Signal Processing Libraries for FAUST

Julius Smith
CCRMA, Stanford University

Linux Audio Conference 2012 (LAC-12)

April 14, 2012
FAUST Signal Processing Libraries

Overview

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

Conclusion

Julius Smith
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)

Overview

- `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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>moog_vcf(res,fr)</td>
<td>analog-form Moog VCF</td>
</tr>
<tr>
<td>res</td>
<td>corner-resonance amount [0-1]</td>
</tr>
<tr>
<td>fr</td>
<td>corner-resonance frequency in Hz</td>
</tr>
<tr>
<td>moog_vcf_2b(res,fr)</td>
<td>Moog VCF implemented as two biquads (tf2)</td>
</tr>
<tr>
<td>moog_vcf_2bn(res,fr)</td>
<td>two protected, normalized-ladder biquads (tf2np)</td>
</tr>
</tbody>
</table>

See FAUST example vcf_wah_pedals.dsp

---

### Analog-Form Moog VCF

The Moog VCF (Voltage Controlled Filter) is a fundamental component in analog synthesizers. It is primarily used for frequency modulation and filtering. The moog_vcf function is used to implement an analog-form Moog VCF with specified corner-resonance amount and frequency.

- **moog_vcf(res,fr)**
  - res: Corner-resonance amount (0-1)
  - fr: Corner-resonance frequency in Hz

### Moog VCF Implemented as Two Biquads

The moog_vcf_2b function implements the Moog VCF as two biquadratic sections (tf2).

- **moog_vcf_2b(res,fr)**
  - res: Corner-resonance amount (0-1)
  - fr: Corner-resonance frequency in Hz

### Two Protected, Normalized-Ladder Biquads

The moog_vcf_2bn function provides two protected, normalized-ladder biquadratic sections (tf2np). This implementation offers improved stability and precision.

- **moog_vcf_2bn(res,fr)**
  - res: Corner-resonance amount (0-1)
  - fr: Corner-resonance frequency in Hz
<table>
<thead>
<tr>
<th>Phasing and Flanging</th>
<th>See FAUST example phaser_flanger.dsp</th>
</tr>
</thead>
<tbody>
<tr>
<td>vibrato2_mono(...)</td>
<td>modulated allpass-chain (see effect.lib for usage)</td>
</tr>
<tr>
<td>phaser2_mono(...)</td>
<td>phasing based on 2nd-order allpasses (see effect.lib)</td>
</tr>
<tr>
<td>phaser2_stereo(...)</td>
<td>stereo phaser based on 2nd-order allpass chains</td>
</tr>
<tr>
<td>flanger_mono(...)</td>
<td>mono flanger</td>
</tr>
<tr>
<td>flanger_stereo(...)</td>
<td>stereo flanger</td>
</tr>
</tbody>
</table>
Artificial Reverberation \(\text{effect.lib}\)

- General Feedback Delay Network (FDN) Reverberation

  See FAUST example \texttt{reverb\_designer.dsp}\n
- Zita-Rev1 Reverb (FDN+Schroeder) by Fons Adriaensen (ported to FAUST)

  See FAUST example \texttt{zita\_rev1.dsp}\n
<table>
<thead>
<tr>
<th>Overview</th>
</tr>
</thead>
<tbody>
<tr>
<td>effect.lib</td>
</tr>
<tr>
<td>filter.lib</td>
</tr>
<tr>
<td>oscillator.lib</td>
</tr>
<tr>
<td>Conclusion</td>
</tr>
</tbody>
</table>

**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!):

```
tf2snp()
```

“transfer function, 2nd-order, s-plane, normalized, protected”

- See FAUST example *vcf_wah_pedals.dsp*
## Ladder and Lattice Digital Filters

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>iir.lat2(bcoeffs,acoefs)</td>
<td>two-multiply lattice digital filter</td>
</tr>
<tr>
<td>iir.kl(bcoeffs,acoefs)</td>
<td>Kelly-Lochbaum ladder digital filter</td>
</tr>
<tr>
<td>iir.lat1(bcoeffs,acoefs)</td>
<td>one-multiply lattice digital filter</td>
</tr>
<tr>
<td>iir.nl(bcoeffs,acoefs)</td>
<td>normalized ladder digital filter</td>
</tr>
<tr>
<td>tf2np(b0,b1,b2,a1,a2)</td>
<td>biquad based on stabilized second-order normalized ladder filter</td>
</tr>
<tr>
<td>nlf2(f,r)</td>
<td>second-order normalized ladder digital filter special API</td>
</tr>
</tbody>
</table>
Block Diagrams

Overview

effect.lib

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

oscillator.lib

Conclusion

```plaintext
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)
:;
Audio Filter Banks (*filter.lib*)

**Overview**
- *effect.lib*
- *filter.lib*
  - ladder/lattice
  - normalized ladder
  - filter banks
- *oscillator.lib*

**Conclusion**

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

  See [FAUST example spectral_level.dsp](#)  

- **“Filterbank”** \(\triangleleft\) 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)
# Sinusoid Generators in oscillator.lib

## Overview

<table>
<thead>
<tr>
<th>library</th>
<th>content</th>
</tr>
</thead>
<tbody>
<tr>
<td>effect.lib</td>
<td>sinusoids</td>
</tr>
<tr>
<td>filter.lib</td>
<td>oscr</td>
</tr>
<tr>
<td>oscillator.lib</td>
<td>oscrs</td>
</tr>
<tr>
<td></td>
<td>oscrc</td>
</tr>
<tr>
<td></td>
<td>oscs</td>
</tr>
<tr>
<td></td>
<td>oscw</td>
</tr>
<tr>
<td></td>
<td>oscws</td>
</tr>
<tr>
<td></td>
<td>oscwc</td>
</tr>
</tbody>
</table>
Inspect the following test program:

```plaintext
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 \( \text{freq}=0 \) ("dc")
- *Nice choice for high, fixed frequencies*
Sinusoidal Oscillator \texttt{oscr}

\texttt{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 $s = \sin(2\pi\text{freq}/\text{SR})$ and $c = \cos(2\pi\text{freq}/\text{SR})$, so $s^2 + c^2 \neq 1$)
- \textit{Nice for rapidly varying frequencies}
**Sinusoidal Oscillator** \texttt{oscs}

\texttt{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 \texttt{oscr}
- Good down to dc
- No long-term amplitude drift
- In-phase and quadrature components available at low frequencies (exact at dc)
- \textit{Nice lower-cost replacement for \texttt{oscr} when amplitude can vary slightly with frequency, and exact phase-quadrature outputs are not needed}

Julius Smith
Sinusoidal Oscillator \texttt{oscw}

\texttt{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 \texttt{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 \texttt{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**

```fmla
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** \(\text{saw2}\)
\[
(\text{saw2}(freq) = \text{saw1}(freq) <: * <: -(\text{mem}) : *(0.25' * \text{SR}/freq);)
\]

- \(<\text{faust}>/\text{examples}/\text{graphic_eq}.dsp\)
- \(<\text{faust}>/\text{examples}/\text{gate_compressor}.dsp\)
- \(<\text{faust}>/\text{examples}/\text{parametric_eq}.dsp\)
- \(<\text{faust}>/\text{examples}/\text{phaser_flanger}.dsp\)
- \(<\text{faust}>/\text{examples}/\text{vcf_wah_pedals}.dsp\)
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