Signal Processing Libraries for Faust

Julius SMITH
Center for Computer Research in Music and Acoustics (CCRMA)
Music Dept., Stanford University
Stanford, CA 94306,
USA
jos@ccrma.stanford.edu

Abstract

Signal-processing tools written in the FAUST language are described. Developments in FAUST libraries, `oscillator.lib`, `filter.lib`, and `effect.lib` since LAC-2008 are summarized. A good collection of sinusoidal oscillators is included, as well as a large variety of digital filter structures, including means for specifying digital filters using analog coefficients (on the other side of a bilinear transform). Facilities for filter-bank design are described, including optional delay equalization for phase alignment in the filter-bank sum.

Keywords

FAUST, audio signal processing, filters, effects, oscillators

1 Introduction

The FAUST (Functional AUdio STream) language, developed at GRAME[1] [1; 2], is well known for its compact specification of signal-processing block diagrams, and its compilation into efficient C++ audio applications and plugins. Thanks to the use of architecture files that encapsulate platform-specific details, FAUST applications can be conveniently generated for a wide variety of host environments (Linux, Mac, Windows), and audio plugins can be generated for a wide variety of host applications such as Pd and SuperCollider, to name just two [3; 4; 5].

In the `architecture` subdirectory within the FAUST distribution, there are presently seven `.lib` files containing various utility functions. Possibly the most commonly used of these is `music.lib`, which also imports `math.lib`. Three other `.lib` files pertain more specifically to signal processing utilities:

- `oscillator.lib` — signal sources
- `filter.lib` — general-purpose digital filters
- `effect.lib` — digital audio effects

The remaining two FAUST library files are `maxmsp.lib`—a Max/MSP compatibility library, and `reduce.lib`—enabling function application across a signal in time, such as `maxn(n) = reduce(max,n)` to compute the maximum amplitude of a signal.

The directory `examples/faust-stk/` additionally contains `instrument.lib`, providing common utility functions for the FAUST-STK collection [6], such as envelope generators and table-lookup utilities.

The libraries `oscillator|filter|effect.lib` were first discussed at LAC-08 [7]. This paper provides an overview of developments since then and up to FAUST release version 0.9.46 (Dec. 2011).

2 Faust Library `oscillator.lib`

The purpose of `oscillator.lib` is to provide reference implementations of various elementary waveform generators, such as sinusoidal, sawtooth, and pulse-train, as well as other classic signals such as pink-noise, etc.

2.1 Sinusoid Generators

All sinusoidal oscillators in `oscillator.lib` are invoked via the same API as `osc(freq)` (defined in `music.lib`), where `freq` is the desired oscillation frequency in Hz. However, some provide *two* outputs instead of one when both "in-phase" and "quadrature" (sine and cosine) are available. All are *filter-based*. That is, they are implemented as lossless second-order filters driven by an *impulse* signal `[1,0,0,...]`, and they use no wave

[1] http://faust.grame.fr/

tables.[2] All algorithms have been previously published [8; 9; 10; 11].[3]

Presently, the following algorithms are implemented:

`oscb`	"biquad" two-pole filter section (impulse response)
`oscr`	2D vector rotation (second-order normalized ladder) provides sine and cosine outputs
`oscrs`	sine output of `oscr`
`oscrc`	cosine output of `oscr`
`oscs`	state variable osc., cosine output (modified coupled form resonator)
`oscw`	digital waveguide oscillator
`oscws`	sine output of `oscw`
`oscwc`	cosine output of `oscw`

The relative merits of each oscillator type are summarized below. Note that all differences have to do with finite numerical precision effects and dynamic range variations under time-varying conditions. The best overall choice depends on the situation.

- `oscb`, the impulsed direct-form biquad[4] is the fastest computationally, requiring only one multiplication and two additions per sample of output. However, as is well known, the amplitude of oscillation varies strongly with frequency, and it becomes numerically poor toward `freq=0` ("dc").
- `oscr`, the "2D vector rotation," requires four multiplies and two additions per sample. Its amplitude is invariant with respect to frequency, and it is good all the way down to dc. Since its coefficients are numerically inexact roundings of `s = sin(2*PI*freq/SR)` and `c = cos(2*PI*freq/SR)`, where `SR` denotes the sampling rate (defined in `music.lib`), there is long-term amplitude drift corresponding to the extent the identity $\mathtt{s}^2+\mathtt{c}^2 = 1$ is violated. This oscillator provides in-phase (cosine) and phase-quadrature (sine) outputs.
- `oscs`, based on the classic "state variable filter," [12, p. 530] and known as the "magic circle algorithm" in computer graphics, is quite fast, requiring only two multiplies and two additions per output sample. Its amplitude varies much less with frequency, and it too is good down to dc. There is no amplitude drift over time, so this one can be used for very long signal durations. On the other hand, there is some dependence of oscillation amplitude on frequency. At low frequencies, its two state variables are nearly in phase quadrature, but they become in-phase at SR/2. Thus, two outputs with approximately 90-degrees relative phase at low frequencies could be brought out. The output that is brought out is the "cosine" choice.
- `oscw`, the second-order digital waveguide oscillator, requires one multiply and three additions when frequency is constant, and another multiply when frequency is changing. Otherwise it has all of the good properties of `oscr` (except for internal dynamic range normalization), providing sine and cosine outputs in exact phase quadrature, and no dependence of amplitude on frequency. However, unlike `oscr`, `oscw` exhibits no amplitude drift while frequency is fixed. This is because it uses a "structurally lossless" algorithm derived by transformer coupling of normalized digital waveguides [10; 8].[5] A negative point relative to `oscr` is that numerical difficulties may arise below 10 Hz or so, implying that `oscw` is not a good choice for LFOs. Internally, the state variables of `oscw` require a larger dynamic range than those of `oscr`. It is likely that `oscw` would be the most economical choice for special-purpose VLSI.

2.2 Virtual Analog Waveforms

The following waveform generators are presently included, among others:

`imptrain(freq)`	periodic impulse train
`squarewave(freq)`	zero-mean square wave
`sawtooth(freq)`	alias-suppressed sawtooth
`sawN(freq)`	order N anti-aliased saw

[2] `osc(freq)` in `music.lib` uses a length 2^{16} wave table. The linearly interpolated variant `osci(freq)` adds linear interpolation.

[3] `https: //ccrma.stanford.edu/~jos/pasp/-Digital_Waveguide_Oscillator.html`

[4] `https://ccrma.stanford.edu/~jos/filters/-Direct_Form_II.html`

[5] `https: //ccrma.stanford.edu/~jos/pasp/-Digital_Waveguide_Oscillator.html`

The `sawtooth` and `sawN` algorithms are based on recently developed "Differentiated Polynomial Waveform" (DPW) methods for virtual analog waveform generation [13; 14]. The default case is `sawtooth = saw2`, where `saw2` is a differentiated parabolic waveform (order 2). More generally, `sawN` is based on a differentiated polynomial of order N. The higher the order, the less aliasing is incurred. Bandlimited square, triangle, and pulse-train are derived as linear filterings of bandlimited sawtooth in FAUST releases beyond 0.9.46.

2.3 Noise Generation

The basic white-noise generator, uniformly distributed between -1 and 1, is `noise`, defined in `music.lib`. Based on that, `oscillator.lib` also defines `pink_noise`, also called "$1/f$ noise" [15], implemented (approximately) as white noise through a three-pole, three-zero IIR filter that approximates a $1/f$ power response.[6] The third-order IIR filter was designed using `invfreqz` in Octave (matlab).

3 Faust Library `filter.lib`

Filter-related utilities are provided in `filter.lib`. The principal functions defined appear in Fig. 1, p. 4, and Fig. 2, p. 5. To save space, functions introduced at LAC-08 [7] are not repeated here (such as EKS string synthesizer elements, comb filters, cubic distortion overdrive, dc blockers, speaker bandpass, Crybaby wah pedal, etc.). The subsections below provide further discussion of various groups. The source is documented with comments and references so that anyone knowledgeable in basic digital filter theory [16] should be able to use it as a (terse) manual and starting point for further reading.

3.1 Direct-Form Digital Filters

The four direct-form digital filter structures have coefficients that appear in the filter transfer function.[7] The functions `tf1(b0,b1,a1)` and `tf2(b0,b1,b2,a1,a2)` specify first- and second-order (biquad) digital filter sections, respectively. Often larger filters are built by stringing first- and second-order sections in series and/or parallel. The FAUST language makes this especially easy to do.

The function `iir(bcoeffs,acoeffs)` allows specification of an arbitrary-order IIR digital filter in direct form. The arguments `bcoeffs` and `acoeffs` are each parallel signal banks that provide the coefficients, and they may be thought of as "lists" of coefficients. The *pattern matching* facility in FAUST allows recursive definition in terms of such lists. As an example, `tf2(b0,b1,b2,a1,a2)` can be alternatively specified as `iir((b0,b1,b2),(a1,a2))`. As usual in FAUST, the specification is compact:

```
iir(bv,av) = sub ~ fir(av) : fir(bv);
```

where `fir(bv)` specifies a general causal FIR digital filter, with `bv` the list ("vector") of FIR "tap" coefficients. It is given by

```
fir(bv,x)
  = sum(i,count(bv),take(i+1,bv) * x@i);
```

where `count` and `take` are defined in `math.lib`.

3.2 Ladder and Lattice Digital Filters

Ladder and lattice digital filters have superior numerical properties. Using the pattern-matching facility, it was possible to specify all four major types recursively in FAUST. A particularly valuable case is the *normalized ladder filter* [17] `iir_nl(bcoeffs,acoeffs)`, used as the basis for the super-robust biquad `tf2snp()`. While normalized biquads are straightforward to design (*e.g.*, `nlf2()` in `filter.lib`), the normalized ladder filter realizes rational transfer functions of any order (any number of poles and zeros) in terms of a power-normalized ladder structure. For an introduction and pointers to references, see [8] and `filter.lib`.[8]

3.3 Digital Filter Sections Specified as Analog Filter Sections

It is convenient to be able to specify basic filter section in terms of *analog* filter coefficients, as opposed to the usual digital-filter coefficients. This is easy to do in FAUST by including a built-in *bilinear transform*.[9] This makes use of the wonderful

[6] `https: //ccrma.stanford.edu/~jos/sasp/-Example_Pink_Noise_Analysis.html`

[7] `https: //ccrma.stanford.edu/~jos/filters/-filters/Four_Direct_Forms.html`

[8] `https: //ccrma.stanford.edu/~jos/pasp/-Conventional_Ladder_Filters.html`

[9] `https: //ccrma.stanford.edu/~jos/pasp/-Bilinear_Transformation.html`

Direct-Form Digital Filters	(§3.1, p. 3)
`fir(bcoeffs)`	general FIR digital filter
`iir(bcoeffs,acoeffs)`	general IIR digital filter
`tf1(b0,b1,a1)`	first-order direct-form digital filter = `iir((b0,b1),(a1))`
`tf2(b0,b1,b2,a1,a2)`	`iir((b0,b1,b2),(a1,a2))`
Lattice/Ladder Filters	(§3.2, p. 3)
`iir_lat2(bcoeffs,acoeffs)`	two-multiply lattice digital filter
`iir_kl(bcoeffs,acoeffs)`	Kelly-Lochbaum ladder digital filter
`iir_lat1(bcoeffs,acoeffs)`	one-multiply lattice digital filter
`iir_nl(bcoeffs,acoeffs)`	normalized ladder digital filter
`tf2np(b0,b1,b2,a1,a2)`	biquad based on stabilized 2nd-order normalized ladder
`nlf2(f,r)`	second-order normalized ladder digital filter, special API
Analog-Specified Filters	(§3.3, p. 3)
`tf2s(b2,b1,b0,a1,a0,w1)`	`tf2` specified via *s*-plane (analog) coefficients
`tf2snp(b2,b1,b0,a1,a0,w1)`	tf2s using a protected normalized ladder filter for `tf2`
`tf2sb(b2,b1,b0,a1,a0,w1,wc)`	`tf2s` plus a mapping to *bandpass* in the digital domain
`tf1sb(b1,b0,a0,w1,wc)`	same as `tf2sb` but for first-order filter sections
IIR Low/High/Band-Pass	(§3.4, p. 6)
`lowpass(N,fc)`	Nth-order Butterworth lowpass, -3 dB frequency at `fc` Hz
`highpass(N,fc)`	Nth-order Butterworth highpass, -3 dB frequency at `fc` Hz
`bandpass(Nh,fl,fu)`	Order `2*Nh` Butterworth bandpass, -3 dB frequencies `fl,fu` Hz
`bandstop(Nh,fl,fu)`	Order `2*Nh` Butterworth bandstop filter, -3 dB gain at `fl,fu` Hz
`lowpass3e(fc)`	3rd-order elliptic lowpass, 60 dB stopband rejection, 0.2 dB passband rip.
`lowpass6e(fc)`	6th-order elliptic lowpass, 80 dB stopband rejection, 0.2 dB passband rip.
`highpass6e(fc)`	highpass transformation of `lowpass6e` ($\omega \leftarrow 1/\omega$)
`bandpass12e(fl,fu)`	bandpass transformation of `lowpass6e`
`bandpass6e(fl,fu)`	bandpass transformation of `lowpass3e`
Shelfs, Peaking Equalizers	See FAUST example `parametric_eq.dsp`
`low_shelf1(L0,fx,x)`	1st-order shelf, dc gain `L0` dB, crossover to unity gain at `fx` Hz
`low_shelf1_l(G0,fx,x)`	dc gain `G0` (linear), crossover to unity gain at `fx` Hz
`low_shelf3(L0,fx,x)`	3rd-order low shelf
`low_shelf5(L0,fx,x)`	5th-order low shelf
`low_shelf`	`= low_shelf3; // default = third-order case`
`high_*`	same high-shelf cases as for `low_shelf`
`peak_eq(Lfx,fx,B)`	2nd-order "peaking equalizer", peak level `Lfx` dB, width `B` Hz at `fx` Hz
`peak_eq_cq(Lfx,fx,Q)`	Constant-Q 2nd-order peaking equalizer section, Q = fx/B
`peak_eq_rm(Lfx,fx,w)`	Regalia-Mitra 2nd-order peaking equalizer section, `w ~ PI*B/SR`
Fractional Delay Lines	(§3.6, p. 6)
`fdelayN(maxdelay, delay)`	`N`th-order FIR Lagrange-interpolated delay line, `N`=1,2,3,4
`fdelayNa(maxdelay, delay)`	`N`th-order IIR allpass-interpolated delay line, `N`=1,2,3,4

Figure 1: Functions defined in `filter.lib` since LAC-08. See the source code for full usage documentation and literature references.

Filter Banks	(§3.7, p. 6)
`mth_octave_analyzer(O,M,ftop,N)`	`N`-band octave filter-bank, `M` band-slices per octave, Butterworth band-split order `O` (not `0`, must be an odd integer), `N` = total number of bands (including dc and Nyquist), `ftop` = highest band-split crossover frequency (e.g., 20 kHz)
`mth_octave_analyzer6e(M,ftop,N)`	uses order 6 elliptic band-split filters
`mth_octave_filterbank(O,M,ftop,N)`	`mth_octave_analyzer` followed by delay equalizer
`mth_octave_filterbank_alt`	dc-inverted variant (cheaper for odd `O`)
`mth_octave_spectral_level`	spectrum analyzer using `mth_octave_analyzer(5)`, displays (in bar graphs) the average signal level in each spectral band
`mth_octave_spectral_level6e`	order 6 elliptic crossovers
`spectral_level`	`= mth_octave_spectral_level(2,10000,20); // simplest`
`half_octave_analyzer(N)`	`= mth_octave_analyzer6e(2,10000,N);`
`half_octave_filterbank(N)`	`= mth_octave_filterbank5(2,10000,N);`
`octave_filterbank(N)`	`= mth_octave_filterbank5(1,10000,N);`
`octave_analyzer(N)`	`= mth_octave_analyzer6e(1,10000,N);`
`analyzer(O,lfreqs)`	general analyzer, order `O` Butterworth crossovers at listed freqs
`filterbank(O,lfreqs)`	`analyzer(O,lfreqs) :` delay equalizer (allpass-complementary)
`filterbanki(O,lfreqs)`	Inverted-dc variant

Figure 2: Filter-bank functions defined in `filter.lib`. See the source code for full usage documentation and literature references.

feature of FAUST that if the coefficients are constant, all expressions will compile away to leave numerical digital-filter coefficients. On the other hand, if a slider-control, say, is providing an analog coefficient, the bilinear transform will be computed in real time (at the control rate) from the controller by the compiled result. Normally a one-pole smoother such as `smooth(0.99)` is used to interface the final computed coefficient into the filtering computation at the full sampling rate.

In particular, `tf2s(b2a,b1a,b0a,a1a,a0a,w1)` equals `tf2(b0d,b1d,b2d,a1d,a2d)` specified in the analog domain, where a last-letter 'a' means 'analog', and 'd' means 'digital'. (Note the opposite numbering of the coefficients, in conformance with typical notation.) Thus, the analog transfer function specified is

$$H(s) = \frac{b_{2,a}\,s^2 + b_{1,a}\,s + b_{0,a}}{s^2 + a_{1,a}\,s + a_{0,a}}.$$

The parameter `w1` is the digital frequency ω_d to which analog frequency $\omega_a = 1$ is mapped; it determines the frequency-scaling parameter of the bilinear transform. In lowpass or highpass filter design, the frequency mapping is applied to the cutoff frequency (-3 dB point).

Butterworth filters are particularly easy to specify in analog form [18; 19; 16], [10] because, for order N, all N zeros are at infinity and all N poles lie along a circle in the left-half s-plane. For example, the second-order Butterworth lowpass filter with its -3 dB frequency normalized to $\omega_a = 1$ is simply

$$H(s) = \frac{1}{s^2 + \sqrt{2}\,s + 1}$$

and can be specified as `tf2s(0,0,1,sqrt(2),1)`.

3.3.1 Normalization and Stability Protection

For extreme time-varying filtering applications, a practically useful variant named `tf2snp` is provided that implements `tf2s` using a normalized ladder filter (for decoupling signal and coefficient energy, §3.2) together with stability projection (easy to do in ladder/lattice digital filters by simply clipping their reflection coefficients to the range $(-1, 1)$). This is used in the most numerically robust Moog VCF implementation `moog_vcf_2bn` (`effect.lib`, §4).

[10] `https://ccrma.stanford.edu/~jos/filters/-Butterworth_Lowpass_Design.html`

The example `vcf_wah_pedals.dsp` in the FAUST distribution provides a comparison of three Moog VCF implementations as well as the second-order Crybaby wah-pedal and a fourth-order wah-pedal based on the Moog VCF.

3.3.2 Bandpass Mapped Biquad

The function `tf2sb(b2,b1,b0,a1,a0,w1,wc)` is a bandpass mapping of the basic analog-specified biquad `tf2s`. In addition to the frequency-scaling parameter `w1` (which gets set to *half* the desired passband width in radians per second), there is a desired center-frequency parameter `wc` (also in rad/s). Thus, `tf2sb` implements a fourth-order *digital* bandpass filter section specified by the coefficients of a second-order *analog* lowpass prototype section. Such sections can be combined in series for higher orders. The order of mappings is (1) frequency scaling (to set lowpass cutoff `w1`), (2) bandpass mapping to `wc`, then (3) the bilinear transform, with the usual scale parameter `2*SR`, where `SR` denotes the sampling rate. The FAUST implementation for this was based on algebra carried out in `maxima`.

3.4 Butterworth Lo/Hi/Bandpass Filters

Butterworth lowpass and highpass filters of any order can be defined recursively in FAUST thanks again to the pattern-matching facility in the language. The elliptic (Cauer) filters[11] are special-cased because the pole locations are computed using the elliptic rational function, which is not available in typical computer math libraries. Such a function could of course be supplied as a foreign function in FAUST.

3.5 Shelf and Equalizer Sections

The low/high shelf and peaking equalizer sections implemented in `filter.lib` are described further in `filter.lib` and in [16].[12]

3.6 Lagrange/Thiran-Interpolated Fractional Delay Lines

Delay lines interpolated using higher-order FIR Lagrange interpolation are all used as follows:

```
fdelayN(maxdelay, delay, inputsignal)
```

where `N`=1,2,3, or 4 is the order of the Lagrange interpolation polynomial. Note that this API follows that of `fdelay` in `music.lib`. The requested delay should not be less than $(N-1)/2$ because the interpolating polynomial needs to be able to "reach" that far into the "past" when interpolating.

Delay lines interpolated using higher-order IIR allpass Thiran interpolation are all invoked as

```
fdelayNa(maxdelay, delay, inputsignal)
```

where `N`=1,2,3, or 4 is the order of the allpass interpolation filter. In this case, it is recommended that the requested delay be at least $N-1/2$ because an Nth-order allpass provides a delay of N samples as its coefficients approach zero. Note that delay arguments that are too small can produce an *unstable* allpass filter. For rapid delay modulations, Lagrange (FIR) interpolation is generally preferred. However, allpass interpolation introduces no gain distortion and may therefore be preferred in nearly lossless feedback loops.

3.7 Filter Banks

A *filter bank* splits its input signal into a bank of parallel signals, one for each spectral band. If the bandpass filters used to create the channel signals are carefully designed, one may sum the channel signals to get back the original input signal (possibly scaled and/or delayed). In this case, the filter bank is said to be a *Perfect Reconstruction* (PR) filter bank [20]. However, for purposes of spectrum analysis, in which only the channel signal powers are displayed, the PR condition is overkill. Therefore, the filter banks implemented in `filter.lib` are divided into *"analyzers"*, which do not have the PR property, and "filter banks" which are "allpass complementary". Allpass-complementary filter banks are reasonable choices for "graphic equalizer" applications. An allpass-complementary filter bank is PR when the allpass reduces to a pure delay and possible scaling. In this terminology, the filter banks in `filter.lib` are implemented as analyzers in cascade with delay equalizers that convert the analyzer to an allpass-complementary filter bank. Spectrum analyzer outputs should at least be nearly "power complementary", *i.e.*, the power spectra of the individual bands should at least approximately sum to the original power spectrum.

[11] `http://en.wikipedia.org/wiki/Elliptic_filter`

[12] `https: //ccrma.stanford.edu/~jos/filters/-Low_High_Shelf_Filters.html`

The typical filter bank or analyzer is constructed as a *dyadic filter bank*, meaning that it consists of a sequence of band-splits, forming a binary tree of lowpass/highpass filter sections. Since audio applications are presumed, only the lower band is split when going from one stage to the next.

In the FAUST distribution, both filter banks and spectrum analyzers are illustrated in the example `graphic_eq.dsp`. See also `spectral_level.dsp` which is a standalone spectrum analyzer (nice as a standalone JACK app).

The example `gate_compressor.dsp` included with the FAUST distribution exercises the gate and compression utilities.

Space limitations preclude further discussion here. Please see comments in `filter.lib` for further usage details.

4 Faust Library `effect.lib`

The modules in `effect.lib` classify as "digital audio effects". In general, they tend to be special-purpose *filters*, frequently nonlinear and/or time varying.

4.1 Moog Voltage Controlled Filters

New since the analog-form Moog VCF [7] is the implementation `moog_vcf_2b` of the ideal Moog VCF transfer function factored into second-order sections. As a result, its static frequency response is more accurate than `moog_vcf` which suffers from an unwanted one-sample delay in its feedback path. On the downside, its coefficient formulas are more complex when one or both parameters are varied. The `res` parameter of `moog_vcf_2b[n]` is the fourth root of that in `moog_vcf`, so, as the sampling rate approaches infinity, `moog_vcf(res,fr)` becomes equivalent to `moog_vcf_2b[n](res^4,fr)` (when `res` and `fr` are constant).

4.2 Artificial Reverberation

The reverberation modules in `effect.lib` are described in [8]. Of special note is the high-quality reverberator called `zita_rev1`, ported to FAUST from the C++ source of zita-rev1 written by Fons Adriaensen.[13] It combines Schroeder allpass and FDN reverberation techniques [8].[14]

[13] `http://kokkinizita.linuxaudio.org/linuxaudio/-zita-rev1-doc/quickguide.html`

[14] `https://ccrma.stanford.edu/~jos/pasp/Zita_Rev1.html`

5 Conclusion

Developments since LAC-08 for FAUST libraries `oscillator|filter|effect.lib` were outlined. The overall goal is to accumulate reference implementations of commonly used algorithms in music/audio signal processing, with a general preference for expressive parametric algorithms yielding the highest performed sound quality per unit of computation.

6 Acknowledgments

Special thanks to Yann Orlarey for contributing various improvements to the functions described in this paper and making others possible at all, particularly with respect to the use of pattern matching. Special thanks also to Albert Gräf for adding the pattern-matching facility to the FAUST compiler.

References

[1] Y. Orlarey, D. Fober, and S. Letz, "Syntactical and semantical aspects of FAUST", *Soft Computing*, vol. 8, no. 9, pp. 623–632, 2004.

[2] A. Gräf, "Term rewriting extension for the FAUST programming language", in *Proc. 8th Int. Linux Audio Conf. (LAC2010), Utrecht, http://lac.linuxaudio.org/*, 2010, `http://lac.linuxaudio.org/2010/papers/-30.pdf`.

[3] Y. Orlarey, A. Gräf, and S. Kersten, "DSP programming with FAUST, Q and SuperCollider", in *Proc. 4th Int. Linux Audio Conf. (LAC2006), ZKM Karlsruhe, http://lac.zkm.de/2006/proceedings.shtml*, 2006, pp. 39–40, `http://lac.zkm.de/2006/proceedings.shtml-#orlarey_et_al`.

[4] A. Gräf, "Interfacing Pure Data with FAUST", in *Proc. 5th Int. Linux Audio Conf. (LAC2007), TU Berlin, http://www.kgw.tu-berlin.de/~lac2007/-proceedings.shtml*, 2007, `http://www.kgw.tu-berlin.de/~lac2007/-papers/lac07_graef.pdf`.

[5] J. O. Smith, "Audio signal processing in FAUST", 2012, `https://ccrma.stanford.edu/~jos/aspf/`.

Moog VCF	See FAUST example `vcf_wah_pedals.dsp`
`moog_vcf(res,fr)`	analog-form Moog VCF `res` = corner-resonance amount between 0 (none) and 1 (max) `fr` = corner-resonance frequency in Hz (less than SR/6.3 or so)
`moog_vcf_2b(res,fr)`	Moog VCF implemented as two biquads (`tf2`)
`moog_vcf_2bn(res,fr)`	two protected, normalized-ladder biquads (`tf2np`)
Phasing and Flanging	See FAUST example `phaser_flanger.dsp`
`vibrato2_mono(...)`	modulated allpass-chain (see `effect.lib` for usage)
`phaser2_mono(...)`	phasing based on 2nd-order allpasses (see `effect.lib` for usage)
`phaser2_stereo(...)`	stereo phaser based on 2nd-order allpass chains
`flanger_mono(...)`	mono flanger
`flanger_stereo(...)`	stereo flanger
Envelopes/Compression/Expansion	See FAUST example `gate_compressor.dsp`
`amp_follower_ud(att,rel)`	`att` = attack time-constant (sec) going up
`amp_follower(rel)`	`rel` = release time = time-constant (sec) going down (att=0)
`autowah(level)`	`level` 0 to 1
`gate_mono(thresh,att,hold,rel)`	squelch signal when below `thresh` (in dB), for at least `hold` seconds
`gate_gain_mono(thresh,att,hold,rel,x)`	"gain computer"
`gate_stereo(thresh,att,hold,rel,x,y)`	two mono gates using same gain computer
`compressor_mono(ratio,thresh,att,rel)`	single-channel dynamic-range compression: `ratio` = compression ratio dB-in over dB-out above `thresh` `thresh` = dB level threshold above which compression kicks in
`compressor_stereo(...)`	stereo case, common gain computer
`limiter_1176_R4_mono`	= `compressor_mono(4,-6,0.0008,0.5);`
`limiter_1176_R4_stereo`	stereo case
Artificial Reverberation	See FAUST examples `freeverb|reverb_designer|zita_rev1.dsp`
`jcrev`, `satrev`	Historical early Schroeder reverberators
`fdnrev0(...)`	Feedback Delay Network (FDN) reverberator [8]
`prime_power_delays(N,pathmin,pathmax)`	utility for finding prime-power delays across a range
`zita_rev_fdn(f1,f2,t60dc,t60m,fsmax)`	order 8 FDN used in zita-rev1 - see `effect.lib` for usage
`zita_rev_stereo(...)`	stereo version of zita-rev1 - see `effect.lib` for usage
`zita_rev1_ambi(...)`	zita-rev1 in ambisonics mode
`mesh_square(N)`	`N` by `N` square digital waveguide mesh
Other Modules	
`stereo_width(w)`	stereo width effect based on the Blumlein Shuffler
`apnl(a1,a2,x)`	nonlinear allpass filter used in FAUST-STK [6]
`piano_dispersion_filter(M,B,f0)`	closed-form piano-string allpass by Rauhala et. al [21]

Figure 3: Functions defined in `effect.lib` since LAC-08.

[6] R. Michon and J. O. Smith, "FAUST-STK: A set of linear and nonlinear physical models for the FAUST programming language", in *Proc. 14th Int. Conf. Digital Audio Effects (DAFx-11), Paris, France, September 19–23*, 2011.

[7] J. O. Smith, "Virtual electric guitars and effects using FAUST and Octave", in *Proc. 6th Int. Linux Audio Conf. (LAC2008), http://lac.linuxaudio.org/*, 2008.

[8] J. O. Smith, *Physical Audio Signal Processing*, https://-ccrma.stanford.edu/~jos/pasp/, Dec. 2010, online book.

[9] J. Dattorro, "Effect design: Part 3: Oscillators: Sinusoidal and pseudonoise", *J. Audio Eng. Soc.*, vol. 50, no. 3, pp. 115–146, 2002.

[10] J. O. Smith and P. R. Cook, "The second-order digital waveguide oscillator", in *Proc. 1992 Int. Computer Music Conf., San Jose*. 1992, pp. 150–153, Computer Music Association, http://ccrma.stanford.edu/~jos/wgo/.

[11] J. W. Gordon and J. O. Smith, "A sine generation algorithm for VLSI applications", in *Proc. 1985 Int. Computer Music Conf., Vancouver*. 1985, Computer Music Association.

[12] H. Chamberlin, *Musical Applications of Microprocessors*, Hayden Book Co., Inc., New Jersey, 1980.

[13] V. Välimäki, "Discrete-time synthesis of the sawtooth waveform with reduced aliasing", *IEEE Signal Processing Letters*, vol. 12, no. 3, pp. 214–217, 2005.

[14] V. Välimäki, J. Nam, J. O. Smith, and J. S. Abel, "Alias-suppressed oscillators based on differentiated polynomial waveforms", *IEEE Trans. Audio, Speech, and Language Processing*, vol. 18, no. 5, May 2010.

[15] R. F. Voss and J. Clarke, "'1/f noise' in music: Music from 1/f noise", *J. Acoust. Soc. of Amer.*, vol. 63, no. 1, pp. 258–263, Jan. 1978.

[16] J. O. Smith, *Introduction to Digital Filters with Audio Applications*, http://-ccrma.stanford.edu/~jos/filters/, Sept. 2007, online book.

[17] A. H. Gray and J. D. Markel, "A normalized digital filter structure", *IEEE Trans. Acoustics, Speech, Signal Processing*, vol. ASSP-23, no. 3, pp. 268–277, June 1975.

[18] C. S. Burrus, *Digital Signal Processing and Digital Filter Design (Draft)*, Connexions, Sept. 2009, online book: http://cnx.org/content/col10598/latest/.

[19] T. W. Parks and C. S. Burrus, *Digital Filter Design*, John Wiley and Sons, Inc., New York, June 1987, contains FORTRAN software listings.

[20] P. P. Vaidyanathan, *Multirate Systems and Filter Banks*, Prentice-Hall, 1993.

[21] J. Rauhala and V. Välimäki, "Tunable dispersion filter design for piano synthesis", *IEEE Signal Processing Letters*, vol. 13, no. 5, pp. 253–256, May 2006.