The Filter Wizard
issue 33: More Direct Waveform Synthesis: Mr Chebychev Helps Out
Kendall Castor-Perry

Last time [add link] we looked at additive synthesis, one harmonic at a time, as a way of creating the key waveforms in a music synthesizer, which we could them process subtractively by passing them through a filter. Then we wondered whether we could eliminate the (usually time-varying) filter by making the additive synthesis time-varying in a way that emulates the filter behaviour. Let's dig further now into what we might be able to do to actually create these harmonic sums using digital hardware that's normally intended to run the filters we're trying to replace!

We'll start with a digitally generated sinewave, because that can be done cleanly for arbitrary frequency resolution. We need that for musical applications. One approach is to pass the initial sinewave through a precalculated non-linearity map in order to generate the harmonic-rich waveform. This has a superficial simplicity about it, but has many pitfalls. You need to be sure that your non-linearity is not going to create unacceptably high frequencies that will alias round. So you might need several different carefully calculated maps, depending on which register you find yourself in. Also, you have to store those maps to quite high resolution, since you're feeding in a high-accuracy digital sinewave that can occupy a large number of states. As in the synthesis of the sinewave, relying on large high-resolution tables doesn't feel like a good PSoC way of doing this.

We could replace large lookup tables by calculation of that non-linearity in real time. To do that, we need to choose forms of non-linear function that fit the hardware we're using, and that calculate quickly. We also need to determine analytically what these functions should do, so that we can derive the functions needed by inspecting the desired result.

So, I know at any given point in time what levels of each of the N harmonics of the fundamental I want to mix together. According to Russ, it seems to be common practice to limit N to 64. We need make sure that any gain coefficients for harmonics of too high a frequency are set to zero by the control processor; this gets round the aliasing issue highlighted last time. Now all we need are those actual harmonics. Let's assume that our nice digital sinewave generator is giving us samples sin(x) of the desired fundamental f (assuming a unit amplitude at the moment, for simplicity). In other words, or rather, in other symbols:

$$V_{out} = a_0 + a_1 \cdot \sin(2\pi ft) + a_2 \cdot \sin(2 \cdot 2\pi ft) + a_3 \cdot \sin(3 \cdot 2\pi ft) + \ldots$$

i.e.

$$V_{out} = \sum_{k=0}^{n} a_k \cdot \sin(k \cdot 2\pi ft) \qquad [1]$$

with an input of

$$V_{in} = \sin(2\pi ft)$$

(in the text, we write x for $2\pi ft$)

One approach that initially seemed attractive was to work out the necessary **polynomial** function of sin(x) that, when multiplied out, contains the right level of each of the harmonics. It's easy to show that sin^n(x) contains all harmonics of sin(x) up to sin(nx), and none higher. This method is easy to code and quick to execute. The DFB is good at doing a Horner's Method expansion of a polynomial, which is the recommended approach for precision and speed. In other words, we attempt to find a set of 'b' coefficients as a function of the 'a' values in [1] such that

$$V_{out} = b_0 + b_1 \cdot (V_{in}) + b_2 \cdot (V_{in})^2 + b_3 \cdot (V_{in})^3 + \ldots$$

i.e.

$$V_{out} = b_0 + b_1 \cdot \sin(2\pi ft) + b_2 \cdot \sin^2(2\pi ft) + b_3 \cdot \sin^3(2\pi ft) + \ldots$$

or

$$V_{out} = \sum_{k=0}^{n} b_k \cdot \sin^k(2\pi ft) \qquad [2]$$

Does the set of coefficients b[] exist for all sets of requirements a[]? Sadly, **no**. For even values of n, you can only create **cos(nx)**, not sin(nx), from power terms of the form sin^n(x). It turns out that if we had been working with cosines instead of sines, then it **would** have been possible to develop coefficients for cos^n(x) to achieve any desired weighted sum of cos(nx) terms. But figure 1 shows why this is no good. It's the same sum as shown in the last episode's figure 1, except with the sines changed to cosines. The wave shape is utterly different, it's nothing like a sawtooth waveform any more. If you compare it with figure 1 in the previous episode, you'll see that the peak value is higher as well, and the waveform is noticeably asymmetrical. It doesn't actually have a DC content, so this won't change when you pass it through an AC coupling circuit.

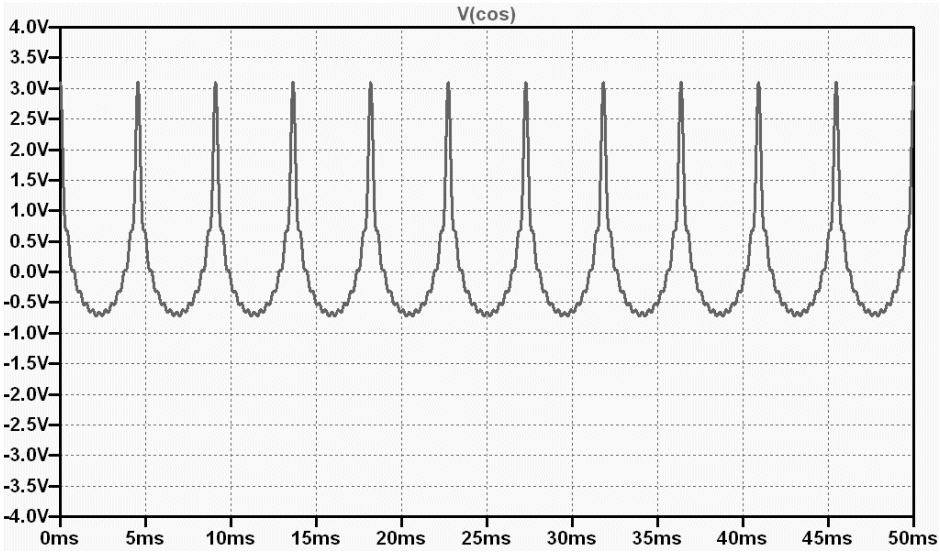

Figure 1: What we get when we change a sawtooth's sines into cosines.

“But the spectrum is just the same – surely it sounds identical”, I can just about here you cry above this rather strident buzzy tone. Well, it’s not hard to try that out, since our favourite simulator allows you to write .wav files that you can listen to. And, no, they don’t sound quite the same, to me at least. Most prominent is a small but still-noticeable difference in **pitch** – even though FFT analysis of the waveforms in the files confirms that the frequencies of the fundamental and all the harmonics are pixel-perfectly aligned. This is consistent with extensive research into the audible effects of harmonic phase, and a salutary reminder that pitch, in the musical sense, is not identical to frequency.

This is unsatisfactory when we’re trying to get as close as we reasonably can to the sonic behaviour of an old-school analogue synth. So, despite the efficient, FIR-like structure of polynomial evaluation on the DFB, I decided that it wasn’t the way to go.

Fortunately, you don’t need to look outside the canon of the great mathematicians whose work has driven the development of filters. Recurrence relations [3] and [4] are readily found that enable the calculation of harmonics of either sin(x) or cos(x) from lower-order harmonics that you already calculated using the same formula. These are generally attributed to the great Russian mathematician Chebychev, whose work crops up so often in filter design and in polynomial approximation theory.

$$\sin(nx) = 2 \cdot \cos(x) \cdot \sin((n-1)x) - \sin((n-2)x) \qquad [3]$$

and

$$\cos(nx) = 2 \cdot \cos(x) \cdot \cos((n-1)x) - \cos((n-2)x) \qquad [4]$$

There’s a catch, though – in order to calculate sin(nx), you need to have access to **both** sin(x) and cos(x). We need a bit of the old quadrature!

Remember that digital sinewave oscillator? This was built around an oscillating state variable filter very similar to the one reported in Chamberlin’s book. And we are in luck, my friend, because this form of oscillator inherently provides quadrature outputs! The key to the solution of our problem was built into the proposed architecture all along.

The creation of our harmonic-laden waveform is now a three-step process. First, you have to run the oscillator model to get instantaneous sample values of sin(x) and cos(x) for the sample point at hand. Then, a vector of values for all the desired sin(nx) terms is calculated iteratively right in the Digital Filter Block’s memory using [3]. I reckon this process will fit nicely on the DFB, including a bit of scaling for safety margin.

Lastly, equation [1] is calculated as a sum of products between this vector of sin(nx) values and another vector in memory that contains the desired fractional values of all the harmonic components at this point in time. This is almost the same form of sum you do for an FIR filter, and is very efficient on the DFB because of its dedicated address generation hardware. That vector can be dynamically manipulated by the CPU; at any kind of normal sample rate it would be possible to change every single value, by executing a DMA transfer into DFB memory in between sample calculations. In practice,

coefficients can probably be updated rather more slowly than once every audio sample period. This is something to experiment with.

Now, the eagle-eyed among you may have spotted a little inconsistency here. I've been waxing lyrical about how important it is to emulate the effect of a correctly-generated waveform passing through a dynamically-tuned old-school continuous time filter. But in building my harmonic sums, I just multiplied all the sin(nx) harmonic levels by real scaling factors. This means that I did not take the phase of the filter into account – or, rather, I made a zero-phase filter. And, for all we know, the phase shift through that filter could be as critical to the phat sound of the synth as the amplitude response is! Standard analogue filters have a decidedly non-linear phase response, especially at the high Q values that give the 'best' sounds.

Don't panic! There's a final architectural augmentation (sounds so much more impressive than 'tweak', doesn't it?) we can do to get round this. As well as generating all the sin(nx) terms using [3], we generate all the cos(nx) terms as well, using [4]. This should still just fit in the internal memory of the DFB. Then, we express the desired amplitude response of the filter to be emulated in real+imaginary format, and use these parts to separately multiply the sin(nx) and cos(nx) terms, before finally adding the whole darned lot together. Then, we get what we came for – a waveform with all the harmonics present in the analogue synth, with both amplitudes and phases varying with time just as if we'd gone to the bother of implementing a filter to process a static waveform.

Will it be worth it? Well, I think so. We had to do most of the work anyway just to accurately synthesize the required time-invariant waveform cleanly in the sampled domain. It's only a matter of programming and control to make those coefficients change over time, according to the same scheme you would have used in your MiniMoog or other favourite analogue synth. On top of that, there are no filter-related noise, instability or coefficient precision issues to deal with, so this should actually perform much better than a regular digital filter, especially one that's being dragged all over the z-plane by the dynamic programming required.

This method should emulate any filter and envelope generation combination that you can find objective details of. It will also implement filters that could never have been built into analogue synthesizers, such as time-reversed filters, that produce exactly the right waveform and timbre when you play the sound backwards! By cyclically varying the real and imaginary mix of harmonics, you can realize frequency shifts, allowing higher harmonics to be flattened (physically realistic) or sharpened (unusual). The possibilities seem endless.

Some time, I hope to be able to compare the performance and sonics of this synthesis method with the behaviour of some old-school analogue synth circuits that I also want to try out on the analogue elements in PSoC 3. All I need is time! I have high hopes for this filtering-without-filtering method. Maybe I'll even post some sound files. Meanwhile, don't vacillate – oscillate with PSoC! best / Kendall