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\footnote{http://faust.grame.fr/}, 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\footnote{\cite{1,4,5}}.

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 \(\text{maxn}(n) = \text{reduce}(\text{max},n)\) 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\footnote{\cite{6}}, such as envelope generators and table-lookup utilities.

The libraries oscillator/filter/effect.lib were first discussed at LAC-08\footnote{\cite{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 \texttt{osc(freq)} (defined in music.lib), where \(\text{freq}\) 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 \textit{filter-based}. That is, they are implemented as lossless second-order filters driven by an \textit{impulse} signal \([1,0,0,\ldots]\), and they use no wave
All algorithms have been previously published [8; 9; 10; 11].

Presently, the following algorithms are implemented:

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>oscb</td>
<td>“biquad” two-pole filter section (impulse response)</td>
</tr>
<tr>
<td>oscr</td>
<td>2D vector rotation (second-order normalized ladder) provides sine and cosine outputs</td>
</tr>
<tr>
<td>oscrs</td>
<td>sine output of oscr</td>
</tr>
<tr>
<td>oscrc</td>
<td>cosine output of oscr</td>
</tr>
<tr>
<td>oscs</td>
<td>state variable osc., cosine output (modified coupled form resonator)</td>
</tr>
<tr>
<td>oscw</td>
<td>digital waveguide oscillator</td>
</tr>
<tr>
<td>oscws</td>
<td>sine output of oscw</td>
</tr>
<tr>
<td>oscwc</td>
<td>cosine output of oscw</td>
</tr>
</tbody>
</table>

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 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 \( \text{freq}=0 \) (“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 \( s = \sin(2\pi \text{freq}/\text{SR}) \) and \( c = \cos(2\pi \text{freq}/\text{SR}) \), where \( \text{SR} \) denotes the sampling rate (defined in music.lib), there is long-term amplitude drift corresponding to the extent the identity \( s^2 + c^2 = 1 \) 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 \( \text{SR}/2 \). 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]. 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:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>imptrain(freq)</td>
<td>periodic impulse train</td>
</tr>
<tr>
<td>squarewave(freq)</td>
<td>zero-mean square wave</td>
</tr>
<tr>
<td>sawtooth(freq)</td>
<td>alias-suppressed sawtooth</td>
</tr>
<tr>
<td>sawN(freq)</td>
<td>order ( N ) anti-aliased saw</td>
</tr>
</tbody>
</table>

---

1. 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/filters/Direct_Form_II.html](https://ccrma.stanford.edu/~jos/filters/Direct_Form_II.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” [13], implemented (approximately) as white noise through a three-pole, three-zero IIR filter that approximates a 1/f power response. 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. 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:

\[
\text{iir}(bv,av) = \text{sub} \cdot \text{fir}(av) : \text{fir}(bv);
\]

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

\[
\text{fir}(bv,x) = \text{sum}(i,\text{count}(bv),\text{take}(i+1,bv) \cdot x@i);
\]

where \text{count} and \text{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.

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. This makes use of the wonderful

### Direct-Form Digital Filters

- `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)`

### Lattice/Ladder Filters

- `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

- `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

- `lowpass(N, fc)` $N$th-order Butterworth lowpass, $-3$ dB frequency at $fc$ Hz
- `highpass(N, fc)` $N$th-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.
- `highpass8e(fc)` highpass transformation of `lowpass6e` ($\omega \leftarrow 1/\omega$)
- `bandpass12e(fl, fu)` bandpass transformation of `lowpass6e`
- `bandpass6e(fl, fu)` bandpass transformation of `lowpass3e`

### Shelves, Peaking Equalizers

- `low_shelf1(L0, fx, x)` 1st-order shelf, dc gain $L0$ dB, crossover to unity gain at $fx$ Hz
- `low_shelf1L(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, $w \sim PI*B/SR$

### Fractional Delay Lines

- `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.
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)

uses order 6 elliptic band-split filters

mth_octave_analyzer6e(M, ftop, N) mth_octave analyzer followed by delay equalizer
dc-inverted variant (cheaper for odd \(O\))

mth_octave_filterbank(O, M, ftop, N) mth octave analyzer followed by delay equalizer

delay equalizer (allpass-complementary)

mth_octave_filterbank_alt mth octave analyzer followed by delay equalizer

dc-inverted variant (cheaper for odd \(O\))

mth_octave_spectral_level6e order 6 elliptic crossovers

inverted-dc variant

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

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 low-pass 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, \(4\)).

\(https://ccrma.stanford.edu/~jos/filters/Butterworth_Lowpass_Design.html\)

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 low-pass 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, \(4\)).

\(https://ccrma.stanford.edu/~jos/filters/Butterworth_Lowpass_Design.html\)
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\(^\dagger\) 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].

### 3.6 Lagrange/Thiran-Interpolated Fractional Delay Lines

Delay lines interpolated using higher-order FIR 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 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

```
fdelayN(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 `N`th-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\(^\dagger\). 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”, \textit{i.e.}, the power spectra of the individual bands should at least approximately sum to the original power spectrum.

\(^\dagger\)http://en.wikipedia.org/wiki/Elliptic_filter

\(^\dagger\)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].

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


### Moog VCF

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>moog_vcf(res,fr)</code></td>
<td>analog-form Moog VCF</td>
</tr>
<tr>
<td><code>moog_vcf_2b(res,fr)</code></td>
<td>Moog VCF implemented as two biquads (tf2)</td>
</tr>
<tr>
<td><code>moog_vcf_2bn(res,fr)</code></td>
<td>two protected, normalized-ladder biquads (tf2np)</td>
</tr>
</tbody>
</table>

#### Phasing and Flanging

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>vibrato2_mono(...)</code></td>
<td>modulated allpass-chain (see effect.lib for usage)</td>
</tr>
<tr>
<td><code>phasert2_mono(...)</code></td>
<td>phasing based on 2nd-order allpasses (see effect.lib for usage)</td>
</tr>
<tr>
<td><code>phasert2_stereo(...)</code></td>
<td>stereo phaser based on 2nd-order allpass chains</td>
</tr>
<tr>
<td><code>flanger_mono(...)</code></td>
<td>mono flanger</td>
</tr>
<tr>
<td><code>flanger_stereo(...)</code></td>
<td>stereo flanger</td>
</tr>
</tbody>
</table>

#### Envelopes/Compression/Expansion

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>amp_followed Ud(att,rel)</code></td>
<td>attack time-constant (sec) going up</td>
</tr>
<tr>
<td><code>amp_followed(rel)</code></td>
<td>release time = time-constant (sec) going down (att=0)</td>
</tr>
<tr>
<td><code>gate_mono(thresh,att,hold,rel)</code></td>
<td>squelch signal when below <code>thresh</code> (in dB), for at least <code>hold</code> seconds</td>
</tr>
<tr>
<td><code>gate_gain_mono(thresh,att,hold,rel,x)</code></td>
<td>two mono gates using same gain computer</td>
</tr>
<tr>
<td><code>gate_stereo(thresh,att,hold,rel,x,y)</code></td>
<td>single-channel dynamic-range compression:</td>
</tr>
<tr>
<td><code>compressorMono(ratio,thresh,att,rel)</code></td>
<td><code>ratio = compression ratio dB-in over dB-out above thresh</code></td>
</tr>
<tr>
<td><code>compressorStereo(...)</code></td>
<td><code>thresh = dB level threshold above which compression kicks in</code></td>
</tr>
<tr>
<td><code>limiter_1176_R4_mono</code></td>
<td>stereo case, common gain computer</td>
</tr>
<tr>
<td><code>limiter_1176_R4_stereo</code></td>
<td><code>= compressor_stereo(4,-6,0.0008,0.5);</code></td>
</tr>
</tbody>
</table>

#### Artificial Reverberation

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>jcrev, satrev</code></td>
<td>Historical early Schroeder reverberators</td>
</tr>
<tr>
<td><code>prime_power_delays(N,pathmin,pathmax)</code></td>
<td>Feedback Delay Network (FDN) reverberator [8]</td>
</tr>
<tr>
<td><code>fdnrev0(...)</code></td>
<td>utility for finding prime-power delays across a range</td>
</tr>
<tr>
<td><code>zita_rev_fdn(f1,f2,t60dc,t60m,fsmax)</code></td>
<td>order 8 FDN used in zita-rev1 - see effect.lib for usage</td>
</tr>
<tr>
<td><code>zita_rev_stereo(...)</code></td>
<td>stereo version of zita-rev1 - see effect.lib for usage</td>
</tr>
<tr>
<td><code>zita_rev1_ambi(...)</code></td>
<td>zita-rev1 in ambisonics mode</td>
</tr>
<tr>
<td><code>mesh_square(N)</code></td>
<td><code>N</code> by <code>N</code> square digital waveguide mesh</td>
</tr>
</tbody>
</table>

#### Other Modules

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>stereo_width(w)</code></td>
<td>stereo width effect based on the Blumlein Shuffler</td>
</tr>
<tr>
<td><code>apnl(a1,a2,x)</code></td>
<td>nonlinear allpass filter used in FAUST-STK [6]</td>
</tr>
<tr>
<td><code>piano_dispersion_filter(M,B,f0)</code></td>
<td>closed-form piano-string allpass by Rauhala et. al [21]</td>
</tr>
</tbody>
</table>

---


