Signal Processing Libraries for FAUST

Julius Smith
CCRMA, Stanford University

Linux Audio Conference 2012 (LAC-12)

April 14, 2012

Overview
`effect.lib`
`filter.lib`
`oscillator.lib`
Conclusion

Overview

Overview

`effect.lib`

`filter.lib`

`oscillator.lib`

Conclusion

FAUST Signal Processing Libraries

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

Highlights of Additions Since LAC-08

- `oscillator.lib`
  - Filter-Based Sinusoid Generators
  - Alias-Suppressed Classic Waveform Generators
- `filter.lib`
  - Ladder/Lattice Digital Filters
  - Audio Filter Banks
- `effect.lib`
  - Biquad-Based Moog VCFs
  - Phasing/Flanging/Compression
  - Artificial Reverberation

`effect.lib`

Moog Voltage Controlled Filters (VCF)

- `moog_vcf_2b` = ideal Moog VCF transfer function factored into second-order "biquad" sections
  - Static frequency response is more accurate than `moog_vcf` (which has an unwanted one-sample delay in its feedback path)
  - Coefficient formulas are more complex when one or both parameters are varied
- `moog_vcf_2bn` = same but using normalized ladder biquads
  - Super-robust to time-varying resonant-frequency changes (no pops!)
  - See FAUST example `vcf_wah_pedals.dsp`

Moog VCF

Moog VCF	See FAUST example `vcf_wah_pedals.dsp`
`moog_vcf(res,fr)`	analog-form Moog VCF
	`res` = corner-resonance amount [0-1]
	`fr` = corner-resonance frequency in Hz
`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

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`
`phaser2_stereo(...)`	stereo phaser based on 2nd-order allpass chains
`flanger_mono(...)`	mono flanger
`flanger_stereo(...)`	stereo flanger

Artificial Reverberation (`effect.lib`)

- General Feedback Delay Network (FDN) Reverberation

  See FAUST example `reverb_designer.dsp`

- Zita-Rev1 Reverb (FDN+Schroeder) by Fons Adriaensen (ported to FAUST)

  See FAUST example `zita_rev1.dsp`

Overview
`effect.lib`
`filter.lib`
`oscillator.lib`
Conclusion

`filter.lib`

Ladder/Lattice Digital Filters (`filter.lib`)

- Ladder and lattice digital filters have superior numerical properties
- Arbitrary Order (thanks to *pattern matching* in FAUST)
- Arbitrary (Stable) Poles and Zeros
- All Four Major Types:
    - Kelly-Lochbaum Ladder Filter
    - One-Multiply Lattice Filter
    - Two-Multiply Lattice Filter
    - Normalized Ladder Filter

Normalized Ladder Digital Filters (`filter.lib`)

Advantages of the Normalized Ladder Filter Structure:

- Signal Power Invariant wrt Coefficient Variation

⇒ Extreme Modulation is Safe

- Super-Solid Biquad (sweep it as fast as you want!):

$$\boxed{\texttt{tf2snp()}}$$

"transfer function, 2nd-order, s-plane, normalized, protected"

- See FAUST example `vcf_wah_pedals.dsp`

Ladder and Lattice Digital Filters

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 second-order normalized ladder filter
`nlf2(f,r)`	second-order normalized ladder digital filter special API

Block Diagrams

```
import("filter.lib");

bcoeffs = (1,2,3);
acoeffs = (0.1,0.2);

process = impulse <:
          iir(bcoeffs,acoeffs),
          iir_lat2(bcoeffs,acoeffs),
          iir_kl(bcoeffs,acoeffs),
          iir_lat1(bcoeffs,acoeffs)
          :> _;
```

Overview

`effect.lib`

`filter.lib`
- ladder/lattice
- normalized ladder
- filter banks

`oscillator.lib`

Conclusion

Audio Filter Banks (`filter.lib`)

- “Analyzer” $\triangleq$ Power-Complementary Band-Division (*e.g.*, for Spectral Display)

  See FAUST example `spectral_level.dsp`

- “Filterbank $\triangleq$ Allpass-Complementary Band-Division (Bands Summable Without Notch Formation)

  See FAUST example `graphic_eq.dsp`

- Filterbanks in `filter.lib` are implemented as *analyzers* in cascade with *delay equalizers* that convert the (power-complementary) analyzer to an (allpass-complementary) filter bank

`oscillator.lib`

`oscillator.lib`

Reference implementations of elementary signal generators:

- sinusoids (filter-based)
- sawtooth (bandlimited)
  - pulse-train = saw minus delayed saw
  - square = 50% duty-cycle pulse-train
  - triangle = (leakily) integrated square
  - impulse-train = differentiated saw
  - (all alias-suppressed)
- pink-noise ($1/f$ noise)

Overview

`effect.lib`

`filter.lib`

`oscillator.lib`
- sinusoids
- oscb
- oscr
- oscs
- oscw
- virtual analog
- `sawN`
- sawtooth examples
- pink noise

Conclusion

Sinusoid Generators in `oscillator.lib`

`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`

Block Diagrams

Inspect the following test program:

```
import("oscillator.lib");

freq = 100;

process = oscb(freq),
          oscrs(freq),
          oscs(freq),
          oscws(freq);
```

Sinusoidal Oscillator `oscb`

`oscb` (impulsed direct-form biquad)

- One multiply and two adds per sample of output
- Amplitude varies strongly with frequency
- Numerically poor toward `freq=0` (“dc”)
- *Nice choice for high, fixed frequencies*

Sinusoidal Oscillator `oscr`

`oscr` (2D vector rotation)

- Four multiplies and two adds per sample
- Amplitude is invariant wrt frequency
- Good down to dc
- In-phase (cosine) and phase-quadrature (sine) outputs
- Amplitude drifts over long durations at most frequencies (coefficients are roundings of `s = sin(2*PI*freq/SR)` and `c = cos(2*PI*freq/SR)`, so $s^2 + c^2 \neq 1$)
- *Nice for rapidly varying frequencies*

Sinusoidal Oscillator `oscs`

`oscs` (digitized “state variable filter”)

- “Magic Circle Algorithm” in computer graphics
- Two multiplies and two additions per output sample
- Amplitude varies much less with frequency than `oscr`
- Good down to dc
- No long-term amplitude drift
- In-phase and quadrature components available at low frequencies (exact at dc)
- *Nice lower-cost replacement for `oscr` when amplitude can vary slightly with frequency, and exact phase-quadrature outputs are not needed*

Sinusoidal Oscillator `oscw`

`oscw` (2nd-order digital waveguide oscillator)

- One multiply and three additions per sample (fixed frequency)
- Two multiplies and three additions when frequency is changing
- Same good properties as `oscr`, except
    - *No long-term amplitude drift*
    - Numerical difficulty below 10 Hz or so (not for LFOs)
    - One of the two state variables is not normalized (higher dynamic range)
- *Nice lower-cost replacement for* `oscr` *when state-variable dynamic range can be accommodated (e.g., in VLSI)*

Virtual Analog Waveforms in `oscillator.lib`

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

- `sawtooth` and `sawN` based on “Differentiated Polynomial Waveform” (DPW) method for aliasing suppression
- `sawN` uses a differentiated polynomial of order N
Increase N to reduce aliasing further
- Default case is `sawtooth = saw2 = sawN(2)`
(sounds quite good already!)
- Bandlimited square, triangle, and pulse-train derived as linear filterings of bandlimited sawtooth

FAUST Source for `sawN`

```
sawN(N,freq) = saw1 : poly(N) : D(N-1) : gate(N-1)
with {
  p0n = float(ml.SR)/float(freq); // period in samples
  lfsawpos = (_,1:fmod) ~ +(1.0/p0n); // sawtooth in [0,1)
  saw1 = 2*lfsawpos - 1; // zero-mean, amplitude +/- 1
  poly(1,x) =  x;          poly(2,x) =  x*x;
  poly(3,x) =  x*x*x - x;  ...
  diff1(x) =  (x - x’)/(2.0/p0n);
  diff(N) = seq(n,N,diff1); // N diff1s in series
  D(0) = _;
  D(1) = diff1/2.0;
  D(2) = diff(2)/6.0;
  ...
  gate(N) = *(1@(N)); // blanks startup glitch
};
```

Sawtooth Examples

FAUST Examples Using Bandlimited Sawtooth `saw2`

`(saw2(freq) = saw1(freq) <: * <: -(mem) : *(0.25'*SR/freq);)`

- `<faust>/examples/graphic_eq.dsp`
- `<faust>/examples/gate_compressor.dsp`
- `<faust>/examples/parametric_eq.dsp`
- `<faust>/examples/phaser_flanger.dsp`
- `<faust>/examples/vcf_wah_pedals.dsp`

Pink Noise

- Pink noise has the same power in every octave, making it perceptually more uniform than white noise
- `oscillator.lib` implements `pink_noise` (“$1/f$ noise”) (approximately) as white noise through a three-pole, three-zero IIR filter that approximates a $1/f$ power response:

```
pink_noise = noise :
  iir((0.049922035, -0.095993537, 0.050612699, -0.004408786),
                  (-2.494956002, 2.017265875, -0.522189400));
```

- This filter was designed using `invfreqz` in Octave (matlab) by fitting three poles and zeros to a minimum-phase $1/\sqrt{f}$ amplitude response

Conclusion

Conclusion

- Main developments in Faust signal-processing libraries `oscillator|filter|effect.lib` since LAC-08 were summarized
- Ongoing goal is accumulation of reference implementations in music/audio signal processing

Acknowledgments

Special thanks to

- Yann Orlarey for FAUST and for assistance with pattern matching
- Albert Gräf for contributing the pattern-matching facility to FAUST