2z^{-1})}$$

In this case, the numerator and denominator only have one term, but there is no limit. In fact, the maximum of the number of factored terms in the numerator and denominator is the order of the transfer function. So if there were three terms in the numerator

and four terms in the denominator, the transfer function would be "fourth-order." This is where the value x comes from when a filter is described as an x-order filter.

In general, a higher order means a better quality filter (with a steeper slope). At the same time, a higher order means more computational effort is needed. Can you also see why longer delay lines are needed for higher order? (Hint: Think about what the difference equation would look like as more z^{-1} terms are added. How many previous samples will the calculation need?)

With the `filterdesign`, `filterdetail`, and `plot~` objects in Max, you can play around to see the effect of filter order on the corresponding frequency response.

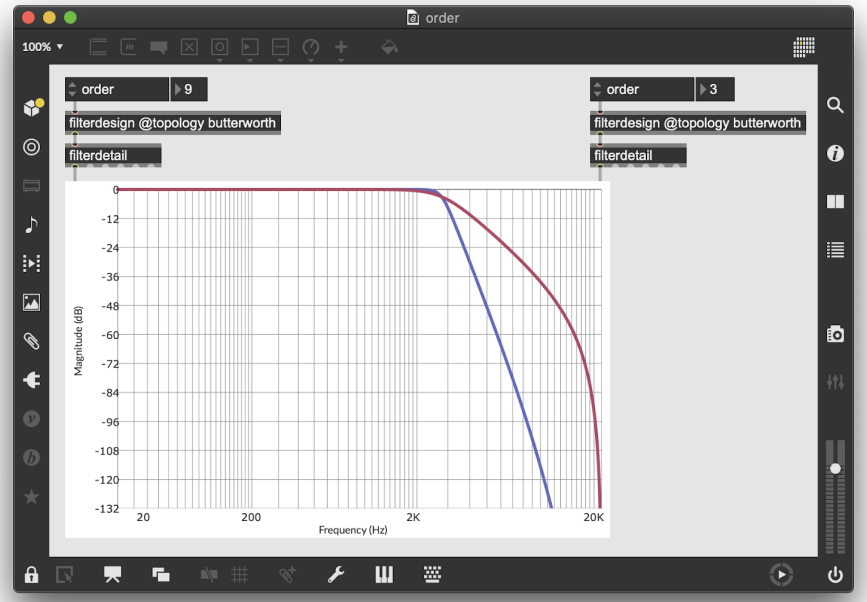

Figure 26: Using `plot~`.

Poles, zeros, and stability

Let's make up a new transfer function that is higher-order and a bit more exciting to analyze.

$$H(z) = \frac{(1 - 0.4z^{-1})(1 - 0.3z^{-1})(1 + 0.15z^{-1})(1 - 0.35z^{-1})}{(1 - 0.75z^{-1})(1 - 0.25z^{-1})}$$

First, what if one of the terms in the top was zero? For instance, what if z was 0.4, 0.3, -0.15, or 0.35? In other words, what if z was a root of the numerator polynomial? The term in parentheses would

become $1 - 1 = 0$, making the entire transfer function equal zero for that point. These values would be the so-called **zeros** of the transfer function.

Similarly, what if one of the terms on the bottom was zero? If z was one of the roots, 0.75 or 0.25, the bottom would equal zero, making $H(z)$ undefined. These are the **poles** of the transfer function.

These poles and zeros are often illustrated on a **pole-zero** plot. Remember that z is an imaginary number, so plotting it occurs on the **z-plane** (the complex plane) where the horizontal axis is the real part and the vertical axis is the imaginary part. Note that in our example, all of the poles and zeros have imaginary parts that equal zero, but this doesn't have to be the case (and it often won't be).

For an example of a pole-zero plot, the `zplane~` object in Max is a useful tool. You can manipulate the poles (shown as x's) and zeros (shown as o's), and it will output filter coefficients that `filtergraph` will understand and plot on a frequency response graph.

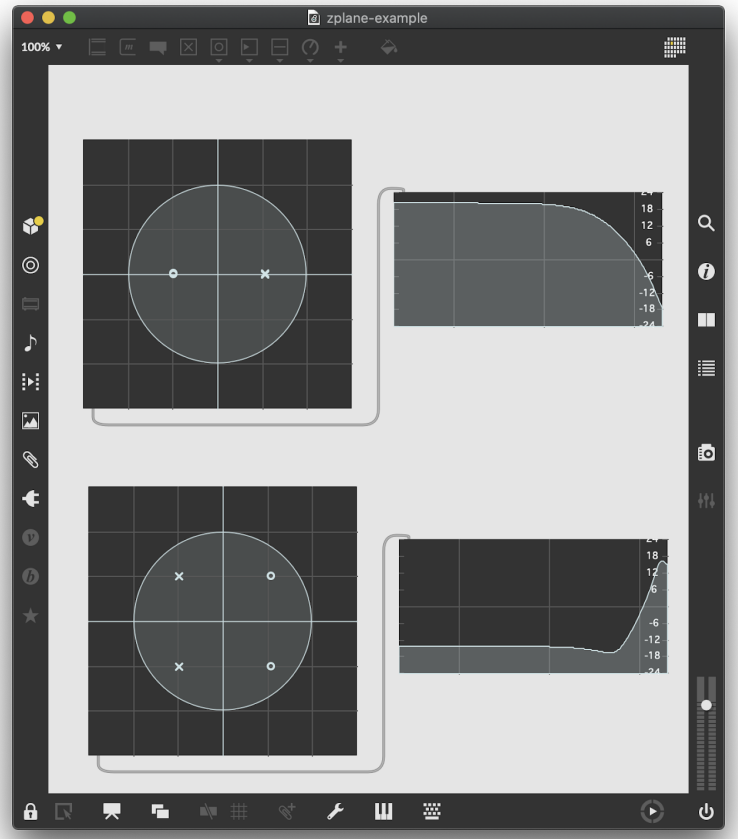

Figure 27: Using `zplane~`.

As you play with `zplane~`, you will notice that the poles and zeros

mirror across the horizontal axis. This is explained by the complex conjugate root theorem. The theorem basically says that for any root of a complex-valued function $a + bi$, there will be another root $a - bi$.

Finally, you'll notice that pole-zero plots usually show the unit circle. Why? In short, it's a way to indicate where a filter will be stable. If any poles or zeros lie outside of the unit circle, the filter will become unstable, causing any input to blow up to infinity at the output.

Conceptual map

As you reflect on the concepts introduced so far in this tutorial, the illustration in Figure 28 may be a helpful visualization of the relationships between them.

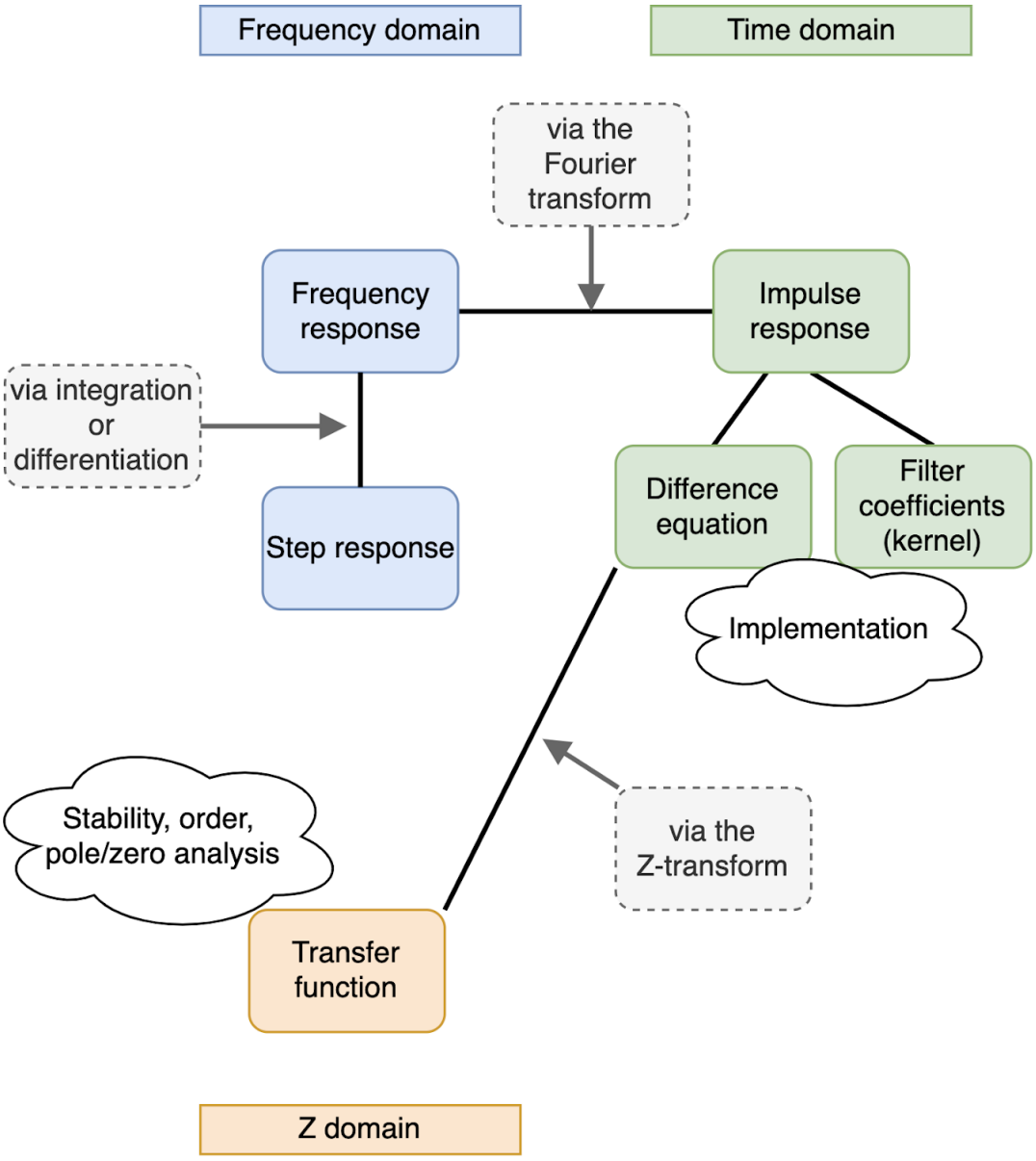

Figure 28: A concept map.

Part Four: IIR Filter Implementation

In this part, we'll take our first steps into the implementation of IIR filter designs. While the field of recursive digital filters is enormous, understanding the basic landscape will help you to navigate and take your first steps into exploring these new territories.

This section will cover:

- Analog filter prototypes
- Methods for digitizing those prototypes
- Elementary filter sections
- Biquads and cascades

When we design a filter, we're usually referring to optimizing for a set of "design" parameters such as the steepness of the transition band or the flatness of a stopband or passband. We do this by describing the filter mathematically and then solving a set of equations to get the corresponding filter coefficients.

Since analog filter design has been studied for decades, many of the most useful design equations have already been solved. These solutions are known as **filter prototypes**. Wikipedia nicely summarizes their usefulness:

> The utility of a prototype filter comes from the property that all these other filters can be derived from it by applying a scaling factor to the components of the prototype. The filter design need thus only be carried out once in full, with other filters being obtained by simply applying a scaling factor.

There are a several useful techniques available to us that allow us to use these prototypes in the digital domain, including:

- Impulse invariant transformation
- Bilinear transformation
- Matched Z-transform

Before we discuss these digitizing techniques, though, let's look at some of the most common filter prototypes — Butterworth, Chebyshev, and elliptic.

Filter prototypes

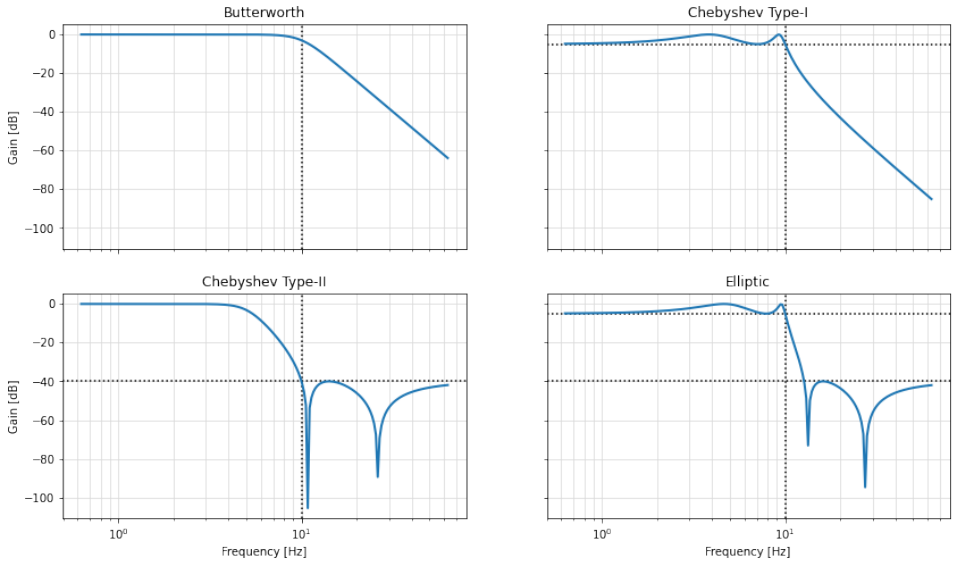

Figure 29: Four common filter prototype frequency responses. Keep in mind that the y-axis of these frequency response plots are in units of dB (on a logarithmic scale) so the large "bumps" are less audible than they appear.

Butterworth Filters

The Butterworth filter is named after Stephen Butterworth, an electrical engineer who first published the design in 1930. The goal of the Butterworth filter is to achieve a maximally-flat passband — there should be no distortion or "ripples." Out of the four prototypes here, Butterworth filters have the slowest rolloff for a given filter order.

Chebyshev Filters (Type-I and Type-II)

Chebyshev filters are named for the Chebyshev polynomials used to derive this filter design. Both Type-I and Type-II Chebyshev filters aim to minimize the error between an ideal and actual filter across the frequency response.

The rolloff of both types of filters are steeper than that of Butterworth filters — they do this by allowing ripple in the passband (for Type-I) or ripple in the stopband (for Type-II).

Type-I Chebyshev filters have slightly faster rolloffs than Type-II, but the ripple in the passband (which is likely low enough to be inaudible) may be less acceptable than ripple in the stopband.

Elliptic Filters

Elliptic filters (also known as Cauer or Zolotarev filters) have an even steeper rolloff than either of the Chebyshev filters. They achieve this

by allowing ripple in both the stopband and passband. The amount of ripple in either band can be adjusted independently from the other.

Digitization methods

Before we begin discussing digitization methods, let's start by reminding ourselves of the distinction between analog and digital systems and signals.

In an analog signal, there is no limit to how far you can "zoom in" to find a value at any point; for any time t along the signal, there's an exact value available. This is referred to as having a **continuous-time domain**.

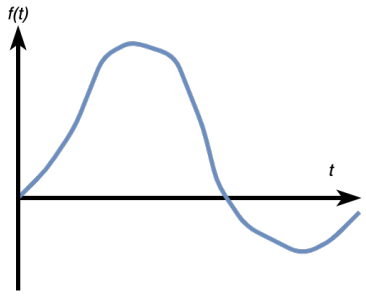

Figure 30: A continuous-time domain signal.

The time domain of a digital signal, on the other hand, is discrete. This means that a finite number of samples n along the signal are defined. If you want to know the value at a point that isn't defined, you have to use the surrounding samples and interpolate to make a reasonable guess about what that value is.

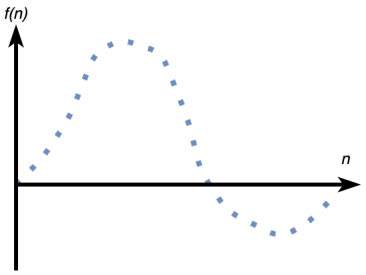

Figure 31: A discrete-time domain signal.

Since analog filter design equations are already available thanks to the work of electrical engineers, all we need to do is to be able to change the domain for these equations from the continuous analog representation to the discrete digital domain. If that sounds scary, just remember that you're actually already a pro at changing domains! Any time in the course of this tutorial that you've worked with a "transform" such as a Fourier transform or a Z-transform, you've been moving between domains. All that those transformations do is change the dependent variable of an equation by providing some kind of map between them.

For instance, you're probably already familiar with how to transform continuous-time signals to discrete-time ones. If you have a sine wave signal $f(t)$, in the continuous-time domain (meaning t can be any value), you would write:

$$f(t) = \sin t$$

If you wanted that same signal to be transformed to the discrete-time domain (where samples are defined for integer values n), you'll need to have some kind of relationship between t and n. The easiest way to do that is to define a constant T as the distance between samples (the sampling interval), and get the following relationship between t and n:

$$t = nT$$

By substituting that relationship into the continuous-time signal representation, you would get back out a discrete-time representation

that only depends on integer values n.

$$f(n) = \sin nT$$

Okay, so if the relationship between continuous- and discrete-time signals is so simple, could it be just as easy to transform between continuous and discrete systems?

Here's the bad news first: notice that when you go from analog to digital signal representations, you're losing information. All of the data between samples in a digital signal are lost! The only way to find out the values between data points is to make some kind of educated guess as to what the value would be (usually using interpolation). The same goes for mapping the continuous-time equations governing analog systems to equations for digital ones. By making any kind of mapping, you will lose some fidelity with digitization.

The good news is that with digitizing analog filter prototypes, you can choose which kinds of fidelity you are willing to give up in exchange for improvements in other areas. Let's explore the characteristics of three different transformations from the analog to the digital domain.

Impulse invariant transformation

The impulse invariant transformation takes a continuous-time impulse response $h_c(t)$ from an analog filter, samples it in intervals of T, and uses that impulse $h[n]$ as the basis for the digital filter.

$$h[n] = Th_c(nT)$$

A characteristic of the impulse invariant transformation is an aliased frequency response due to the spectral folding explained by the sampling theorem.

Bilinear transformation

The bilinear transform is mapping between the analog and digital transfer functions.

In continuous-time systems, instead of z, transfer functions use the variable s. By using the following relationship between s and z, you can take an analog transfer function (in terms of s) and digitize it.

$$s = \frac{2}{T} \cdot \frac{1 - z^{-1}}{1 + z^{-1}}$$

A characteristic of the bilinear transform is maintenance of stability. Since the bilinear transform maps the s-plane to the z-plane,

stable poles of an analog filter will always maintain their stable position when mapped to a different domain. One downside of the bilinear transform is "frequency warping" — the frequency response of a digitized filter is slightly different from its analog counterpart.

$$H_d(z) = H_a\left(\frac{2}{T} \cdot \frac{1 - z^{-1}}{1 + z^{-1}}\right)$$

Matched Z-transform

The matched Z-transform (also known as the root matching method) works by taking all poles and zeros s of the analog filter and mapping them to the Z-plane with the following relationship:

$$z = e^{sT}$$

This method is simple and maintains stability of the original analog filter, but it warps both the time- and frequency-based characteristics of the digital filter result. Since that's the case, both the impulse invariant and bilinear transforms are usually the transforms of choice for filter design.

Elementary filter sections

As you build higher and higher order filters, you will notice that the order of magnitude of the coefficients start to grow further apart. For example, a sixth-order filter can have some coefficients with a magnitude of 10^{-10} and others with a magnitude of 10^{-1}. These differences in order of magnitude can lead to major floating point errors and numerical instability.

To minimize this kind of numerical error, higher order filters can be split up into smaller, lower order "elementary filter sections" (a term coined by Julius O. Smith III) like one-pole, one-zero, two-pole, and two-zero filters.

This "splitting" of higher order filters is possible due to two properties of transfer functions.

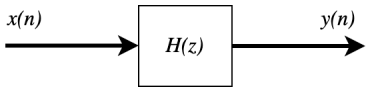

Figure 32: A generic higher-order system.

1. Transfer functions in series can be combined by multiplying

Imagine you are splitting up $H(z)$ into two separate systems where the first section outputs $v(n)$ as the input to the second section (as in Figure 33).

The transfer function of the "lumped" system is:

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

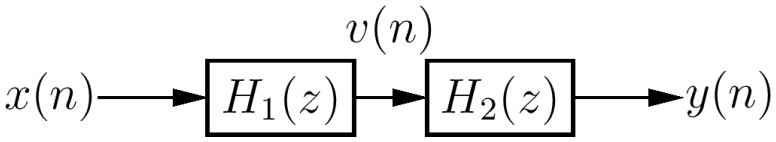

Figure 33: Series split system. Image source: *Introduction to Digital Filters with Audio Applications*.

You can also separately write the transfer functions for each section in the series version.

$$H_1(z) = \frac{V(z)}{X(z)}, H_2(z) = \frac{Y(z)}{V(z)}$$

Let's check that multiplying those transfer functions will give us the same result as the lumped system.

$$H(z) = H_1(z)H_2(z) = \frac{V(z)}{X(z)} \cdot \frac{Y(z)}{V(z)} = \frac{Y(z)}{X(z)}$$

Great! This property of transfer functions is true.

2. Transfer functions in parallel can be combined by adding

How about the other case of sections in parallel?

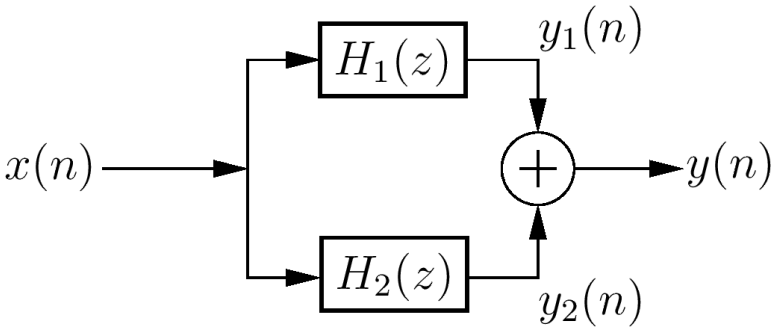

Figure 34: Parallel split system. Image source: *Introduction to Digital Filters with Audio Applications*.

Again, the lumped model is:

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

The individual sections are:

$$H_1(z) = \frac{Y_1(z)}{X(z)}, H_2(z) = \frac{Y_2(z)}{X(z)}$$

Let's check that summing H_1 and H_2 returns the expected result.

$$H(z) = H_1(z) + H_2(z) = \frac{Y_1(z)}{X(z)} + \frac{Y_2(z)}{X(z)} = \frac{Y_1(z) + Y_2(z)}{X(z)}$$

From the diagram, you can see that $y_1 + y_2 = y$, so you can now write:

$$\frac{Y_1(z) + Y_2(z)}{X(z)} = \frac{Y(z)}{X(z)}$$

Perfect — summing sections in parallel works as expected.

To summarize, if you can turn a high order transfer function into a product of lower order transfer functions, you can take advantage of these properties and implement the higher order filter as a set of lower order filters in series. A similar idea would work for creating a sum of lower order transfer functions and implementing them as a set of parallel filters.

Biquads and cascades — IIR in practice

The biquadratic ("biquad") filter is a two-pole, two-zero filter that is the workhorse building block of IIR filters. Since it is a second order filter, it provides numerical stability. Also, the digital implementation is flexible — you can easily make any of the other elementary filter sections (like a one-pole or one-zero filter) by simply using zero as a filter coefficient.

There are four different forms of the biquad filter:

- Direct form 1
- Direct form 2
- Transposed direct form 1
- Transposed direct form 2

The details of these different forms are out of the scope of this tutorial, but it is worthwhile to know they exist since you may see different variations of the biquad out in different block diagrams. Different forms have different strengths and weaknesses, and you can learn more about them in this section of *Introduction to Digital Filters with Audio Applications*.

In practice, when building higher-order IIR filters, series implementations are generally preferred. This technique is known as "cascading."

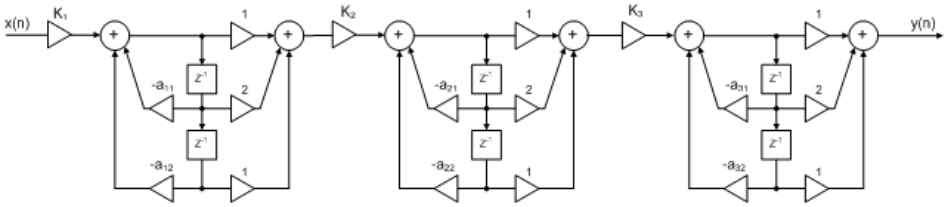

Figure 35: Cascaded biquads. Image source: "Design IIR Filters Using Cascaded Biquads"

Armed with an understanding of how IIR filters are designed, you can explore the `biquad~`, `cascade~`, `filterdesign`, and `filtergraph~` objects in Max with a better idea of what's occurring behind the scenes.

Wrapping up

Now that you've made it to the end of this whirlwind tour of digital filters, you should have new insight into what a digital filter is, how one is implemented, and what design decisions can be made.

Check out the references section for suggested further reading on digital filter design.

Resources

Alan V. Oppenheim. 6.007 signals and systems, 2011. URL https://ocw.mit.edu/resources/res-6-007-signals-and-systems-spring-2011.

Alan V. Oppenheim and Ronald W. Schafer. *Digital Signal Processing*. Prentice Hall, Inc., 1975.

Lawrence R. Rabiner and Bernard Gold. *Theory and Application of Digital Signal Processing*. Prentice Hall, Inc., 1975.

Derek Rowell. 2.161 signal processing: Continuous and discrete, 2008. URL https://ocw.mit.edu/courses/mechanical-engineering/2-161-signal-processing-continuous-and-discrete-fall-2008.

Julius O. Smith. Introduction to digital filters with audio applications, 2021a. URL https://ccrma.stanford.edu/~jos/filters/.

Julius O. Smith. Spectral audio signal processing, 2021b. URL https://ccrma.stanford.edu/~jos/sasp/.

Steven W. Smith. The scientist and engineers guide to digital signal ..., 2021c. URL https://dspguide.com/.