lr2ms2stone

Giuseppe Silvi

December 6, 2017

author	Giuseppe Silvi
copyright	Giuseppe Silvi 2017
description	2 tracks LR to S.T.ONE a-format through MS image control
effect.lib/author	Julius O. Smith (jos at ccrma.stanford.edu)
effect.lib/copyright	Julius O. Smith III
effect.lib/deprecated	This library is deprecated and is not maintained anymore. It will be remov
effect.lib/exciter˙author	Priyanka Shekar (pshekar@ccrma.stanford.edu)
effect.lib/exciter˙copyright	Copyright (c) 2013 Priyanka Shekar
effect.lib/exciter˙license	MIT License (MIT)
effect.lib/exciter˙name	Harmonic Exciter
effect.lib/exciter˙version	1.0
effect.lib/license	STK-4.3
effect.lib/name	Faust Audio Effect Library
effect.lib/version	1.33
filter.lib/author	Julius O. Smith (jos at ccrma.stanford.edu)
filter.lib/copyright	Julius O. Smith III
filter.lib/deprecated	This library is deprecated and is not maintained anymore. It will be remov
filter.lib/license	STK-4.3
filter.lib/name	Faust Filter Library
filter.lib/reference	https://ccrma.stanford.edu/ jos/filters/
filter.lib/version	1.29
license	BSD
math.lib/author	GRAME
math.lib/copyright	GRAME
math.lib/deprecated	This library is deprecated and is not maintained anymore. It will be remov
math.lib/license	LGPL with exception
math.lib/name	Math Library
math.lib/version	1.0
music.lib/author	GRAME
music.lib/copyright	GRAME
music.lib/deprecated	This library is deprecated and is not maintained anymore. It will be remov
music.lib/license	LGPL with exception
music.lib/name	Music Library
music.lib/version	1.0
name	lr2ms2stone
reference	giuseppesilvi.com
version	1.0

This document provides a mathematical description of the Faust program text

stored in the `lr2ms2stone.dsp` file. See the notice in Section 3 (page 4) for details.

1 Mathematical definition of `process`

The *lr2ms2stone* program evaluates the signal transformer denoted by `process`, which is mathematically defined as follows:

1. Output signals y_i for $i \in [1, 4]$ such that

$$\begin{aligned} y_1(t) &= 0.5 \cdot (s_2(t) + s_1(t)) \\ y_2(t) &= 0.5 \cdot (s_2(t) - s_1(t)) \\ y_3(t) &= s_3(t) \\ y_4(t) &= 0 - s_3(t) \end{aligned}$$

2. Input signals x_i for $i \in [1, 2]$

3. User-interface input signals u_{si} for $i \in [1, 2]$ such that

   - S.T.ONE/

$$\begin{array}{lll} \text{"side"} \quad (\text{dB}) & u_{s1}(t) \in [-70, 0] & (\text{default value} = 0) \\ \text{"mid"} \quad (\text{dB}) & u_{s2}(t) \in [-70, 0] & (\text{default value} = 0) \end{array}$$

4. Intermediate signals p_i for $i \in [1, 2]$, s_i for $i \in [1, 3]$ and r_i for $i \in [1, 2]$ such that

$$\begin{aligned} p_1(t) &= 0.001 \cdot 10^{0.05 \cdot u_{s1}(t)} \\ p_2(t) &= 0.001 \cdot 10^{0.05 \cdot u_{s2}(t)} \end{aligned}$$

$$\begin{aligned} s_1(t) &= r_1(t) \cdot (x_1(t) - x_2(t)) \\ s_2(t) &= r_2(t) \cdot (x_1(t) + x_2(t)) \\ s_3(t) &= 0.5 \cdot s_1(t) \end{aligned}$$

$$\begin{aligned} r_1(t) &= p_1(t) + 0.999 \cdot r_1(t-1) \\ r_2(t) &= p_2(t) + 0.999 \cdot r_2(t-1) \end{aligned}$$

2 Block diagram of `process`

The block diagram of `process` is shown on Figure 1 (page 4).

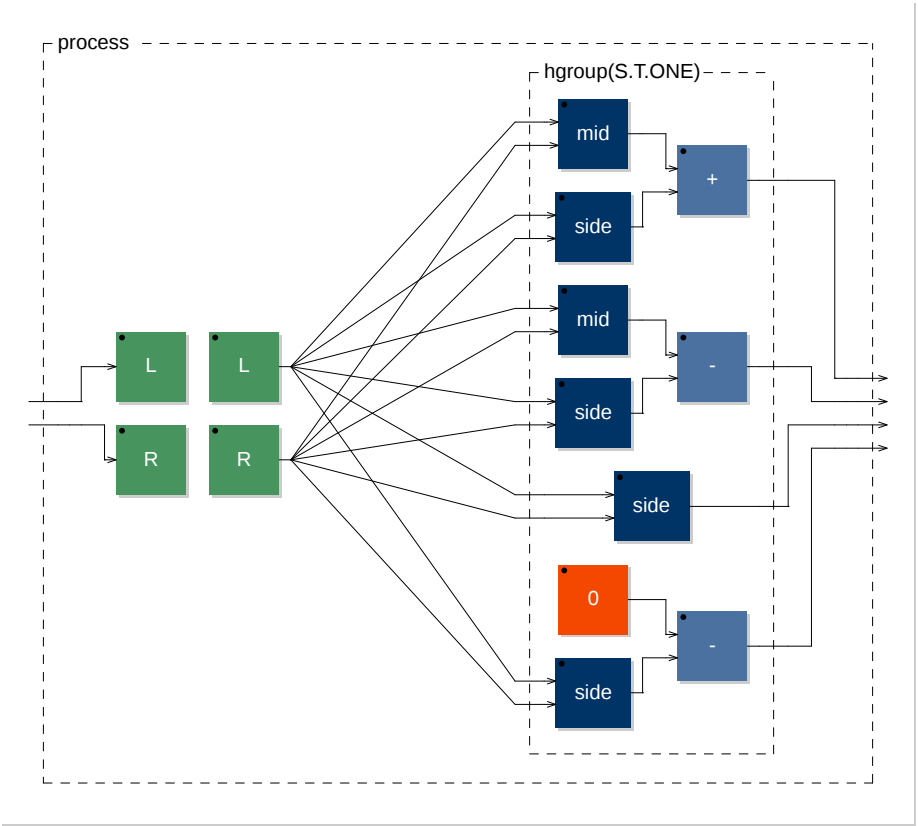

Figure 1: Block diagram of `process`

3 Notice

- This document was generated using Faust version 2.3.4 on December 06, 2017.
- The value of a Faust program is the result of applying the signal transformer denoted by the expression to which the `process` identifier is bound to input signals, running at the f_S sampling frequency.
- Faust (*Functional Audio Stream*) is a functional programming language designed for synchronous real-time signal processing and synthesis applications. A Faust program is a set of bindings of identifiers to expressions that denote signal transformers. A signal s in S is a function mapping[1] times $t \in \mathbb{Z}$ to values $s(t) \in \mathbb{R}$, while a signal transformer is a function from S^n to S^m, where $n, m \in \mathbb{N}$. See the Faust manual for additional information (http://faust.grame.fr).

[1] Faust assumes that $\forall\, s \in S, \forall\, t \in \mathbb{Z}, s(t) = 0$ when $t < 0$.

- Every mathematical formula derived from a Faust expression is assumed, in this document, to having been normalized (in an implementation-dependent manner) by the Faust compiler.
- A block diagram is a graphical representation of the Faust binding of an identifier I to an expression E; each graph is put in a box labeled by I. Subexpressions of E are recursively displayed as long as the whole picture fits in one page.
- The `lr2ms2stone-mdoc/` directory may also include the following subdirectories:
  - `cpp/` for Faust compiled code;
  - `pdf/` which contains this document;
  - `src/` for all Faust sources used (even libraries);
  - `svg/` for block diagrams, encoded using the Scalable Vector Graphics format (http://www.w3.org/Graphics/SVG/);
  - `tex/` for the LaTeX source of this document.

4 Faust code listings

This section provides the listings of the Faust code used to generate this document, including dependencies.

Listing 1: `lr2ms2stone.dsp`

```
//-----------------------------------------------------------
// 2 tracks LR to S.T.ONE a-format through MS image control -
//-----------------------------------------------------------

declare name "lr2ms2stone";
declare version "1.0";
declare author "Giuseppe Silvi";
declare copyright "Giuseppe Silvi 2017";
declare license "BSD";
declare reference "giuseppesilvi.com";
declare description "2 tracks LR to S.T.ONE a-format through MS image control";

import("effect.lib");
import("stdfaust.lib");

gain(x) = x : db2linear : smooth(0.999);

mid = ((_+_)/2)*(gain(vslider("[1] mid [unit:dB]", 0, -70, 0, 0.1)));
side = ((_-_)/2)*(gain(vslider("[2] side [unit:dB]", 0, -70, 0, 0.1)));

process(L,R) = (L,R)<:hgroup("S.T.ONE",(
  (mid + side),
  (mid - side),
  (side),
  (-side)
  ));
```

Listing 2: `effect.lib`

```
////////////////////////////////////////////////////////////////////////////////////
// WARNING: Deprecated Library!!
// Read the README file in /libraries for more information
////////////////////////////////////////////////////////////////////////////////////

declare name "Faust Audio Effect Library";
declare author "Julius O. Smith (jos at ccrma.stanford.edu)";
declare copyright "Julius O. Smith III";
declare version "1.33";
declare license "STK-4.3"; // Synthesis Tool Kit 4.3 (MIT style license)
declare deprecated "This library is deprecated and is not maintained anymore. It will be
     removed in August 2017.";

import("filter.lib"); // dcblocker*, lowpass, filterbank, ...

// The following utilities (or equivalents) could go in music.lib:

//----------------------- midikey2hz,pianokey2hz ------------------------
midikey2hz(mk) = 440.0*pow(2.0, (mk-69.0)/12); // MIDI key 69 = A440
pianokey2hz(pk) = 440.0*pow(2.0, (pk-49.0)/12); // piano key 49 = A440
hz2pianokey(f) = 12*log2(f/440.0) + 49.0;
log2(x) = log(x)/log(2.0);

//---------------- cross2, bypass1, bypass2, select2stereo --------------
//
cross2 = _,_,_,_ <: _,!,_,!,!,_,!,_;

bypass1(bpc,e) = _ <: select2(bpc,(inswitch:e),_)
               with {inswitch = select2(bpc,_,0);};

bypass2(bpc,e) = _,_ <: ((inswitch:e),_,_) : select2stereo(bpc) with {
  inswitch = _,_ : (select2(bpc,_,0), select2(bpc,_,0)) : _,_;
};

select2stereo(bpc) = cross2 : select2(bpc), select2(bpc) : _,_;

//---------------------- levelfilter, levelfilterN ----------------------
// Dynamic level lowpass filter:
//
// USAGE: levelfilter(L,freq), where
//  L    = desired level (in dB) at Nyquist limit (SR/2), e.g., -60
//  freq = corner frequency (-3dB point) usually set to fundamental freq
//
// REFERENCE:
// https://ccrma.stanford.edu/realsimple/faust_strings/Dynamic_Level_Lowpass_Filter.html
//
levelfilter(L,freq,x) = (L * L0 * x) + ((1.0-L) * lp2out(x))
with {
  L0 = pow(L,1/3);
  Lw = PI*freq/SR; // = w1 T / 2
  Lgain = Lw / (1.0 + Lw);
  Lpole2 = (1.0 - Lw) / (1.0 + Lw);
  lp2out = *(Lgain) : + ~ *(Lpole2);
};

levelfilterN(N,freq,L) = seq(i,N,levelfilter((L/N),freq));

//------------------------- speakerbp -------------------------------
// Dirt-simple speaker simulator (overall bandpass eq with observed
// roll-offs above and below the passband).
//
// Low-frequency speaker model = +12 dB/octave slope breaking to
// flat near f1. Implemented using two dc blockers in series.
//
// High-frequency model = -24 dB/octave slope implemented using a
// fourth-order Butterworth lowpass.
```

```
66  //
67  // Example based on measured Celestion G12 (12" speaker):
68  // speakerbp(130,5000);
69  //
70  // Requires filter.lib
71  //
72  speakerbp(f1,f2) = dcblockerat(f1) : dcblockerat(f1) : lowpass(4,f2);
73
74
75  //---------------------- cubicnl(drive,offset) -----------------------
76  // Cubic nonlinearity distortion
77  //
78  // USAGE: cubicnl(drive,offset), where
79  //   drive = distortion amount, between 0 and 1
80  //   offset = constant added before nonlinearity to give even harmonics
81  //           Note: offset can introduce a nonzero mean - feed
82  //           cubicnl output to dcblocker to remove this.
83  //
84  // REFERENCES:
85  //   https://ccrma.stanford.edu/~jos/pasp/Cubic_Soft_Clipper.html
86  //   https://ccrma.stanford.edu/~jos/pasp/Nonlinear_Distortion.html
87  //
88  cubicnl(drive,offset) = *(pregain) : +(offset) : clip(-1,1) : cubic
89  with {
90      pregain = pow(10.0,2*drive);
91      clip(lo,hi) = min(hi) : max(lo);
92      cubic(x) = x - x*x*x/3;
93      postgain = max(1.0,1.0/pregain);
94  };
95
96  cubicnl_nodc(drive,offset) = cubicnl(drive,offset) : dcblocker;
97
98  //-------------------------- cubicnl_demo --------------------------
99  // USAGE: _ : cubicnl_demo : _;
100 //
101 cubicnl_demo = bypass1(bp,
102    cubicnl_nodc(drive:smooth(0.999),offset:smooth(0.999)))
103 with {
104    cnl_group(x) = vgroup("CUBIC NONLINEARITY cubicnl
105         [tooltip: Reference:
106          https://ccrma.stanford.edu/~jos/pasp/Cubic_Soft_Clipper.html]", x);
107    bp = cnl_group(checkbox("[0] Bypass
108         [tooltip: When this is checked, the nonlinearity has no effect]"));
109    drive = cnl_group(hslider("[1] Drive
110                      [tooltip: Amount of distortion]",
111                      0, 0, 1, 0.01));
112    offset = cnl_group(hslider("[2] Offset
113                      [tooltip: Brings in even harmonics]",
114                      0, 0, 1, 0.01));
115 };
116
117 //------------------------------ exciter -------------------------------
118 // Psychoacoustic harmonic exciter, with GUI
119 //
120 // USAGE: _ : exciter : _
121 // REFERENCES:
122 //  https://secure.aes.org/forum/pubs/ebriefs/?elib=16939
123 //  https://www.researchgate.net/publication/258333577_Modeling_the_Harmonic_Exciter
124
125 declare exciter_name "Harmonic Exciter";
126 declare exciter_author "Priyanka Shekar (pshekar@ccrma.stanford.edu)";
127 declare exciter_copyright "Copyright (c) 2013 Priyanka Shekar";
128 declare exciter_version "1.0";
129 declare exciter_license "MIT License (MIT)";
130
131 exciter = _ <: (highpass : compressor : pregain : harmonicCreator : postgain), _ : balance
        with {
132
```

```
133 compressor = bypass1(cbp,compressorMono) with {
134
135   comp_group(x) = vgroup("COMPRESSOR [tooltip: Reference: http://en.wikipedia.org/wiki/
          Dynamic_range_compression]", x);
136
137   meter_group(x) = comp_group(hgroup("[0]", x));
138   knob_group(x) = comp_group(hgroup("[1]", x));
139
140   cbp = meter_group(checkbox("[0] Bypass [tooltip: When this is checked, the compressor has
           no effect]"));
141
142   gainview = compression_gain_mono(ratio,threshold,attack,release) : linear2db :
          meter_group(hbargraph("[1] Compressor Gain [unit:dB] [tooltip: Current gain of the
          compressor in dB]",-50,+10));
143
144   displaygain = _ <: _,abs : _,gainview : attach;
145
146   compressorMono = displaygain(compressor_mono(ratio,threshold,attack,release));
147
148   ctl_group(x) = knob_group(hgroup("[3] Compression Control", x));
149
150   ratio = ctl_group(hslider("[0] Ratio [style:knob] [tooltip: A compression Ratio of N
          means that for each N dB increase in input signal level above Threshold, the output
           level goes up 1 dB]",
151   5, 1, 20, 0.1));
152
153   threshold = ctl_group(hslider("[1] Threshold [unit:dB] [style:knob] [tooltip: When the
          signal level exceeds the Threshold (in dB), its level is compressed according to
          the Ratio]",
154   -30, -100, 10, 0.1));
155
156   env_group(x) = knob_group(hgroup("[4] Compression Response", x));
157
158   attack = env_group(hslider("[1] Attack [unit:ms] [style:knob] [tooltip: Time constant in
          ms (1/e smoothing time) for the compression gain to approach (exponentially) a new
          lower target level (the compression 'kicking in')]",
159   50, 0, 500, 0.1)) : *(0.001) : max(1/SR);
160
161   release = env_group(hslider("[2] Release [unit:ms] [style: knob] [tooltip: Time constant
          in ms (1/e smoothing time) for the compression gain to approach (exponentially) a
          new higher target level (the compression 'releasing')]",
162   500, 0, 1000, 0.1)) : *(0.001) : max(1/SR);
163
164 };
165
166 //Exciter GUI controls
167 ex_group(x) = hgroup("EXCITER [tooltip: Reference: Patent US4150253 A]", x);
168
169 //Highpass - selectable cutoff frequency
170 fc = ex_group(hslider("[0] Cutoff Frequency [unit:Hz] [style:knob] [scale:log]
171                    [tooltip: Cutoff frequency for highpassed components to be excited]",
172                    5000, 1000, 10000, 100));
173 highpass = component("filter.lib").highpass(2, fc);
174
175 //Pre-distortion gain - selectable percentage of harmonics
176 ph = ex_group(hslider("[1] Harmonics [unit:percent] [style:knob] [tooltip: Percentage of
        harmonics generated]", 20, 0, 200, 1)) / 100;
177 pregain = * (ph);
178
179 //Asymmetric cubic soft clipper
180 harmonicCreator(x) = x <: cubDist1, cubDist2, cubDist3 :> _;
181 cubDist1(x) = (x < 0) * x;
182 cubDist2(x) = (x >= 0) * (x <= 1) * (x - x ^ 3 / 3);
183 cubDist3(x) = (x > 1) * 2/3;
184
185 //Post-distortion gain - undoes effect of pre-gain
186 postgain = * (1/ph);
187
```

```
  //Balance - selectable dry/wet mix
  ml = ex_group(hslider("[2] Mix [style:knob] [tooltip: Dry/Wet mix of original signal to
       excited signal]", 0.50, 0.00, 1.00, 0.01));
  balance = (_ * ml), (_ * (1.0 - ml)) :> _;

};

//------------------------- moog_vcf(res,fr) ---------------------------
// Moog "Voltage Controlled Filter" (VCF) in "analog" form
//
// USAGE: moog_vcf(res,fr), where
//   fr = corner-resonance frequency in Hz ( less than SR/6.3 or so )
//   res  = Normalized amount of corner-resonance between 0 and 1
//        (0 is no resonance, 1 is maximum)
//
// REQUIRES: filter.lib
//
// DESCRIPTION: Moog VCF implemented using the same logical block diagram
//   as the classic analog circuit. As such, it neglects the one-sample
//   delay associated with the feedback path around the four one-poles.
//   This extra delay alters the response, especially at high frequencies
//   (see reference [1] for details).
//   See moog_vcf_2b below for a more accurate implementation.
//
// REFERENCES:
//   [1] https://ccrma.stanford.edu/~stilti/papers/moogvcf.pdf
//   [2] https://ccrma.stanford.edu/~jos/pasp/vegf.html
//
moog_vcf(res,fr) = (+ : seq(i,4,pole(p)) : *(unitygain(p))) ~ *(mk)
with {
    p = 1.0 - fr * 2.0 * PI / SR; // good approximation for fr << SR
    unitygain(p) = pow(1.0-p,4.0); // one-pole unity-gain scaling
    mk = -4.0*max(0,min(res,0.999999)); // need mk > -4 for stability
};

//----------------------- moog_vcf_2b[n] ---------------------------
// Moog "Voltage Controlled Filter" (VCF) as two biquads
//
// USAGE:
//   moog_vcf_2b(res,fr)
//   moog_vcf_2bn(res,fr)
// where
//   fr = corner-resonance frequency in Hz
//   res  = Normalized amount of corner-resonance between 0 and 1
//        (0 is min resonance, 1 is maximum)
//
// DESCRIPTION: Implementation of the ideal Moog VCF transfer
//   function factored into second-order sections. As a result, it is
//   more accurate than moog_vcf above, but its coefficient formulas are
//   more complex when one or both parameters are varied. Here, res
//   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).
//
//   moog_vcf_2b uses two direct-form biquads (tf2)
//   moog_vcf_2bn uses two protected normalized-ladder biquads (tf2np)
//
// REQUIRES: filter.lib
//
moog_vcf_2b(res,fr) = tf2s(0,0,b0,a11,a01,w1) : tf2s(0,0,b0,a12,a02,w1)
with {
 s = 1; // minus the open-loop location of all four poles
 frl = max(20,min(10000,fr)); // limit fr to reasonable 20-10k Hz range
 w1 = 2*PI*frl; // frequency-scaling parameter for bilinear xform
 // Equivalent: w1 = 1; s = 2*PI*frl;
 kmax = sqrt(2)*0.999; // 0.999 gives stability margin (tf2 is unprotected)
 k = min(kmax,sqrt(2)*res); // fourth root of Moog VCF feedback gain
 b0 = s^2;
```

```
 s2k = sqrt(2) * k;
 a11 = s * (2 + s2k);
 a12 = s * (2 - s2k);
 a01 = b0 * (1 + s2k + k^2);
 a02 = b0 * (1 - s2k + k^2);
};

moog_vcf_2bn(res,fr) = tf2snp(0,0,b0,a11,a01,w1) : tf2snp(0,0,b0,a12,a02,w1)
with {
 s = 1; // minus the open-loop location of all four poles
 w1 = 2*PI*max(fr,20); // frequency-scaling parameter for bilinear xform
 k = sqrt(2)*0.999*res; // fourth root of Moog VCF feedback gain
 b0 = s^2;
 s2k = sqrt(2) * k;
 a11 = s * (2 + s2k);
 a12 = s * (2 - s2k);
 a01 = b0 * (1 + s2k + k^2);
 a02 = b0 * (1 - s2k + k^2);
};

//------------------------ moog_vcf_demo --------------------------
// Illustrate and compare all three Moog VCF implementations above
// (called by <faust>/examples/vcf_wah_pedals.dsp).
//
// USAGE: _ : moog_vcf_demo : _;

moog_vcf_demo = bypass1(bp,vcf) with {
   mvcf_group(x) = hgroup("MOOG VCF (Voltage Controlled Filter)
      [tooltip: See Faust's effect.lib for info and references]",x);
   cb_group(x) = mvcf_group(hgroup("[0]",x));

   bp = cb_group(checkbox("[0] Bypass [tooltip: When this is checked, the Moog VCF has no
        effect]"));
   archsw = cb_group(checkbox("[1] Use Biquads
   [tooltip: Select moog_vcf_2b (two-biquad) implementation, instead of the default moog_vcf
        (analog style) implementation]"));
   bqsw = cb_group(checkbox("[2] Normalized Ladders
   [tooltip: If using biquads, make them normalized ladders (moog_vcf_2bn)]"));

   freq = mvcf_group(hslider("[1] Corner Frequency [unit:PK]
   [tooltip: The VCF resonates at the corner frequency (specified in PianoKey (PK) units,
        with A440 = 49 PK). The VCF response is flat below the corner frequency, and rolls
        off -24 dB per octave above.]",
   25, 1, 88, 0.01) : pianokey2hz) : smooth(0.999);

   res = mvcf_group(hslider("[2] Corner Resonance [style:knob]
   [tooltip: Amount of resonance near VCF corner frequency (specified between 0 and 1)]",
   0.9, 0, 1, 0.01));

   outgain = mvcf_group(hslider("[3] VCF Output Level [unit:dB] [style:knob]
   [tooltip: output level in decibels]",
   5, -60, 20, 0.1)) : component("music.lib").db2linear : smooth(0.999);

   vcfbq = _ <: select2(bqsw, moog_vcf_2b(res,freq), moog_vcf_2bn(res,freq));
   vcfarch = _ <: select2(archsw, moog_vcf(res^4,freq), vcfbq);
   vcf = vcfarch : *(outgain);
};

//------------------------ wah4(fr) -------------------------------
// Wah effect, 4th order
// USAGE: wah4(fr), where fr = resonance frequency in Hz
// REFERENCE "https://ccrma.stanford.edu/~jos/pasp/vegf.html";
//
wah4(fr) = 4*moog_vcf((3.2/4),fr:smooth(0.999));

//------------------------ wah4_demo ----------------------------
// USAGE: _ : wah4_demo : _;

```

```
wah4_demo = bypass1(bp, wah4(fr)) with {
  wah4_group(x) = hgroup("WAH4
       [tooltip: Fourth-order wah effect made using moog_vcf]", x);
  bp = wah4_group(checkbox("[0] Bypass
       [tooltip: When this is checked, the wah pedal has no effect]"));
  fr = wah4_group(hslider("[1] Resonance Frequency [scale:log]
       [tooltip: wah resonance frequency in Hz]",
     200,100,2000,1));
// Avoid dc with the moog_vcf (amplitude too high when freq comes up from dc)
// Also, avoid very high resonance frequencies (e.g., 5kHz or above).
};

//------------------------ autowah(level) -----------------------------
// Auto-wah effect
// USAGE: _ : autowah(level) : _;
// where level = amount of effect desired (0 to 1).
//
autowah(level,x) = level * crybaby(amp_follower(0.1,x),x) + (1.0-level)*x;

//-------------------------- crybaby(wah) -----------------------------
// Digitized CryBaby wah pedal
// USAGE: _ : crybaby(wah) : _;
// where wah = "pedal angle" from 0 to 1.
// REFERENCE: https://ccrma.stanford.edu/~jos/pasp/vegf.html
//
crybaby(wah) = *(gs) : tf2(1,-1,0,a1s,a2s)
with {
  Q  = pow(2.0,(2.0*(1.0-wah)+1.0)); // Resonance "quality factor"
  fr = 450.0*pow(2.0,2.3*wah);    // Resonance tuning
  g  = 0.1*pow(4.0,wah);          // gain (optional)

  // Biquad fit using z = exp(s T) ~ 1 + sT for low frequencies:
  frn = fr/SR; // Normalized pole frequency (cycles per sample)
  R = 1 - PI*frn/Q; // pole radius
  theta = 2*PI*frn; // pole angle
  a1 = 0-2.0*R*cos(theta); // biquad coeff
  a2 = R*R;              // biquad coeff

  // dezippering of slider-driven signals:
  s = 0.999; // smoothing parameter (one-pole pole location)
  a1s = a1 : smooth(s);
  a2s = a2 : smooth(s);
  gs = g  : smooth(s);

  tf2 = component("filter.lib").tf2;
};

//------------------------ crybaby_demo ---------------------------
// USAGE: _ : crybaby_demo : _ ;

crybaby_demo = bypass1(bp, crybaby(wah)) with {
   crybaby_group(x) = hgroup("CRYBABY [tooltip: Reference: https://ccrma.stanford.edu/~jos/
        pasp/vegf.html]", x);
   bp = crybaby_group(checkbox("[0] Bypass [tooltip: When this is checked, the wah pedal has
        no effect]"));
   wah = crybaby_group(hslider("[1] Wah parameter [tooltip: wah pedal angle between 0 (rocked
         back) and 1 (rocked forward)]",0.8,0,1,0.01));
};

//------------ apnl(a1,a2) ---------------
// Passive Nonlinear Allpass:
// switch between allpass coefficient a1 and a2 at signal zero crossings
// REFERENCE:
//  "A Passive Nonlinear Digital Filter Design ..."
//  by John R. Pierce and Scott A. Van Duyne,
//  JASA, vol. 101, no. 2, pp. 1120-1126, 1997
// Written by Romain Michon and JOS based on Pierce switching springs idea:
  apnl(a1,a2,x) = nonLinFilter
```

```
  with{
   condition = _>0;
   nonLinFilter = (x - _ <: _*(condition*a1 + (1-condition)*a2),_')~_ :> +;
  };

//------------ piano_dispersion_filter(M,B,f0) ---------------
// Piano dispersion allpass filter in closed form
//
// ARGUMENTS:
//   M = number of first-order allpass sections (compile-time only)
//       Keep below 20. 8 is typical for medium-sized piano strings.
//   B = string inharmonicity coefficient (0.0001 is typical)
//  f0 = fundamental frequency in Hz
//
// INPUT:
//   Signal to be filtered by the allpass chain
//
// OUTPUTS:
//  1. MINUS the estimated delay at f0 of allpass chain in samples,
//     provided in negative form to facilitate subtraction
//     from delay-line length (see USAGE below).
//  2. Output signal from allpass chain
//
// USAGE:
//  piano_dispersion_filter(1,B,f0) : +(totalDelay),_ : fdelay(maxDelay)
//
// REFERENCE:
//   "Dispersion Modeling in Waveguide Piano Synthesis
//    Using Tunable Allpass Filters",
//   by Jukka Rauhala and Vesa Valimaki, DAFX-2006, pp. 71-76
//   URL: http://www.dafx.ca/proceedings/papers/p_071.pdf
//   NOTE: An erratum in Eq. (7) is corrected in Dr. Rauhala's
//    encompassing dissertation (and below).
//   See also: http://www.acoustics.hut.fi/research/asp/piano/
//
piano_dispersion_filter(M,B,f0) = -Df0*M,seq(i,M,tf1(a1,1,a1))
with {
 a1 = (1-D)/(1+D); // By Eq. 3, have D >= 0, hence a1 >= 0 also
 D = exp(Cd - Ikey(f0)*kd);
 trt = pow(2.0,1.0/12.0); // 12th root of 2
 logb(b,x) = log(x) / log(b); // log-base-b of x
 Ikey(f0) = logb(trt,f0*trt/27.5);
 Bc = max(B,0.000001);
 kd = exp(k1*log(Bc)*log(Bc) + k2*log(Bc)+k3);
 Cd = exp((m1*log(M)+m2)*log(Bc)+m3*log(M)+m4);
 k1 = -0.00179;
 k2 = -0.0233;
 k3 = -2.93;
 m1 = 0.0126;
 m2 = 0.0606;
 m3 = -0.00825;
 m4 = 1.97;
 wT = 2*PI*f0/SR;
 polydel(a) = atan(sin(wT)/(a+cos(wT)))/wT;
 Df0 = polydel(a1) - polydel(1.0/a1);
};

//====================== Phasing and Flanging Effects ====================

//--------------- flanger_mono, flanger_stereo, flanger_demo -------------
// Flanging effect
//
// USAGE:
//     _ : flanger_mono(dmax,curdel,depth,fb,invert) : _;
//   _,_ : flanger_stereo(dmax,curdel1,curdel2,depth,fb,invert) : _,_;
//   _,_ : flanger_demo : _,_;
//
// ARGUMENTS:
```

```
//   dmax   = maximum delay-line length (power of 2) - 10 ms typical
//   curdel = current dynamic delay (not to exceed dmax)
//   depth  = effect strength between 0 and 1 (1 typical)
//   fb     = feedback gain between 0 and 1 (0 typical)
//   invert = 0 for normal, 1 to invert sign of flanging sum
//
// REFERENCE:
//    https://ccrma.stanford.edu/~jos/pasp/Flanging.html
//
flanger_mono(dmax,curdel,depth,fb,invert)
  = _ <: _, (-:fdelay(dmax,curdel)) ~ *(fb) : _,
  *(select2(invert,depth,0-depth))
  : + : *(0.5);

flanger_stereo(dmax,curdel1,curdel2,depth,fb,invert)
  =  flanger_mono(dmax,curdel1,depth,fb,invert),
     flanger_mono(dmax,curdel2,depth,fb,invert);

//------------------------- flanger_demo ----------------------------
// USAGE: _,_ : flanger_demo : _,_;
//
flanger_demo = bypass2(fbp,flanger_stereo_demo) with {
   flanger_group(x) =
    vgroup("FLANGER [tooltip: Reference: https://ccrma.stanford.edu/~jos/pasp/Flanging.html]"
          , x);
   meter_group(x) = flanger_group(hgroup("[0]", x));
   ctl_group(x) = flanger_group(hgroup("[1]", x));
   del_group(x) = flanger_group(hgroup("[2] Delay Controls", x));
   lvl_group(x) = flanger_group(hgroup("[3]", x));

   fbp = meter_group(checkbox(
        "[0] Bypass [tooltip: When this is checked, the flanger has no effect]"));
   invert = meter_group(checkbox("[1] Invert Flange Sum"));

   // FIXME: This should be an amplitude-response display:
   flangeview = lfor(freq) + lfol(freq) : meter_group(hbargraph(
      "[2] Flange LFO [style: led] [tooltip: Display sum of flange delays]", -1.5,+1.5));

   flanger_stereo_demo(x,y) = attach(x,flangeview),y :
     *(level),*(level) : flanger_stereo(dmax,curdel1,curdel2,depth,fb,invert);

   lfol = component("oscillator.lib").oscrs; // sine for left channel
   lfor = component("oscillator.lib").oscrc; // cosine for right channel
   dmax = 2048;
   dflange = 0.001 * SR *
     del_group(hslider("[1] Flange Delay [unit:ms] [style:knob]", 10, 0, 20, 0.001));
   odflange = 0.001 * SR *
     del_group(hslider("[2] Delay Offset [unit:ms] [style:knob]", 1, 0, 20, 0.001));
   freq  = ctl_group(hslider("[1] Speed [unit:Hz] [style:knob]", 0.5, 0, 10, 0.01));
   depth = ctl_group(hslider("[2] Depth [style:knob]", 1, 0, 1, 0.001));
   fb    = ctl_group(hslider("[3] Feedback [style:knob]", 0, -0.999, 0.999, 0.001));
   level = lvl_group(hslider("Flanger Output Level [unit:dB]", 0, -60, 10, 0.1)) : db2linear
         ;
   curdel1 = odflange+dflange*(1 + lfol(freq))/2;
   curdel2 = odflange+dflange*(1 + lfor(freq))/2;
};

//------- phaser2_mono, phaser2_stereo, phaser2_demo -------
// Phasing effect
//
// USAGE:
//   _ : phaser2_mono(Notches,phase,width,frqmin,fratio,frqmax,speed,depth,fb,invert) : _;
// _,_ : phaser2_stereo(") : _,_;
// _,_ : phaser2_demo : _,_;
//
// ARGUMENTS:
//   Notches = number of spectral notches (MACRO ARGUMENT - not a signal)
//   phase  = phase of the oscillator (0-1)
```

```
//   width = approximate width of spectral notches in Hz
//   frqmin = approximate minimum frequency of first spectral notch in Hz
//   fratio = ratio of adjacent notch frequencies
//   frqmax = approximate maximum frequency of first spectral notch in Hz
//   speed = LFO frequency in Hz (rate of periodic notch sweep cycles)
//   depth = effect strength between 0 and 1 (1 typical) (aka "intensity")
//           when depth=2, "vibrato mode" is obtained (pure allpass chain)
//   fb    = feedback gain between -1 and 1 (0 typical)
//   invert = 0 for normal, 1 to invert sign of flanging sum
//
// REFERENCES:
//    https://ccrma.stanford.edu/~jos/pasp/Phasing.html
//    http://www.geofex.com/Article_Folders/phasers/phase.html
//    'An Allpass Approach to Digital Phasing and Flanging', Julius O. Smith III,
//    Proc. Int. Computer Music Conf. (ICMC-84), pp. 103-109, Paris, 1984.
//    CCRMA Tech. Report STAN-M-21: https://ccrma.stanford.edu/STANM/stanms/stanm21/

vibrato2_mono(sections,phase01,fb,width,frqmin,fratio,frqmax,speed) =
 (+ : seq(i,sections,ap2p(R,th(i)))) ~ *(fb)
with {
    tf2 = component("filter.lib").tf2;
    // second-order resonant digital allpass given pole radius and angle:
    ap2p(R,th) = tf2(a2,a1,1,a1,a2) with {
      a2 = R^2;
      a1 = -2*R*cos(th);
    };
    SR = component("music.lib").SR;
    R = exp(-pi*width/SR);
    cososc = component("oscillator.lib").oscrc;
    sinosc = component("oscillator.lib").oscrs;
    osc = cososc(speed) * phase01 + sinosc(speed) * (1-phase01);
    lfo = (1-osc)/2; // in [0,1]
    pi = 4*atan(1);
    thmin = 2*pi*frqmin/SR;
    thmax = 2*pi*frqmax/SR;
    th1 = thmin + (thmax-thmin)*lfo;
    th(i) = (fratio^(i+1))*th1;
};

phaser2_mono(Notches,phase01,width,frqmin,fratio,frqmax,speed,depth,fb,invert) =
     _ <: *(g1) + g2mi*vibrato2_mono(Notches,phase01,fb,width,frqmin,fratio,frqmax,speed)
with {             // depth=0 => direct-signal only
    g1 = 1-depth/2; // depth=1 => phaser mode (equal sum of direct and allpass-chain)
    g2 = depth/2;  // depth=2 => vibrato mode (allpass-chain signal only)
    g2mi = select2(invert,g2,-g2); // inversion negates the allpass-chain signal
};

phaser2_stereo(Notches,width,frqmin,fratio,frqmax,speed,depth,fb,invert)
   = phaser2_mono(Notches,0,width,frqmin,fratio,frqmax,speed,depth,fb,invert),
     phaser2_mono(Notches,1,width,frqmin,fratio,frqmax,speed,depth,fb,invert);

//------------------------ phaser2_demo --------------------------
// USAGE: _,_ : phaser2_demo : _,_;
//
phaser2_demo = bypass2(pbp,phaser2_stereo_demo) with {
   phaser2_group(x) =
    vgroup("PHASER2 [tooltip: Reference: https://ccrma.stanford.edu/~jos/pasp/Flanging.html]"
          , x);
   meter_group(x) = phaser2_group(hgroup("[0]", x));
   ctl_group(x) = phaser2_group(hgroup("[1]", x));
   nch_group(x) = phaser2_group(hgroup("[2]", x));
   lvl_group(x) = phaser2_group(hgroup("[3]", x));

   pbp = meter_group(checkbox(
        "[0] Bypass [tooltip: When this is checked, the phaser has no effect]"));
   invert = meter_group(checkbox("[1] Invert Internal Phaser Sum"));
   vibr  = meter_group(checkbox("[2] Vibrato Mode")); // In this mode you can hear any "
        Doppler"
```

```
584
585     // FIXME: This should be an amplitude-response display:
586     //flangeview = phaser2_amp_resp : meter_group(hspectrumview("[2] Phaser Amplitude Response
            ", 0,1));
587     //phaser2_stereo_demo(x,y) = attach(x,flangeview),y : ...
588
589     phaser2_stereo_demo = *(level),*(level) :
590       phaser2_stereo(Notches,width,frqmin,fratio,frqmax,speed,mdepth,fb,invert);
591
592     Notches = 4; // Compile-time parameter: 2 is typical for analog phaser stomp-boxes
593
594     // FIXME: Add tooltips
595     speed = ctl_group(hslider("[1] Speed [unit:Hz] [style:knob]", 0.5, 0, 10, 0.001));
596     depth = ctl_group(hslider("[2] Notch Depth (Intensity) [style:knob]", 1, 0, 1, 0.001));
597     fb    = ctl_group(hslider("[3] Feedback Gain [style:knob]", 0, -0.999, 0.999, 0.001));
598
599     width = nch_group(hslider("[1] Notch width [unit:Hz] [style:knob] [scale:log]", 1000, 10,
            5000, 1));
600     frqmin = nch_group(hslider("[2] Min Notch1 Freq [unit:Hz] [style:knob] [scale:log]", 100,
            20, 5000, 1));
601     frqmax = nch_group(hslider("[3] Max Notch1 Freq [unit:Hz] [style:knob] [scale:log]", 800,
            20, 10000, 1)) : max(frqmin);
602     fratio = nch_group(hslider("[4] Notch Freq Ratio: NotchFreq(n+1)/NotchFreq(n) [style:knob]
            ", 1.5, 1.1, 4, 0.001));
603
604     level = lvl_group(hslider("Phaser Output Level [unit:dB]", 0, -60, 10, 0.1)) : component(
            "music.lib").db2linear;
605
606     mdepth = select2(vibr,depth,2); // Improve "ease of use"
607 };
608
609 //---------------------------- vocoder -------------------------
610 // A very simple vocoder where the spectrum of the modulation signal
611 // is analyzed using a filter bank.
612 //
613 // USAGE:
614 //             vocoder(nBands,att,rel,BWRatio,source,excitation) : _;
615 //
616 // where
617 // nBands = Number of vocoder bands
618 // att = Attack time in seconds
619 // rel = Release time in seconds
620 // BWRatio = Coefficient to adjust the bandwidth of each band (0.1 - 2)
621 // source = Modulation signal
622 // excitation = Excitation/Carrier signal
623
624 oneVocoderBand(band,bandsNumb,bwRatio,bandGain,x) = x : resonbp(bandFreq,bandQ,bandGain)
        with{
625       bandFreq = 25*pow(2,(band+1)*(9/bandsNumb));
626       BW = (bandFreq - 25*pow(2,(band)*(9/bandsNumb)))*bwRatio;
627       bandQ = bandFreq/BW;
628 };
629
630 vocoder(nBands,att,rel,BWRatio,source,excitation) = source <: par(i,nBands,oneVocoderBand(i,
        nBands,BWRatio,1) : amp_follower_ar(att,rel) : _,excitation : oneVocoderBand(i,nBands,
        BWRatio)) :> _ ;
631
632 //---------------------------- vocoder_demo -------------------------
633 // Use example of the vocoder function where an impulse train is used
634 // as excitation.
635 // USAGE:
636 //             _ : vocoder_demo : _;
637
638 vocoder_demo = hgroup("My Vocoder",_,impTrain(freq)*gain : vocoder(bands,att,rel,BWRatio) <:
          _,_) with{
639       bands = 32;
640       impTrain = component("oscillator.lib").lf_imptrain;
641       vocoderGroup(x) = vgroup("Vocoder",x);
```

```
642         att = vocoderGroup(hslider("[0] Attack [style:knob] [tooltip: Attack time in seconds]
                ",5,0.1,100,0.1)*0.001);
643         rel = vocoderGroup(hslider("[1] Release [style:knob] [tooltip: Release time in
                seconds]",5,0.1,100,0.1)*0.001);
644         BWRatio = vocoderGroup(hslider("[2] BW [style:knob] [tooltip: Coefficient to adjust
                the bandwidth of each band]",0.5,0.1,2,0.001));
645         excitGroup(x) = vgroup("Excitation",x);
646         freq = excitGroup(hslider("[0] Freq [style:knob]",330,50,2000,0.1));
647         gain = excitGroup(vslider("[1] Gain",0.5,0,1,0.01) : smooth(0.999));
648 };
649
650 //------------------------ stereo_width(w) ---------------------------
651 // Stereo Width effect using the Blumlein Shuffler technique.
652 //
653 // USAGE: "_,_ : stereo_width(w) : _,_", where
654 //   w = stereo width between 0 and 1
655 //
656 // At w=0, the output signal is mono ((left+right)/2 in both channels).
657 // At w=1, there is no effect (original stereo image).
658 // Thus, w between 0 and 1 varies stereo width from 0 to "original".
659 //
660 // REFERENCE:
661 // "Applications of Blumlein Shuffling to Stereo Microphone Techniques"
662 // Michael A. Gerzon, JAES vol. 42, no. 6, June 1994
663 //
664 stereo_width(w) = shuffle : *(mgain),*(sgain) : shuffle
665 with {
666     shuffle = _,_ <: +,-; // normally scaled by 1/sqrt(2) for orthonormality,
667     mgain = 1-w/2; // but we pick up the needed normalization here.
668     sgain = w/2;
669 };
670
671 //--------------------------- amp_follower ---------------------------
672 // Classic analog audio envelope follower with infinitely fast rise and
673 // exponential decay. The amplitude envelope instantaneously follows
674 // the absolute value going up, but then floats down exponentially.
675 //
676 // USAGE:
677 //   _ : amp_follower(rel) : _
678 //
679 // where
680 //  rel = release time = amplitude-envelope time-constant (sec) going down
681 //
682 // REFERENCES:
683 //  Musical Engineer's Handbook, Bernie Hutchins, Ithaca NY, 1975
684 //  Electronotes Newsletter, Bernie Hutchins
685
686 amp_follower(rel) = abs : env with {
687  p = tau2pole(rel);
688  env(x) = x * (1.0 - p) : (+ : max(x,_)) ~ *(p);
689 };
690
691 //--------------------------- amp_follower_ud ---------------------------
692 // Envelope follower with different up and down time-constants
693 // (also called a "peak detector").
694 //
695 // USAGE:
696 //   _ : amp_follower_ud(att,rel) : _
697 //
698 // where
699 //  att = attack time = amplitude-envelope time constant (sec) going up
700 //  rel = release time = amplitude-envelope time constant (sec) going down
701 //
702 // NOTE: We assume rel >> att. Otherwise, consider rel ~ max(rel,att).
703 // For audio, att is normally faster (smaller) than rel (e.g., 0.001 and 0.01).
704 // Use
705 //
706 //  _ : amp_follower_ar(att,rel) : _
```

```
//
// below to remove this restriction.
//
// REFERENCE:
//   "Digital Dynamic Range Compressor Design --- A Tutorial and Analysis", by
//   Dimitrios Giannoulis, Michael Massberg, and Joshua D. Reiss
//   http://www.eecs.qmul.ac.uk/~josh/documents/GiannoulisMassbergReiss-
     dynamicrangecompression-JAES2012.pdf

amp_follower_ud(att,rel) = amp_follower(rel) : smooth(tau2pole(att));

// Begin contributions by Jonatan Liljedahl at http://kymatica.com
// (in addition to his refinement of amp_follower above)

/*************************************************
    _ : amp_follower_ar(att,rel) : _;

Envelope follower with independent attack and release times. The
release can be shorter than the attack (unlike in amp_follower_ud
above).

*************************************************/

amp_follower_ar(att,rel) = abs : lag_ud(att,rel);

/*************************************************
    _ : lag_ud(up, dn, signal) : _;

    Lag filter with separate times for up and down.
*************************************************/

lag_ud(up,dn) = _ <: ((>,tau2pole(up),tau2pole(dn):select2),_:smooth) ~ _;

/*************************************************
    _ : peakhold(mode,sig) : _;

    Outputs current max value above zero.
    'mode' means:
    0 - Pass through. A single sample 0 trigger will work as a reset.
    1 - Track and hold max value.
*************************************************/

peakhold = (*,_:max) ~ _;

/*************************************************
    sweep(period,run);

    Counts from 0 to 'period' samples repeatedly, while 'run' is 1.
    Outsputs zero while 'run' is 0.
*************************************************/

sweep = %(int(*:max(1)))~+(1);

/*************************************************
    peakholder(holdtime, sig);

    Tracks abs peak and holds peak for 'holdtime' samples.
*************************************************/

peakholder(holdtime) = peakhold2 ~ reset : (!,_) with {
    reset = sweep(holdtime) > 0;
    // first out is gate that is 1 while holding last peak
    peakhold2 = _,abs <: peakhold,!,_ <: >=,_,!;
};

// End of contributions (so far) by Jonatan Liljedahl at http://kymatica.com

//=============== Gates, Limiters, and Dynamic Range Compression ============
```

```
774
775 //----------------- gate_mono, gate_stereo -------------------
776 // Mono and stereo signal gates
777 //
778 // USAGE:
779 //   _ : gate_mono(thresh,att,hold,rel) : _
780 // or
781 // _,_ : gate_stereo(thresh,att,hold,rel) : _,_
782 //
783 // where
784 //  thresh = dB level threshold above which gate opens (e.g., -60 dB)
785 //  att    = attack time = time constant (sec) for gate to open (e.g., 0.0001 s = 0.1 ms)
786 //  hold   = hold time = time (sec) gate stays open after signal level < thresh (e.g., 0.1 s)
787 //  rel    = release time = time constant (sec) for gate to close (e.g., 0.020 s = 20 ms)
788 //
789 // REFERENCES:
790 // - http://en.wikipedia.org/wiki/Noise_gate
791 // - http://www.soundonsound.com/sos/apr01/articles/advanced.asp
792 // - http://en.wikipedia.org/wiki/Gating_(sound_engineering)
793
794 gate_mono(thresh,att,hold,rel,x) = x * gate_gain_mono(thresh,att,hold,rel,x);
795
796 gate_stereo(thresh,att,hold,rel,x,y) = ggm*x, ggm*y with {
797   ggm = gate_gain_mono(thresh,att,hold,rel,abs(x)+abs(y));
798 };
799
800 gate_gain_mono(thresh,att,hold,rel,x) = x : extendedrawgate : amp_follower_ar(att,rel) with
      {
801   extendedrawgate(x) = max(float(rawgatesig(x)),holdsig(x));
802   rawgatesig(x) = inlevel(x) > db2linear(thresh);
803   minrate = min(att,rel);
804   inlevel = amp_follower_ar(minrate,minrate);
805   holdcounter(x) = (max(holdreset(x) * holdsamps,_) ~-(1));
806   holdsig(x) = holdcounter(x) > 0;
807   holdreset(x) = rawgatesig(x) < rawgatesig(x)'; // reset hold when raw gate falls
808   holdsamps = int(hold*SR);
809 };
810
811 //-------------------- compressor_mono, compressor_stereo ----------------------
812 // Mono and stereo dynamic range compressors
813 //
814 // USAGE:
815 //   _ : compressor_mono(ratio,thresh,att,rel) : _
816 // or
817 // _,_ : compressor_stereo(ratio,thresh,att,rel) : _,_
818 //
819 // where
820 //  ratio = compression ratio (1 = no compression, >1 means compression)
821 //  thresh = dB level threshold above which compression kicks in (0 dB = max level)
822 //  att   = attack time = time constant (sec) when level & compression going up
823 //  rel   = release time = time constant (sec) coming out of compression
824 //
825 // REFERENCES:
826 // - http://en.wikipedia.org/wiki/Dynamic_range_compression
827 // - https://ccrma.stanford.edu/~jos/filters/Nonlinear_Filter_Example_Dynamic.html
828 // - Albert Graef's <faust2pd>/examples/synth/compressor_.dsp
829 // - More features: https://github.com/magnetophon/faustCompressors
830
831 compressor_mono(ratio,thresh,att,rel,x) = x * compression_gain_mono(ratio,thresh,att,rel,x);
832
833 compressor_stereo(ratio,thresh,att,rel,x,y) = cgm*x, cgm*y with {
834   cgm = compression_gain_mono(ratio,thresh,att,rel,abs(x)+abs(y));
835 };
836
837 compression_gain_mono(ratio,thresh,att,rel) =
838   amp_follower_ar(att,rel) : linear2db : outminusindb(ratio,thresh) :
839   kneesmooth(att) : db2linear
840 with {
```

```
  // kneesmooth(att) installs a "knee" in the dynamic-range compression,
  // where knee smoothness is set equal to half that of the compression-attack.
  // A general 'knee' parameter could be used instead of tying it to att/2:
  kneesmooth(att) = smooth(tau2pole(att/2.0));
  // compression gain in dB:
   outminusindb(ratio,thresh,level) = max(level-thresh,0.0) * (1.0/float(ratio)-1.0);
  // Note: "float(ratio)" REQUIRED when ratio is an integer > 1!
};

//---------------------------- gate_demo -------------------------
// USAGE: _,_ : gate_demo : _,_;
//
gate_demo = bypass2(gbp,gate_stereo_demo) with {

   gate_group(x) = vgroup("GATE [tooltip: Reference: http://en.wikipedia.org/wiki/Noise_gate]
        ", x);
   meter_group(x) = gate_group(hgroup("[0]", x));
   knob_group(x) = gate_group(hgroup("[1]", x));

   gbp = meter_group(checkbox("[0] Bypass [tooltip: When this is checked, the gate has no
        effect]"));

   gateview = gate_gain_mono(gatethr,gateatt,gatehold,gaterel) : linear2db :
     meter_group(hbargraph("[1] Gate Gain [unit:dB] [tooltip: Current gain of the gate in dB]
          ",
      -50,+10)); // [style:led]

   gate_stereo_demo(x,y) = attach(x,gateview(abs(x)+abs(y))),y :
     gate_stereo(gatethr,gateatt,gatehold,gaterel);

   gatethr = knob_group(hslider("[1] Threshold [unit:dB] [style:knob] [tooltip: When the
        signal level falls below the Threshold (expressed in dB), the signal is muted]",
     -30, -120, 0, 0.1));

   gateatt = knob_group(hslider("[2] Attack [unit:us] [style:knob] [scale:log]
     [tooltip: Time constant in MICROseconds (1/e smoothing time) for the gate gain to go (
          exponentially) from 0 (muted) to 1 (unmuted)]",
     10, 10, 10000, 1)) : *(0.000001) : max(1.0/float(SR));

   gatehold = knob_group(hslider("[3] Hold [unit:ms] [style:knob] [scale:log]
     [tooltip: Time in ms to keep the gate open (no muting) after the signal level falls
          below the Threshold]",
     200, 1, 1000, 1)) : *(0.001) : max(1.0/float(SR));

   gaterel = knob_group(hslider("[4] Release [unit:ms] [style:knob] [scale:log]
     [tooltip: Time constant in ms (1/e smoothing time) for the gain to go (exponentially)
          from 1 (unmuted) to 0 (muted)]",
     100, 1, 1000, 1)) : *(0.001) : max(1.0/float(SR));
};

//---------------------------- compressor_demo -------------------------
// USAGE: _,_ : compressor_demo : _,_;
//
compressor_demo = bypass2(cbp,compressor_stereo_demo) with {

   comp_group(x) = vgroup("COMPRESSOR [tooltip: Reference: http://en.wikipedia.org/wiki/
        Dynamic_range_compression]", x);

   meter_group(x) = comp_group(hgroup("[0]", x));
   knob_group(x) = comp_group(hgroup("[1]", x));

   cbp = meter_group(checkbox("[0] Bypass [tooltip: When this is checked, the compressor has
        no effect]"));

   gainview =
     compression_gain_mono(ratio,threshold,attack,release) : linear2db :
     meter_group(hbargraph("[1] Compressor Gain [unit:dB] [tooltip: Current gain of the
          compressor in dB]",
```

```
899       -50,+10));
900
901     displaygain = _,_ <: _,_,(abs,abs:+) : _,_,gainview : _,attach;
902
903     compressor_stereo_demo =
904       displaygain(compressor_stereo(ratio,threshold,attack,release)) :
905       *(makeupgain), *(makeupgain);
906
907     ctl_group(x) = knob_group(hgroup("[3] Compression Control", x));
908
909     ratio = ctl_group(hslider("[0] Ratio [style:knob]
910       [tooltip: A compression Ratio of N means that for each N dB increase in input signal
              level above Threshold, the output level goes up 1 dB]",
911       5, 1, 20, 0.1));
912
913     threshold = ctl_group(hslider("[1] Threshold [unit:dB] [style:knob]
914       [tooltip: When the signal level exceeds the Threshold (in dB), its level is compressed
              according to the Ratio]",
915       -30, -100, 10, 0.1));
916
917     env_group(x) = knob_group(hgroup("[4] Compression Response", x));
918
919     attack = env_group(hslider("[1] Attack [unit:ms] [style:knob] [scale:log]
920       [tooltip: Time constant in ms (1/e smoothing time) for the compression gain to approach
              (exponentially) a new lower target level (the compression 'kicking in')]",
921       50, 1, 1000, 0.1)) : *(0.001) : max(1/SR);
922
923     release = env_group(hslider("[2] Release [unit:ms] [style: knob] [scale:log]
924       [tooltip: Time constant in ms (1/e smoothing time) for the compression gain to approach
              (exponentially) a new higher target level (the compression 'releasing')]",
925       500, 1, 1000, 0.1)) : *(0.001) : max(1/SR);
926
927     makeupgain = comp_group(hslider("[5] Makeup Gain [unit:dB]
928       [tooltip: The compressed-signal output level is increased by this amount (in dB) to make
               up for the level lost due to compression]",
929       40, -96, 96, 0.1)) : db2linear;
930 };
931
932 //------------------------------ limiter_* -----------------------------------
933 // USAGE:
934 //     _ : limiter_1176_R4_mono : _;
935 //   _,_ : limiter_1176_R4_stereo : _,_;
936 //
937 // DESCRIPTION:
938 //   A limiter guards against hard-clipping. It can be can be
939 //   implemented as a compressor having a high threshold (near the
940 //   clipping level), fast attack and release, and high ratio. Since
941 //   the ratio is so high, some knee smoothing is
942 //   desirable ("soft limiting"). This example is intended
943 //   to get you started using compressor_* as a limiter, so all
944 //   parameters are hardwired to nominal values here.
945 //
946 // REFERENCE: http://en.wikipedia.org/wiki/1176_Peak_Limiter
947 //   Ratios: 4 (moderate compression), 8 (severe compression),
948 //          12 (mild limiting), or 20 to 1 (hard limiting)
949 //   Att: 20-800 MICROseconds (Note: scaled by ratio in the 1176)
950 //   Rel: 50-1100 ms (Note: scaled by ratio in the 1176)
951 //   Mike Shipley likes 4:1 (Grammy-winning mixer for Queen, Tom Petty, etc.)
952 //     Faster attack gives "more bite" (e.g. on vocals)
953 //     He hears a bright, clear eq effect as well (not implemented here)
954 //
955 limiter_1176_R4_mono = compressor_mono(4,-6,0.0008,0.5);
956 limiter_1176_R4_stereo = compressor_stereo(4,-6,0.0008,0.5);
957
958 //========================== Schroeder Reverberators =======================
959
960 //------------------------------ jcrev,satrev --------------------------------
961 // USAGE:
```

```
//    _ : jcrev : _,_,_,_
//    _ : satrev : _,_
//
// DESCRIPTION:
//   These artificial reverberators take a mono signal and output stereo
//   (satrev) and quad (jcrev). They were implemented by John Chowning
//   in the MUS10 computer-music language (descended from Music V by Max
//   Mathews). They are Schroeder Reverberators, well tuned for their size.
//   Nowadays, the more expensive freeverb is more commonly used (see the
//   Faust examples directory).

// The reverb below was made from a listing of "RV", dated April 14, 1972,
// which was recovered from an old SAIL DART backup tape.
// John Chowning thinks this might be the one that became the
// well known and often copied JCREV:

jcrev = *(0.06) : allpass_chain <: comb_bank : mix_mtx with {

  rev1N = component("filter.lib").rev1;

  rev12(len,g) = rev1N(2048,len,g);
  rev14(len,g) = rev1N(4096,len,g);

  allpass_chain =
    rev2(512,347,0.7) :
    rev2(128,113,0.7) :
    rev2( 64, 37,0.7);

  comb_bank =
    rev12(1601,.802),
    rev12(1867,.773),
    rev14(2053,.753),
    rev14(2251,.733);

    mix_mtx = _,_,_,_ <: psum, -psum, asum, -asum : _,_,_,_ with {
    psum = _,_,_,_ :> _;
    asum = *(-1),_,*(-1),_ :> _;
  };
};

// The reverb below was made from a listing of "SATREV", dated May 15, 1971,
// which was recovered from an old SAIL DART backup tape.
// John Chowning thinks this might be the one used on his
// often-heard brass canon sound examples, one of which can be found at
// https://ccrma.stanford.edu/~jos/wav/FM_BrassCanon2.wav

satrev = *(0.2) <: comb_bank :> allpass_chain <: _,*(-1) with {

  rev1N = component("filter.lib").rev1;

  rev11(len,g) = rev1N(1024,len,g);
  rev12(len,g) = rev1N(2048,len,g);

  comb_bank =
    rev11( 778,.827),
    rev11( 901,.805),
    rev11(1011,.783),
    rev12(1123,.764);

  rev2N = component("filter.lib").rev2;

  allpass_chain =
    rev2N(128,125,0.7) :
    rev2N( 64, 42,0.7) :
    rev2N( 16, 12,0.7);
};

//---------------------------- mono_freeverb, stereo_freeverb -------------------------
```

```
// A simple Schroeder reverberator primarily developed by "Jezar at Dreampoint" that
// is extensively used in the free-software world. It uses four Schroeder allpasses in
// series and eight parallel Schroeder-Moorer filtered-feedback comb-filters for each
//audio channel, and is said to be especially well tuned.
//
// USAGE:
//             _ : mono_freeverb(fb1, fb2, damp, spread) : _;
// or
//             _,_ : stereo_freeverb(fb1, fb2, damp, spread) : _,_;
//
// where
// fb1 = coefficient of the lowpass comb filters (0-1)
// fb2 = coefficient of the allpass comb filters (0-1)
// damp = damping of the lowpass comb filter (0-1)
// spread = spatial spread in number of samples (for stereo)

mono_freeverb(fb1, fb2, damp, spread) = _ <: par(i,8,lbcf(combtuningL(i)+spread,fb1,damp))
     :> seq(i,4,allpass_comb(1024, allpasstuningL(i)+spread, -fb2))
with{
       origSR = 44100;

       // Filters parameters
       combtuningL(0) = 1116*SR/origSR : int;
       combtuningL(1) = 1188*SR/origSR : int;
       combtuningL(2) = 1277*SR/origSR : int;
       combtuningL(3) = 1356*SR/origSR : int;
       combtuningL(4) = 1422*SR/origSR : int;
       combtuningL(5) = 1491*SR/origSR : int;
       combtuningL(6) = 1557*SR/origSR : int;
       combtuningL(7) = 1617*SR/origSR : int;

       allpasstuningL(0) = 556*SR/origSR : int;
       allpasstuningL(1) = 441*SR/origSR : int;
       allpasstuningL(2) = 341*SR/origSR : int;
       allpasstuningL(3) = 225*SR/origSR : int;
       // Lowpass Feedback Combfiler:
       // https://ccrma.stanford.edu/~jos/pasp/Lowpass_Feedback_Comb_Filter.html
       lbcf(dt, fb, damp) = (+:@(dt)) ~ (*(1-damp) : (+ ~ *(damp)) : *(fb));
};

stereo_freeverb(fb1, fb2, damp, spread) = + <: mono_freeverb(fb1, fb2, damp,0),
     mono_freeverb(fb1, fb2, damp, spread);

//---------------------------- freeverb_demo -------------------------
// USAGE: _,_ : freeverb_demo : _,_;
//

freeverb_demo = _,_ <: (*(g)*fixedgain,*(g)*fixedgain : stereo_freeverb(combfeed,
     allpassfeed, damping, spatSpread)), *(1-g), *(1-g) :> _,_ with{
       scaleroom  = 0.28;
       offsetroom = 0.7;
       allpassfeed = 0.5;
       scaledamp  = 0.4;
       fixedgain  = 0.1;
       origSR = 44100;
       parameters(x) = hgroup("Freeverb",x);
       knobGroup(x) = parameters(vgroup("[0]",x));
       damping = knobGroup(vslider("[0] Damp [style: knob] [tooltip: Somehow control the
            density of the reverb.]",0.5, 0, 1, 0.025)*scaledamp*origSR/SR);
       combfeed = knobGroup(vslider("[1] RoomSize [style: knob] [tooltip: The room size
            between 0 and 1 with 1 for the largest room.]", 0.5, 0, 1, 0.025)*scaleroom*
            origSR/SR + offsetroom);
       spatSpread = knobGroup(vslider("[2] Stereo Spread [style: knob] [tooltip: Spatial
            spread between 0 and 1 with 1 for maximum spread.]",0.5,0,1,0.01)*46*SR/origSR :
             int);
       g = parameters(vslider("[1] Wet [tooltip: The amount of reverb applied to the signal
            between 0 and 1 with 1 for the maximum amount of reverb.]", 0.3333, 0, 1, 0.025)
            );
```

```
};

//=============== Feedback Delay Network (FDN) Reverberators ==============

//-------------------------------- fdnrev0 ---------------------------------
// Pure Feedback Delay Network Reverberator (generalized for easy scaling).
//
// USAGE:
//   <1,2,4,...,N signals> <:
//   fdnrev0(MAXDELAY,delays,BBSO,freqs,durs,loopgainmax,nonl) :>
//   <1,2,4,...,N signals>
//
// WHERE
//   N = 2, 4, 8, ... (power of 2)
//   MAXDELAY = power of 2 at least as large as longest delay-line length
//   delays = N delay lines, N a power of 2, lengths perferably coprime
//   BBSO = odd positive integer = order of bandsplit desired at freqs
//   freqs  = NB-1 crossover frequencies separating desired frequency bands
//   durs   = NB decay times (t60) desired for the various bands
//   loopgainmax = scalar gain between 0 and 1 used to "squelch" the reverb
//   nonl = nonlinearity (0 to 0.999..., 0 being linear)
//
// REFERENCE:
//   https://ccrma.stanford.edu/~jos/pasp/FDN_Reverberation.html
//
// DEPENDENCIES: filter.lib (filterbank)

fdnrev0(MAXDELAY, delays, BBSO, freqs, durs, loopgainmax, nonl)
  = (bus(2*N) :> bus(N) : delaylines(N)) ~
    (delayfilters(N,freqs,durs) : feedbackmatrix(N))
with {
  N = count(delays);
  NB = count(durs);
//assert(count(freqs)+1==NB);
  delayval(i) = take(i+1,delays);
  dlmax(i) = MAXDELAY; // must hardwire this from argument for now
//dlmax(i) = 2^max(1,nextpow2(delayval(i))) // try when slider min/max is known
//          with { nextpow2(x) = ceil(log(x)/log(2.0)); };
// -1 is for feedback delay:
  delaylines(N) = par(i,N,(delay(dlmax(i),(delayval(i)-1))));
  delayfilters(N,freqs,durs) = par(i,N,filter(i,freqs,durs));
  feedbackmatrix(N) = bhadamard(N);
  vbutterfly(n) = bus(n) <: (bus(n):>bus(n/2)) , ((bus(n/2),(bus(n/2):par(i,n/2,*(-1)))) :>
       bus(n/2));
  bhadamard(2) = bus(2) <: +,-;
  bhadamard(n) = bus(n) <: (bus(n):>bus(n/2)) , ((bus(n/2),(bus(n/2):par(i,n/2,*(-1)))) :>
       bus(n/2))
               : (bhadamard(n/2) , bhadamard(n/2));

  // Experimental nonlinearities:
  // nonlinallpass = apnl(nonl,-nonl);
  // s = nonl*PI;
  // nonlinallpass(x) = allpassnn(3,(s*x,s*x*x,s*x*x*x)); // filter.lib
     nonlinallpass = _; // disabled by default (rather expensive)

  filter(i,freqs,durs) = filterbank(BBSO,freqs) : par(j,NB,*(g(j,i)))
                       :> *(loopgainmax) / sqrt(N) : nonlinallpass
  with {
    dur(j) = take(j+1,durs);
    n60(j) = dur(j)*SR; // decay time in samples
    g(j,i) = exp(-3.0*log(10.0)*delayval(i)/n60(j));
        // ~ 1.0 - 6.91*delayval(i)/(SR*dur(j)); // valid for large dur(j)
  };
};

// ---------- prime_power_delays -----
// Prime Power Delay Line Lengths
//
```

```
// USAGE:
//   bus(N) : prime_power_delays(N,pathmin,pathmax) : bus(N);
//
// WHERE
//   N = positive integer up to 16
//       (for higher powers of 2, extend 'primes' array below.)
//   pathmin = minimum acoustic ray length in the reverberator (in meters)
//   pathmax = maximum acoustic ray length (meters) - think "room size"
//
// DEPENDENCIES:
//   math.lib (SR, selector, take)
//   music.lib (db2linear)
//
// REFERENCE:
//   https://ccrma.stanford.edu/~jos/pasp/Prime_Power_Delay_Line.html
//
prime_power_delays(N,pathmin,pathmax) = par(i,N,delayvals(i)) with {
  Np = 16;
  primes = 2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53;
  prime(n) = primes : selector(n,Np); // math.lib

  // Prime Power Bounds [matlab: floor(log(maxdel)./log(primes(53)))]
  maxdel=8192; // more than 63 meters at 44100 samples/sec & 343 m/s
  ppbs = 13,8,5,4, 3,3,3,3, 2,2,2,2, 2,2,2,2; // 8192 is enough for all
  ppb(i) = take(i+1,ppbs);

  // Approximate desired delay-line lengths using powers of distinct primes:
  c = 343; // soundspeed in m/s at 20 degrees C for dry air
  dmin = SR*pathmin/c;
  dmax = SR*pathmax/c;
  dl(i) = dmin * (dmax/dmin)^(i/float(N-1)); // desired delay in samples
  ppwr(i) = floor(0.5+log(dl(i))/log(prime(i))); // best prime power
  delayvals(i) = prime(i)^ppwr(i); // each delay a power of a distinct prime
};

//---------------------- stereo_reverb_tester --------------------
// Handy test inputs for reverberator demos below.

stereo_reverb_tester(revin_group,x,y) = inx,iny with {
  ck_group(x) = revin_group(vgroup("[1] Input Config",x));
  mutegain = 1 - ck_group(checkbox("[1] Mute Ext Inputs
        [tooltip: When this is checked, the stereo external audio inputs are disabled (good
              for hearing the impulse response or pink-noise response alone)]"));
  pinkin = ck_group(checkbox("[2] Pink Noise
        [tooltip: Pink Noise (or 1/f noise) is Constant-Q Noise (useful for adjusting the EQ
              sections)]"));

  impulsify = _ <: _,mem : - : >(0);
  imp_group(x) = revin_group(hgroup("[2] Impulse Selection",x));
  pulseL = imp_group(button("[1] Left
        [tooltip: Send impulse into LEFT channel]")) : impulsify;
  pulseC = imp_group(button("[2] Center
        [tooltip: Send impulse into LEFT and RIGHT channels]")) : impulsify;
  pulseR = imp_group(button("[3] Right
        [tooltip: Send impulse into RIGHT channel]")) : impulsify;

  inx = x*mutegain + (pulseL+pulseC) + pn;
  iny = y*mutegain + (pulseR+pulseC) + pn;
  pn = 0.1*pinkin*component("oscillator.lib").pink_noise;
};

//------------------------ fdnrev0_demo ---------------------------
// USAGE: _,_ : fdnrev0_demo(N,NB,BBSO) : _,_
// WHERE
//    N = Feedback Delay Network (FDN) order
//      = number of delay lines used = order of feedback matrix
//      = 2, 4, 8, or 16 [extend primes array below for 32, 64, ...]
//   NB = number of frequency bands
```

```
//      = number of (nearly) independent T60 controls
//      = integer 3 or greater
// BBSO = Butterworth band-split order
//      = order of lowpass/highpass bandsplit used at each crossover freq
//      = odd positive integer

fdnrev0_demo(N,NB,BBSO,x,y) = stereo_reverb_tester(revin_group,x,y)
         <: fdnrev0(MAXDELAY,delays,BBSO,freqs,durs,loopgainmax,nonl)
         :> *(gain),*(gain)
with {
  MAXDELAY = 8192; // sync w delays and prime_power_delays above
  defdurs = (8.4,6.5,5.0,3.8,2.7); // NB default durations (sec)
  deffreqs = (500,1000,2000,4000); // NB-1 default crossover frequencies (Hz)
  deflens = (56.3,63.0); // 2 default min and max path lengths

  fdn_group(x) = vgroup("FEEDBACK DELAY NETWORK (FDN) REVERBERATOR, ORDER 16
    [tooltip: See Faust's effect.lib for documentation and references]", x);

  freq_group(x) = fdn_group(vgroup("[1] Band Crossover Frequencies", x));
  t60_group(x) = fdn_group(hgroup("[2] Band Decay Times (T60)", x));
  path_group(x) = fdn_group(vgroup("[3] Room Dimensions", x));
  revin_group(x) = fdn_group(hgroup("[4] Input Controls", x));
  nonl_group(x) = revin_group(vgroup("[4] Nonlinearity",x));
  quench_group(x) = revin_group(vgroup("[3] Reverb State",x));

  nonl = nonl_group(hslider("[style:knob] [tooltip: nonlinear mode coupling]",
           0, -0.999, 0.999, 0.001));
  loopgainmax = 1.0-0.5*quench_group(button("[1] Quench
        [tooltip: Hold down 'Quench' to clear the reverberator]"));

  pathmin = path_group(hslider("[1] min acoustic ray length [unit:m] [scale:log]
    [tooltip: This length (in meters) determines the shortest delay-line used in the FDN
          reverberator.
             Think of it as the shortest wall-to-wall separation in the room.]",
           46, 0.1, 63, 0.1));
  pathmax = path_group(hslider("[2] max acoustic ray length [unit:m] [scale:log]
    [tooltip: This length (in meters) determines the longest delay-line used in the FDN
          reverberator.
             Think of it as the largest wall-to-wall separation in the room.]",
           63, 0.1, 63, 0.1));

  durvals(i) = t60_group(vslider("[%i] %i [unit:s] [scale:log]
    [tooltip: T60 is the 60dB decay-time in seconds. For concert halls, an overall
          reverberation time (T60) near 1.9 seconds is typical [Beranek 2004]. Here we may
          set T60 independently in each frequency band. In real rooms, higher frequency bands
           generally decay faster due to absorption and scattering.]",
    take(i+1,defdurs), 0.1, 100, 0.1));
  durs = par(i,NB,durvals(NB-1-i));

  freqvals(i) = freq_group(hslider("[%i] Band %i upper edge in Hz [unit:Hz] [scale:log]
    [tooltip: Each delay-line signal is split into frequency-bands for separate decay-time
          control in each band]",
    take(i+1,deffreqs), 100, 10000, 1));
  freqs = par(i,NB-1,freqvals(i));

  delays = prime_power_delays(N,pathmin,pathmax);

  gain = hslider("[3] Output Level (dB) [unit:dB]
    [tooltip: Output scale factor]", -40, -70, 20, 0.1) : db2linear;
     // (can cause infinite loop:) with { db2linear(x) = pow(10, x/20.0); };
};

//------------------------------ zita_rev_fdn -------------------------------
// Internal 8x8 late-reverberation FDN used in the FOSS Linux reverb zita-rev1
// by Fons Adriaensen <fons@linuxaudio.org>. This is an FDN reverb with
// allpass comb filters in each feedback delay in addition to the
// damping filters.
//
```

```
// USAGE:
//   bus(8) : zita_rev_fdn(f1,f2,t60dc,t60m,fsmax) : bus(8)
//
// WHERE
//   f1    = crossover frequency (Hz) separating dc and midrange frequencies
//   f2    = frequency (Hz) above f1 where T60 = t60m/2 (see below)
//   t60dc = desired decay time (t60) at frequency 0 (sec)
//   t60m  = desired decay time (t60) at midrange frequencies (sec)
//   fsmax = maximum sampling rate to be used (Hz)
//
// REFERENCES:
//   http://www.kokkinizita.net/linuxaudio/zita-rev1-doc/quickguide.html
//   https://ccrma.stanford.edu/~jos/pasp/Zita_Rev1.html
//
// DEPENDENCIES:
//   filter.lib (allpass_comb, lowpass, smooth)
//   math.lib (hadamard, take, etc.)

zita_rev_fdn(f1,f2,t60dc,t60m,fsmax) =
  ((bus(2*N) :> allpass_combs(N) : feedbackmatrix(N)) ~
   (delayfilters(N,freqs,durs) : fbdelaylines(N)))
with {
  N = 8;

  // Delay-line lengths in seconds:
  apdelays = (0.020346, 0.024421, 0.031604, 0.027333, 0.022904,
              0.029291, 0.013458, 0.019123); // feedforward delays in seconds
  tdelays = ( 0.153129, 0.210389, 0.127837, 0.256891, 0.174713,
              0.192303, 0.125000, 0.219991); // total delays in seconds
  tdelay(i) = floor(0.5 + SR*take(i+1,tdelays)); // samples
  apdelay(i) = floor(0.5 + SR*take(i+1,apdelays));
  fbdelay(i) = tdelay(i) - apdelay(i);
  // NOTE: Since SR is not bounded at compile time, we can't use it to
  // allocate delay lines; hence, the fsmax parameter:
  tdelaymaxfs(i) = floor(0.5 + fsmax*take(i+1,tdelays));
  apdelaymaxfs(i) = floor(0.5 + fsmax*take(i+1,apdelays));
  fbdelaymaxfs(i) = tdelaymaxfs(i) - apdelaymaxfs(i);
  nextpow2(x) = ceil(log(x)/log(2.0));
  maxapdelay(i) = int(2.0^max(1.0,nextpow2(apdelaymaxfs(i))));
  maxfbdelay(i) = int(2.0^max(1.0,nextpow2(fbdelaymaxfs(i))));

  apcoeff(i) = select2(i&1,0.6,-0.6); // allpass comb-filter coefficient
  allpass_combs(N) =
    par(i,N,(allpass_comb(maxapdelay(i),apdelay(i),apcoeff(i)))); // filter.lib
  fbdelaylines(N) = par(i,N,(delay(maxfbdelay(i),(fbdelay(i)))));
  freqs = (f1,f2); durs = (t60dc,t60m);
  delayfilters(N,freqs,durs) = par(i,N,filter(i,freqs,durs));
  feedbackmatrix(N) = hadamard(N); // math.lib

  staynormal = 10.0^(-20); // let signals decay well below LSB, but not to zero

  special_lowpass(g,f) = smooth(p) with {
    // unity-dc-gain lowpass needs gain g at frequency f => quadratic formula:
    p = mbo2 - sqrt(max(0,mbo2*mbo2 - 1.0)); // other solution is unstable
    mbo2 = (1.0 - gs*c)/(1.0 - gs); // NOTE: must ensure |g|<1 (t60m finite)
    gs = g*g;
    c = cos(2.0*PI*f/float(SR));
  };

  filter(i,freqs,durs) = lowshelf_lowpass(i)/sqrt(float(N))+staynormal
  with {
    lowshelf_lowpass(i) = gM*low_shelf1_l(g0/gM,f(1)):special_lowpass(gM,f(2));
    low_shelf1_l(G0,fx,x) = x + (G0-1)*lowpass(1,fx,x); // filter.lib
    g0 = g(0,i);
    gM = g(1,i);
    f(k) = take(k,freqs);
    dur(j) = take(j+1,durs);
    n60(j) = dur(j)*SR; // decay time in samples
```

```
    g(j,i) = exp(-3.0*log(10.0)*tdelay(i)/n60(j));
  };
};

// Stereo input delay used by zita_rev1 in both stereo and ambisonics mode:
zita_in_delay(rdel) = zita_delay_mono(rdel), zita_delay_mono(rdel) with {
  zita_delay_mono(rdel) = delay(8192,SR*rdel*0.001) * 0.3;
};

// Stereo input mapping used by zita_rev1 in both stereo and ambisonics mode:
zita_distrib2(N) = _,_ <: fanflip(N) with {
   fanflip(4) = _,_,*(-1),*(-1);
   fanflip(N) = fanflip(N/2),fanflip(N/2);
};

//---------------------------- zita_rev_fdn_demo -----------------------------
// zita_rev_fdn_demo = zita_rev_fdn (above) + basic GUI
//
// USAGE:
//   bus(8) : zita_rev_fdn_demo(f1,f2,t60dc,t60m,fsmax) : bus(8)
//
// WHERE
//   (args and references as for zita_rev_fdn above)

zita_rev_fdn_demo = zita_rev_fdn(f1,f2,t60dc,t60m,fsmax)
with {
  fsmax = 48000.0;
  fdn_group(x) = hgroup(
    "Zita_Rev Internal FDN Reverb [tooltip: ~ Zita_Rev's internal 8x8 Feedback Delay Network
         (FDN) & Schroeder allpass-comb reverberator. See Faust's effect.lib for
         documentation and references]",x);
  t60dc = fdn_group(vslider("[1] Low RT60 [unit:s] [style:knob]
    [style:knob]
    [tooltip: T60 = time (in seconds) to decay 60dB in low-frequency band]",
    3, 1, 8, 0.1));
  f1 = fdn_group(vslider("[2] LF X [unit:Hz] [style:knob] [scale:log]
    [tooltip: Crossover frequency (Hz) separating low and middle frequencies]",
    200, 50, 1000, 1));
  t60m = fdn_group(vslider("[3] Mid RT60 [unit:s] [style:knob] [scale:log]
    [tooltip: T60 = time (in seconds) to decay 60dB in middle band]",
    2, 1, 8, 0.1));
  f2 = fdn_group(vslider("[4] HF Damping [unit:Hz] [style:knob] [scale:log]
    [tooltip: Frequency (Hz) at which the high-frequency T60 is half the middle-band's T60]",
    6000, 1500, 0.49*fsmax, 1));
};

//---------------------------- zita_rev1_stereo ----------------------------
// Extend zita_rev_fdn to include zita_rev1 input/output mapping in stereo mode.
//
// USAGE:
//   _,_ : zita_rev1_stereo(rdel,f1,f2,t60dc,t60m,fsmax) : _,_
//
// WHERE
//   rdel = delay (in ms) before reverberation begins (e.g., 0 to ~100 ms)
//   (remaining args and refs as for zita_rev_fdn above)

zita_rev1_stereo(rdel,f1,f2,t60dc,t60m,fsmax) =
   zita_in_delay(rdel)
 : zita_distrib2(N)
 : zita_rev_fdn(f1,f2,t60dc,t60m,fsmax)
 : output2(N)
with {
 N = 8;
 output2(N) = outmix(N) : *(t1),*(t1);
 t1 = 0.37; // zita-rev1 linearly ramps from 0 to t1 over one buffer
 outmix(4) = !,butterfly(2),!; // probably the result of some experimenting!
 outmix(N) = outmix(N/2),par(i,N/2,!);
};
```

```
1416
1417 //------------------------------ zita_rev1_ambi ---------------------------
1418 // Extend zita_rev_fdn to include zita_rev1 input/output mapping in
1419 // "ambisonics mode", as provided in the Linux C++ version.
1420 //
1421 // USAGE:
1422 //   _,_ : zita_rev1_ambi(rgxyz,rdel,f1,f2,t60dc,t60m,fsmax) : _,_,_,_
1423 //
1424 // WHERE
1425 //   rgxyz = relative gain of lanes 1,4,2 to lane 0 in output (e.g., -9 to 9)
1426 //   (remaining args and references as for zita_rev1_stereo above)
1427
1428 zita_rev1_ambi(rgxyz,rdel,f1,f2,t60dc,t60m,fsmax) =
1429    zita_in_delay(rdel)
1430  : zita_distrib2(N)
1431  : zita_rev_fdn(f1,f2,t60dc,t60m,fsmax)
1432  : output4(N) // ambisonics mode
1433 with {
1434   N=8;
1435   output4(N) = select4 : *(t0),*(t1),*(t1),*(t1);
1436   select4 = _,_,_,!,_,!,!,! : _,_,cross with { cross(x,y) = y,x; };
1437   t0 = 1.0/sqrt(2.0);
1438   t1 = t0 * 10.0^(0.05 * rgxyz);
1439 };
1440
1441 //--------------------------------- zita_rev1 -----------------------------
1442 // Example GUI for zita_rev1_stereo (mostly following the Linux zita-rev1 GUI).
1443 //
1444 // Only the dry/wet and output level parameters are "dezippered" here. If
1445 // parameters are to be varied in real time, use "smooth(0.999)" or the like
1446 // in the same way.
1447 //
1448 // REFERENCE:
1449 //   http://www.kokkinizita.net/linuxaudio/zita-rev1-doc/quickguide.html
1450 //
1451 // DEPENDENCIES:
1452 //   filter.lib (peak_eq_rm)
1453
1454 zita_rev1(x,y) = zita_rev1_stereo(rdel,f1,f2,t60dc,t60m,fsmax,x,y)
1455          : out_eq : dry_wet(x,y) : out_level
1456 with {
1457
1458   fsmax = 48000.0; // highest sampling rate that will be used
1459
1460   fdn_group(x) = hgroup(
1461     "[0] Zita_Rev1 [tooltip: ~ ZITA REV1 FEEDBACK DELAY NETWORK (FDN) & SCHROEDER ALLPASS-
          COMB REVERBERATOR (8x8). See Faust's effect.lib for documentation and references]",
           x);
1462
1463   in_group(x) = fdn_group(hgroup("[1] Input", x));
1464
1465   rdel = in_group(vslider("[1] In Delay [unit:ms] [style:knob]
1466                 [tooltip: Delay in ms before reverberation begins]",
1467                 60,20,100,1));
1468
1469   freq_group(x) = fdn_group(hgroup("[2] Decay Times in Bands (see tooltips)", x));
1470
1471   f1 = freq_group(vslider("[1] LF X [unit:Hz] [style:knob] [scale:log]
1472       [tooltip: Crossover frequency (Hz) separating low and middle frequencies]",
1473       200, 50, 1000, 1));
1474
1475   t60dc = freq_group(vslider("[2] Low RT60 [unit:s] [style:knob] [scale:log]
1476          [style:knob] [tooltip: T60 = time (in seconds) to decay 60dB in low-frequency band]
                ",
1477          3, 1, 8, 0.1));
1478
1479   t60m = freq_group(vslider("[3] Mid RT60 [unit:s] [style:knob] [scale:log]
1480          [tooltip: T60 = time (in seconds) to decay 60dB in middle band]",
```

```
        2, 1, 8, 0.1));

  f2 = freq_group(vslider("[4] HF Damping [unit:Hz] [style:knob] [scale:log]
      [tooltip: Frequency (Hz) at which the high-frequency T60 is half the middle-band's T60
          ]",
      6000, 1500, 0.49*fsmax, 1));

  out_eq = pareq_stereo(eq1f,eq1l,eq1q) : pareq_stereo(eq2f,eq2l,eq2q);
// Zolzer style peaking eq (not used in zita-rev1) (filter.lib):
// pareq_stereo(eqf,eql,Q) = peak_eq(eql,eqf,eqf/Q), peak_eq(eql,eqf,eqf/Q);
// Regalia-Mitra peaking eq with "Q" hard-wired near sqrt(g)/2 (filter.lib):
  pareq_stereo(eqf,eql,Q) = peak_eq_rm(eql,eqf,tpbt), peak_eq_rm(eql,eqf,tpbt)
  with {
    tpbt = wcT/sqrt(max(0,g)); // tan(PI*B/SR), B bw in Hz (Q^2 ~ g/4)
    wcT = 2*PI*eqf/SR; // peak frequency in rad/sample
    g = db2linear(eql); // peak gain
  };

  eq1_group(x) = fdn_group(hgroup("[3] RM Peaking Equalizer 1", x));

  eq1f = eq1_group(vslider("[1] Eq1 Freq [unit:Hz] [style:knob] [scale:log]
      [tooltip: Center-frequency of second-order Regalia-Mitra peaking equalizer section 1]"
          ,
      315, 40, 2500, 1));

  eq1l = eq1_group(vslider("[2] Eq1 Level [unit:dB] [style:knob]
      [tooltip: Peak level in dB of second-order Regalia-Mitra peaking equalizer section 1]"
          ,
      0, -15, 15, 0.1));

  eq1q = eq1_group(vslider("[3] Eq1 Q [style:knob]
      [tooltip: Q = centerFrequency/bandwidth of second-order peaking equalizer section 1]",
      3, 0.1, 10, 0.1));

  eq2_group(x) = fdn_group(hgroup("[4] RM Peaking Equalizer 2", x));

  eq2f = eq2_group(vslider("[1] Eq2 Freq [unit:Hz] [style:knob] [scale:log]
      [tooltip: Center-frequency of second-order Regalia-Mitra peaking equalizer section 2]"
          ,
      1500, 160, 10000, 1));

  eq2l = eq2_group(vslider("[2] Eq2 Level [unit:dB] [style:knob]
      [tooltip: Peak level in dB of second-order Regalia-Mitra peaking equalizer section 2]"
          ,
      0, -15, 15, 0.1));

  eq2q = eq2_group(vslider("[3] Eq2 Q [style:knob]
      [tooltip: Q = centerFrequency/bandwidth of second-order peaking equalizer section 2]",
      3, 0.1, 10, 0.1));

  out_group(x) = fdn_group(hgroup("[5] Output", x));

  dry_wet(x,y) = *(wet) + dry*x, *(wet) + dry*y with {
    wet = 0.5*(drywet+1.0);
    dry = 1.0-wet;
  };

  drywet = out_group(vslider("[1] Dry/Wet Mix [style:knob]
      [tooltip: -1 = dry, 1 = wet]",
      0, -1.0, 1.0, 0.01)) : smooth(0.999);

  out_level = *(gain),*(gain);

  gain = out_group(vslider("[2] Level [unit:dB] [style:knob]
    [tooltip: Output scale factor]", -20, -70, 40, 0.1))
    : db2linear : smooth(0.999);

};
```

```
1544
1545 //---------------------------------- mesh_square -----------------------------
1546 // Square Rectangular Digital Waveguide Mesh
1547 //
1548 // USAGE:
1549 //   bus(4*N) : mesh_square(N) : bus(4*N);
1550 //
1551 // WHERE
1552 //   N = number of nodes along each edge - a power of two (1,2,4,8,...)
1553 //
1554 // REQUIRES: math.lib
1555 //
1556 // REFERENCE:
1557 //   https://ccrma.stanford.edu/~jos/pasp/Digital_Waveguide_Mesh.html
1558 //
1559 // SIGNAL ORDER IN AND OUT:
1560 //   The mesh is constructed recursively using 2x2 embeddings. Thus,
1561 //   the top level of mesh_square(M) is a block 2x2 mesh, where each
1562 //   block is a mesh(M/2). Let these blocks be numbered 1,2,3,4 in the
1563 //   geometry [NW,NE;SW,SE], i.e., as
1564 //         1 2
1565 //         3 4
1566 //   Each block has four vector inputs and four vector outputs, where the
1567 //   length of each vector is M/2. Label the input vectors as Ni,Ei,Wi,Si,
1568 //   i.e., as the inputs from the North, East South, and West,
1569 //   and similarly for the outputs. Then, for example, the upper
1570 //   left input block of M/2 signals is labeled 1Ni. Most of the
1571 //   connections are internal, such as 1Eo -> 2Wi. The 8*(M/2) input
1572 //   signals are grouped in the order
1573 //        1Ni 2Ni
1574 //        3Si 4Si
1575 //        1Wi 3Wi
1576 //        2Ei 4Ei
1577 //   and the output signals are
1578 //        1No 1Wo
1579 //        2No 2Eo
1580 //        3So 3Wo
1581 //        4So 4Eo
1582 //   or
1583 //         In: 1Ni 2Ni 3Si 4Si 1Wi 3Wi 2Ei 4Ei
1584 //        Out: 1No 1Wo 2No 2Eo 3So 3Wo 4So 4Eo
1585 //   Thus, the inputs are grouped by direction N,S,W,E, while the
1586 //   outputs are grouped by block number 1,2,3,4, which can also be
1587 //   interpreted as directions NW, NE, SW, SE. A simple program
1588 //   illustrating these orderings is `process = mesh_square(2);`.
1589 //
1590 // EXAMPLE: Reflectively terminated mesh impulsed at one corner:
1591 //  `mesh_square_test(N,x) = mesh_square(N)~(busi(4*N,x)) // input to corner
1592 //   with { busi(N,x) = bus(N) : par(i,N,*(-1)) : par(i,N-1,_), +(x); };
1593 //   process = 1-1' : mesh_square_test(4); // all modes excited forever`
1594 // In this simple example, the mesh edges are connected as follows:
1595 //        1No -> 1Ni, 1Wo -> 2Ni, 2No -> 3Si, 2Eo -> 4Si,
1596 //        3So -> 1Wi, 3Wo -> 3Wi, 4So -> 2Ei, 4Eo -> 4Ei
1597 // A routing matrix can be used to obtain other connection geometries.
1598
1599 // four-port scattering junction:
1600 mesh_square(1) =
1601         bus(4) <: par(i,4,*(-1)), (bus(4) :> (*(.5)) <: bus(4)) :> bus(4);
1602
1603 // rectangular NxN square waveguide mesh:
1604 mesh_square(N) = bus(4*N) : (route_inputs(N/2) : par(i,4,mesh_square(N/2)))
1605         ~(prune_feedback(N/2))
1606         : prune_outputs(N/2) : route_outputs(N/2) : bus(4*N)
1607 with {
1608   block(N) = par(i,N,!);
1609
1610   // select block i of N, block size = M:
1611   s(i,N,M) = par(j, M*N, Sv(i, j))
```

```
    with { Sv(i,i) = bus(N); Sv(i,j) = block(N); };

  // prune mesh outputs down to the signals which make it out:
  prune_outputs(N)
    = bus(16*N) :
      block(N), bus(N), block(N), bus(N),
      block(N), bus(N), bus(N),  block(N),
      bus(N),  block(N), block(N), bus(N),
      bus(N),  block(N), bus(N), block(N)
      : bus(8*N);

  // collect mesh outputs into standard order (N,W,E,S):
  route_outputs(N)
    = bus(8*N)
      <: s(4,N,8),s(5,N,8), s(0,N,8),s(2,N,8),
         s(3,N,8),s(7,N,8), s(1,N,8),s(6,N,8)
      : bus(8*N);

  // collect signals used as feedback:
  prune_feedback(N) = bus(16*N) :
      bus(N),  block(N), bus(N), block(N),
      bus(N),  block(N), block(N), bus(N),
      block(N), bus(N), bus(N),  block(N),
      block(N), bus(N), block(N), bus(N) :
      bus(8*N);

  // route mesh inputs (feedback, external inputs):
  route_inputs(N) = bus(8*N), bus(8*N)
  <:s(8,N,16),s(4,N,16), s(12,N,16),s(3,N,16),
    s(9,N,16),s(6,N,16), s(1,N,16),s(14,N,16),
    s(0,N,16),s(10,N,16), s(13,N,16),s(7,N,16),
    s(2,N,16),s(11,N,16), s(5,N,16),s(15,N,16)
    : bus(16*N);
};
```

Listing 3: `filter.lib`

```
////////////////////////////////////////////////////////////////////////////////////
// WARNING: Deprecated Library!!
// Read the README file in /libraries for more information
////////////////////////////////////////////////////////////////////////////////////

// filter.lib - digital filters of various types useful in audio and beyond

declare name "Faust Filter Library";
declare author "Julius O. Smith (jos at ccrma.stanford.edu)";
declare copyright "Julius O. Smith III";
declare version "1.29";
declare license "STK-4.3"; // Synthesis Tool Kit 4.3 (MIT style license)
declare reference "https://ccrma.stanford.edu/~jos/filters/";
declare deprecated "This library is deprecated and is not maintained anymore. It will be
     removed in August 2017.";

import("music.lib"); // delay, frac and, from math.lib, SR and PI

//---------------------- zero(z) -------------------------
// z = location of zero along real axis in z-plane
// Difference equation: y(n) = x(n) - z * x(n-1)
// Reference: https://ccrma.stanford.edu/~jos/filters/One_Zero.html

zero(z) = _ <: _,mem : _,*(z) : -;

//----------------------- pole(p) --------------------------
// p = pole location = feedback coefficient
```

```
// Could also be called a "leaky integrator".
// Difference equation: y(n) = x(n) + p * y(n-1)
// Reference: https://ccrma.stanford.edu/~jos/filters/One_Pole.html

pole(p) = + ~ *(p);

//---------------------- integrator --------------------------
// pole(1) [implemented separately for block-diagram clarity]

integrator = + ~ _ ;

//------------------ tau2pole, pole2tau ----------------------
// tau2pole(tau) returns a real pole giving exponential decay with
// tau = time-constant in seconds
// Note that t60 (time to decay 60 dB) is ~6.91 time constants.
// pole2tau(pole) returns the time-constant, in seconds,
// corresponding to the given real, positive pole in (0,1).

tau2pole(tau) = if(tau>0, exp(-1.0/(tau*SR)), 0.0);
pole2tau(pole) = if(pole<1.0, -1.0/(log(pole)*SR), 1.0e10);

//---------------------- smooth(s) --------------------------
// Exponential smoothing by a unity-dc-gain one-pole lowpass
//
// USAGE: smooth(tau2pole(tau)), where
//   tau = desired smoothing time constant in seconds,
// or
//   smooth(s), where s = smoothness between 0 and 1.
//    s=0 for no smoothing
//    s=0.999 is "very smooth"
// s>1 is unstable, and s=1 yields the zero signal for all inputs.
// The exponential time-constant is approximately
// 1/(1-s) samples, when s is close to (but less than) 1.
// Reference:
// https://ccrma.stanford.edu/~jos/mdft/Convolution_Example_2_ADSR.html

smooth(s) = *(1.0 - s) : + ~ *(s);

//------------------- dcblockerat(fb) ------------------------
// fb = "break frequency" in Hz, i.e., -3 dB gain frequency.
// The amplitude response is substantially flat above fb,
// and sloped at about +6 dB/octave below fb.
// Derived from the analog transfer function
// H(s) = s / (s + 2*PI*fb)
// by the low-frequency-matching bilinear transform method
// (i.e., the standard frequency-scaling constant 2*SR).
// Reference:
// https://ccrma.stanford.edu/~jos/pasp/Bilinear_Transformation.html

dcblockerat(fb) = *(b0) : zero(1) : pole(p)
with {
  wn = PI*fb/SR;
  b0 = 1.0 / (1 + wn);
  p = (1 - wn) * b0;
};

//---------------------- dcblocker --------------------------
// Default dc blocker has -3dB point near 35 Hz (at 44.1 kHz)
// and high-frequency gain near 1.0025 (due to no scaling)
//
dcblocker = zero(1) : pole(0.995);

//------------ notchw(width,freq), notch(freq) --------------
// width = "notch width" in Hz (approximate)
// freq = "notch frequency" in Hz
// Reference:
// https://ccrma.stanford.edu/~jos/pasp/Phasing_2nd_Order_Allpass_Filters.html

```

```
notchw(width,freq) = tf2(b0,b1,b2,a1,a2)
with {
  fb = 0.5*width; // First design a dcblockerat(width/2)
  wn = PI*fb/SR;
  b0db = 1.0 / (1 + wn);
  p = (1 - wn) * b0db; // This is our pole radius.
  // Now place unit-circle zeros at desired angles:
  tn = 2*PI*freq/SR;
  a2 = p * p;
  a2p1 = 1+a2;
  a1 = -a2p1*cos(tn);
  b1 = a1;
  b0 = 0.5*a2p1;
  b2 = b0;
};

//---------------------------- latch(c) --------------------------------
// Latch input on positive-going transition of "clock" ("sample-and-hold")
//
// USAGE:
//   _ : latch(clocksig) : _

latch(c,x) = x * s : + ~ *(1-s) with { s = ((c'<=0)&(c>0)); };

//======================== Comb Filters ================================

//---------------------- ff_comb, ff_fcomb ----------------------------
// Feed-Forward Comb Filter
//
// USAGE:
//   _ : ff_comb(maxdel,intdel,b0,bM) : _
//   _ : ff_fcomb(maxdel,del,b0,bM) : _
// where
//   maxdel = maximum delay (a power of 2)
//   intdel = current (integer) comb-filter delay between 0 and maxdel
//   del = current (float) comb-filter delay between 0 and maxdel
//   b0 = gain applied to delay-line input
//   bM = gain applied to delay-line output and then summed with input
//
// Note that ff_comb requires integer delays (uses delay() internally)
// while ff_fcomb takes floating-point delays (uses fdelay() internally).
//
// REFERENCE:
//   https://ccrma.stanford.edu/~jos/pasp/Feedforward_Comb_Filters.html

ff_comb (maxdel,M,b0,bM) = _ <: *(b0), bM * delay(maxdel,M) : + ;
ff_fcomb(maxdel,M,b0,bM) = _ <: *(b0), bM * fdelay(maxdel,M) : + ;

// Typical special case:
ffcombfilter(maxdel,del,g) = ff_comb(maxdel,del,1,g);

//---------------------- fb_comb, fb_fcomb, rev1 -----------------------
// Feed-Back Comb Filter
//
// USAGE:
//   _ : fb_comb(maxdel,intdel,b0,aN) : _
//   _ : fb_fcomb(maxdel,del,b0,aN) : _
//   _ : rev1(maxdel,del,-aN) : _
// where
//   maxdel = maximum delay (a power of 2)
//   intdel = current (integer) comb-filter delay between 0 and maxdel
//   del = current (float) comb-filter delay between 0 and maxdel
//   b0 = gain applied to delay-line input and forwarded to output
//   aN = minus the gain applied to delay-line output before
//        summing with the input and feeding to the delay line
//
// Reference:
//   https://ccrma.stanford.edu/~jos/pasp/Feedback_Comb_Filters.html
```

```
163
164 fb_comb (maxdel,N,b0,aN) = (+ <: delay(maxdel,N-1),_) ~ *(-aN) : !,*(b0):mem;
165 fb_fcomb(maxdel,N,b0,aN) = (+ <: fdelay(maxdel,float(N)-1.0),_) ~ *(-aN) : !,*(b0):mem;
166
167 // The "rev1 section" dates back to the 1960s in computer-music reverberation.
168 // See the jcrev and brassrev in effect.lib for usage examples.
169 rev1(maxdel,N,g) = fb_comb (maxdel,N,1,-g);
170
171 // Typical special case:
172 fbcombfilter(maxdel,intdel,g) = (+ : delay(maxdel,intdel)) ~ *(g);
173 ffbcombfilter(maxdel,del,g) = (+ : fdelay(maxdel,del)) ~ *(g);
174
175 //------------------- allpass_comb, allpass_fcomb, rev2 -----------------
176 // Schroeder Allpass Comb Filter
177 //
178 // USAGE:
179 //   _ : allpass_comb (maxdel,intdel,aN) : _
180 //   _ : allpass_fcomb(maxdel,del,aN) : _
181 //   _ : rev2(maxdel,del,-aN) : _
182 // where
183 //   maxdel = maximum delay (a power of 2)
184 //   intdel = current (integer) comb-filter delay between 0 and maxdel
185 //   del = current (float) comb-filter delay between 0 and maxdel
186 //   aN = minus the feedback gain
187 //
188 // Note that allpass_comb(maxlen,len,aN) =
189 //    ff_comb(maxlen,len,aN,1) :
190 //    fb_comb(maxlen,len-1,1,aN);
191 // which is a direct-form-1 implementation, requiring two delay lines.
192 // The implementation here is direct-form-2 requiring only one delay line.
193 //
194 // REFERENCES:
195 //   https://ccrma.stanford.edu/~jos/pasp/Allpass_Two_Combs.html
196 //   https://ccrma.stanford.edu/~jos/pasp/Schroeder_Allpass_Sections.html
197 //   https://ccrma.stanford.edu/~jos/filters/Four_Direct_Forms.html
198
199 allpass_comb(maxdel,N,aN) = (+ <:
200    delay(maxdel,N-1),*(aN)) ~ *(-aN)
201    : mem,_ : + ;
202
203 allpass_fcomb(maxdel,N,aN) = (+ <: fdelay(maxdel,N-1),*(aN)) ~ *(-aN) : mem,_ : + ;
204 allpass_fcomb5(maxdel,N,aN) = (+ <: fdelay5(maxdel,N-1),*(aN)) ~ *(-aN) : mem,_ : + ;
205 allpass_fcomb1a(maxdel,N,aN) = (+ <: fdelay1a(maxdel,N-1),*(aN)) ~ *(-aN) : mem,_ : + ;
206 // Interpolation helps - look at an fft of faust2octave on
207 // 1-1' <: allpass_fcomb(1024,10.5,0.95), allpass_fcomb5(1024,10.5,0.95);
208
209 // The "rev2 section" dates back to the 1960s in computer-music reverberation:
210 rev2(maxlen,len,g) = allpass_comb(maxlen,len,-g);
211
212 //================ Direct-Form Digital Filter Sections ================
213
214 // Specified by transfer-function polynomials B(z)/A(z) as in matlab
215
216 //---------------------------- iir (tfN) -------------------------------
217 // Nth-order Infinite-Impulse-Response (IIR) digital filter,
218 // implemented in terms of the Transfer-Function (TF) coefficients.
219 // Such filter structures are termed "direct form".
220 //
221 // USAGE:
222 //   _ : iir(bcoeffs,acoeffs) : _
223 // where
224 //   order = filter order (int) = max(#poles,#zeros)
225 //   bcoeffs = (b0,b1,...,b_order) = TF numerator coefficients
226 //   acoeffs =   (a1,...,a_order) = TF denominator coeffs (a0=1)
227 //
228 // REFERENCE:
229 //   https://ccrma.stanford.edu/~jos/filters/Four_Direct_Forms.html
230
```

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

//------------------------------ sub ----------------------------------
sub(x,y) = y-x; // move to math.lib?

//------------------------------ fir ----------------------------------
// FIR filter (convolution of FIR filter coefficients with a signal)
//
// USAGE:
//   _ : fir(bv) : _
// where bv = b0,b1,...,bn is a parallel bank of coefficient signals.
// NOTE: bv is processed using pattern-matching at compile time,
//       so it must have this normal form (parallel signals).
// EXAMPLE: Smoothing white noise with a five-point moving average:
//   bv = .2,.2,.2,.2,.2;
//   process = noise : fir(bv);
// EQUIVALENT (note double parens):
//   process = noise : fir((.2,.2,.2,.2,.2));

//fir(bv) = conv(bv);
fir((b0,bv)) = _ <: *(b0), R(1,bv) :> _ with {
    R(n,(bn,bv)) = (@(n):*(bn)), R(n+1,bv);
    R(n, bn)    = (@(n):*(bn));           };
fir(b0) = *(b0);

//--------------------------- conv, convN -------------------------------
// Convolution of input signal with given coefficients
//
// USAGE:
// _ : conv((k1,k2,k3,...,kN)) : _; // Argument = one signal bank
// _ : convN(N,(k1,k2,k3,...)) : _; // Useful when N < count((k1,...))

//convN(N,kv,x) = sum(i,N,take(i+1,kv) * x@i); // take() defined in math.lib
convN(N,kv)   = sum(i,N, @(i)*take(i+1,kv)); // take() defined in math.lib

//conv(kv,x) = sum(i,count(kv),take(i+1,kv) * x@i); // count() from math.lib
conv(kv) = fir(kv);


// Named special cases:
//------------------------------ tf1, tf2 ---------------------------------
// tfN = N'th-order direct-form digital filter
tf1(b0,b1,a1) = _ <: *(b0), (mem : *(b1)) :> + ~ *(0-a1);
tf2(b0,b1,b2,a1,a2) = iir((b0,b1,b2),(a1,a2)); // cf. TF2 in music.lib)
// tf2 is a variant of tf22 below with duplicated mems
tf3(b0,b1,b2,b3,a1,a2,a3) = iir((b0,b1,b2,b3),(a1,a2,a3));

//=============== Direct-Form second-order biquad sections ================
// REFERENCE: https://ccrma.stanford.edu/~jos/filters/Four_Direct_Forms.html

//---------------------- tf21, tf21t, tf22, tf22t ------------------------
tf21(b0,b1,b2,a1,a2) = // tf2, direct-form 1:
    _ <:(mem<:((mem:*(b2)),*(b1))),*(b0) :>_
    : ((_,_,_:>_) ~(_<:*(-a1),(mem:*(-a2))));
tf22(b0,b1,b2,a1,a2) = // tf2, direct-form 2:
    _ : (((_,_,_:>_)~*(-a1)<:mem,*(b0))~*(-a2))
      : (_<:mem,*(b1)),_ : *(b2),_,_ :> _;
tf22t(b0,b1,b2,a1,a2) = // tf2, direct-form 2 transposed:
    _ : (_,_,(_ <: *(b2)',*(b1)',*(b0))
      : _,+',_,_ :> _)~*(-a1)~*(-a2) : _;
tf21t(b0,b1,b2,a1,a2) = // tf2, direct-form 1 transposed:
    tf22t(1,0,0,a1,a2) : tf22t(b0,b1,b2,0,0); // or write it out if you want

//==================== Ladder/Lattice Digital Filters =====================
// Ladder and lattice digital filters generally have superior numerical
// properties relative to direct-form digital filters. They can be derived
// from digital waveguide filters, which gives them a physical interpretation.

```

```
// REFERENCES:
//  F. Itakura and S. Saito: "Digital Filtering Techniques for Speech Analysis and Synthesis",
//    7th Int. Cong. Acoustics, Budapest, 25 C 1, 1971.
//  J. D. Markel and A. H. Gray: Linear Prediction of Speech, New York: Springer Verlag, 1976.
//  https://ccrma.stanford.edu/~jos/pasp/Conventional_Ladder_Filters.html

//------------------------ block, crossn,crossn1 -----------------------------
// signal block/crossing utilities
// (move to math.lib?)

// block - terminate n signals (goes with bus(n) in math.lib)
block(n) = par(i,n,!);

// crossnn - cross two bus(n)s:
crossnn(n) = bus(n),bus(n) <: block(n),bus(n),bus(n),block(n);

// crossn1 - cross bus(n) and bus(1):
crossn1(n) = bus(n),(bus(1)<:bus(n)) <: block(n),bus(n),bus(n),block(n):bus(1),block(n-1),bus(n);

//------------------------------- av2sv ----------------------------------
// Compute reflection coefficients sv from transfer-function denominator av
//
// USAGE:
//   sv = av2sv(av)
// where
//   av = parallel signal bank a1,...,aN
//   sv = parallel signal bank s1,...,sN
// where si = ith reflection coefficient, and
//      ai = coefficient of z^(-i) in the filter
//         transfer-function denominator A(z).
//
// REFERENCE:
//   https://ccrma.stanford.edu/~jos/filters/Step_Down_Procedure.html
//   (where reflection coefficients are denoted by k rather than s).

av2sv(av) = par(i,M,s(i+1)) with {
  M = count(av);
  s(m) = sr(M-m+1); // m=1..M
  sr(m) = Ari(m,M-m+1); // s_{M-1-m}
  Ari(m,i) = take(i+1,Ar(m-1));
  //step-down recursion for lattice/ladder digital filters:
  Ar(0) = (1,av); // Ar(m) is order M-m (i.e. "reverse-indexed")
  Ar(m) = 1,par(i,M-m, (Ari(m,i+1) - sr(m)*Ari(m,M-m-i))/(1-sr(m)*sr(m)));
};

//---------------------------- bvav2nuv -------------------------------
// Compute lattice tap coefficients from transfer-function coefficients
//
// USAGE:
//   nuv = bvav2nuv(bv,av)
// where
//   av = parallel signal bank a1,...,aN
//   bv = parallel signal bank b0,b1,...,aN
//  nuv = parallel signal bank nu1,...,nuN
// where nui is the i'th tap coefficient,
//      bi is the coefficient of z^(-i) in the filter numerator,
//      ai is the coefficient of z^(-i) in the filter denominator

bvav2nuv(bv,av) = par(m,M+1,nu(m)) with {
  M = count(av);
  nu(m) = take(m+1,Pr(M-m)); // m=0..M
  // lattice/ladder tap parameters:
  Pr(0) = bv; // Pr(m) is order M-m, 'r' means "reversed"
  Pr(m) = par(i,M-m+1, (Pri(m,i) - nu(M-m+1)*Ari(m,M-m-i+1)));
  Pri(m,i) = take(i+1,Pr(m-1));
```

```
  Ari(m,i) = take(i+1,Ar(m-1));
  //step-down recursion for lattice/ladder digital filters:
  Ar(0) = (1,av); // Ar(m) is order M-m (recursion index must start at constant)
  Ar(m) = 1,par(i,M-m, (Ari(m,i+1) - sr(m)*Ari(m,M-m-i))/(1-sr(m)*sr(m)));
  sr(m) = Ari(m,M-m+1); // s_{M-1-m}
};

//--------------------------- iir_lat2, allpassnt ----------------------

iir_lat2(bv,av) = allpassnt(M,sv) : sum(i,M+1,*(take(M-i+1,tg)))
with {
  M = count(av);
  sv = av2sv(av); // sv = vector of sin(theta) reflection coefficients
  tg = bvav2nuv(bv,av); // tg = vector of tap gains
};

// two-multiply lattice allpass (nested order-1 direct-form-ii allpasses):
allpassnt(0,sv) = _;
allpassnt(n,sv) =
//0:  x <: ((+ <: (allpassnt(n-1,sv)),*(s))~*(-s))) : _',_ :+
      _ : ((+ <: (allpassnt(n-1,sv),*(s)))~*(-s)) : fsec(n)
with {
  fsec(1) = crossnn(1) : _, (_<:mem,_) : +,_;
  fsec(n) = crossn1(n) : _, (_<:mem,_),par(i,n-1,_) : +, par(i,n,_);
  innertaps(n) = par(i,n,_);
  s = take(n,sv); // reflection coefficient s = sin(theta)
};

//------------------------------ iir_kl, allpassnklt -------------------------
iir_kl(bv,av) = allpassnklt(M,sv) : sum(i,M+1,*(tghr(i)))
with {
  M = count(av);
  sv = av2sv(av); // sv = vector of sin(theta) reflection coefficients
  tg = bvav2nuv(bv,av); // tg = vector of tap gains for 2mul case
  tgr(i) = take(M+1-i,tg);
  tghr(n) = tgr(n)/pi(n);
  pi(0) = 1;
  pi(n) = pi(n-1)*(1+take(M-n+1,sv)); // all sign parameters '+'
};

// Kelly-Lochbaum ladder allpass with tap lines:
allpassnklt(0,sv) = _;
allpassnklt(n,sv) = _ <: *(s),(*(1+s) : (+
                 : allpassnklt(n-1,sv))~(*(-s))) : fsec(n)
with {
  fsec(1) = _, (_<:mem*(1-s),_) : sumandtaps(n);
  fsec(n) = _, (_<:mem*(1-s),_), par(i,n-1,_) : sumandtaps(n);
  s = take(n,sv);
  sumandtaps(n) = +,par(i,n,_);
};


//------------------------------ iir_lat1, allpassn1mt -------------------------
iir_lat1(bv,av) = allpassn1mt(M,sv) : sum(i,M+1,*(tghr(i+1)))
with {
  M = count(av);
  sv = av2sv(av); // sv = vector of sin(theta) reflection coefficients
  tg = bvav2nuv(bv,av); // tg = vector of tap gains
  tgr(i) = take(M+2-i,tg); // i=1..M+1 (for "takability")
  tghr(n)=tgr(n)/pi(n);
  pi(1) = 1;
  pi(n) = pi(n-1)*(1+take(M-n+2,sv)); // all sign parameters '+'
};

// one-multiply lattice allpass with tap lines:
allpassn1mt(0,sv) = _;
allpassn1mt(n,sv)= _ <: _,_ : ((+:*(s) <: _,_),_ : _,+ : crossnn(1)
         : allpassn1mt(n-1,sv),_)~(*(-1)) : fsec(n)
```

```
with {
//0: fsec(n) = _',_ : +
  fsec(1) = crossnn(1) : _, (_<:mem,_) : +,_;
  fsec(n) = crossn1(n) : _, (_<:mem,_),par(i,n-1,_) : +, par(i,n,_);
  innertaps(n) = par(i,n,_);
  s = take(n,sv); // reflection coefficient s = sin(theta)
};

//------------------------------ iir_nl, allpassnnlt -------------------------
// Normalized ladder filter
//
// REFERENCES:
//   J. D. Markel and A. H. Gray, Linear Prediction of Speech, New York: Springer Verlag, 1976.
//   https://ccrma.stanford.edu/~jos/pasp/Normalized_Scattering_Junctions.html

iir_nl(bv,av) = allpassnnlt(M,sv) : sum(i,M+1,*(tghr(i)))
with {
  M = count(av);
  sv = av2sv(av); // sv = vector of sin(theta) reflection coefficients
  tg = bvav2nuv(bv,av); // tg = vector of tap gains for 2mul case
  tgr(i) = take(M+1-i,tg);
  tghr(n) = tgr(n)/pi(n);
  pi(0) = 1;
  s(n) = take(M-n+1,sv); // reflection coefficient = sin(theta)
  c(n) = sqrt(max(0,1-s(n)*s(n))); // compiler crashes on sqrt(-)
  pi(n) = pi(n-1)*c(n);
};

// Normalized ladder allpass with tap lines:
allpassnnlt(0,sv) = _;
allpassnnlt(n,scl*(sv)) = allpassnnlt(n,par(i,count(sv),scl*(sv(i))));
allpassnnlt(n,sv) = _ <: *(s),(*(c) : (+
                : allpassnnlt(n-1,sv))~(*(-s))) : fsec(n)
with {
  fsec(1) = _, (_<:mem*(c),_) : sumandtaps(n);
  fsec(n) = _, (_<:mem*(c),_), par(i,n-1,_) : sumandtaps(n);
  s = take(n,sv);
  c = sqrt(max(0,1-s*s));
  sumandtaps(n) = +,par(i,n,_);
};

//======================== Useful special cases ============================

//------------------------------- tf2np ------------------------------------
// tf2np - biquad based on a stable second-order Normalized Ladder Filter
// (more robust to modulation than tf2 and protected against instability)
tf2np(b0,b1,b2,a1,a2) = allpassnnlt(M,sv) : sum(i,M+1,*(tghr(i)))
with {
  smax = 0.9999; // maximum reflection-coefficient magnitude allowed
  s2 = max(-smax, min(smax,a2)); // Project both reflection-coefficients
  s1 = max(-smax, min(smax,a1/(1+a2))); // into the defined stability-region.
  sv = (s1,s2); // vector of sin(theta) reflection coefficients
  M = 2;
  nu(2) = b2;
  nu(1) = b1 - b2*a1;
  nu(0) = (b0-b2*a2) - nu(1)*s1;
  tg = (nu(0),nu(1),nu(2));
  tgr(i) = take(M+1-i,tg); // vector of tap gains for 2mul case
  tghr(n) = tgr(n)/pi(n); // apply pi parameters for NLF case
  pi(0) = 1;
  s(n) = take(M-n+1,sv);
  c(n) = sqrt(1-s(n)*s(n));
  pi(n) = pi(n-1)*c(n);
};

//------------------------------ wgr ---------------------------------
// Second-order transformer-normalized digital waveguide resonator
```

```
// USAGE:
//   _ : wgr(f,r) : _
// where
//   f = resonance frequency (Hz)
//   r = loss factor for exponential decay
//       (set to 1 to make a numerically stable oscillator)
//
// REFERENCES:
//  https://ccrma.stanford.edu/~jos/pasp/Power_Normalized_Waveguide_Filters.html
//  https://ccrma.stanford.edu/~jos/pasp/Digital_Waveguide_Oscillator.html
//
wgr(f,r,x) = (*(G),_<:_,((+:*(C))<:_,_),_:+,_,_:+(x),-) ~ cross : _,*(0-gi)
with {
  C = cos(2*PI*f/SR);
  gi = sqrt(max(0,(1+C)/(1-C))); // compensate amplitude (only needed when
  G = r*(1-1' + gi')/gi;      //  frequency changes substantially)
  cross = _,_ <: !,_,_,!;
};

//------------------------------ nlf2 --------------------------------
// Second order normalized digital waveguide resonator
// USAGE:
//   _ : nlf2(f,r) : _
// where
//   f = resonance frequency (Hz)
//   r = loss factor for exponential decay
//       (set to 1 to make a sinusoidal oscillator)
//
// REFERENCE:
// https://ccrma.stanford.edu/~jos/pasp/Power_Normalized_Waveguide_Filters.html
//
nlf2(f,r,x) = ((_<:_,_),(_<:_,_) : (*(s),*(c),*(c),*(0-s)) :
              (*(r),+(x))) ~ cross
with {
  th = 2*PI*f/SR;
  c = cos(th);
  s = sin(th);
  cross = _,_ <: !,_,_,!;
};

//====================== Ladder/Lattice Allpass Filters ======================
// An allpass filter has gain 1 at every frequency, but variable phase.
// Ladder/lattice allpass filters are specified by reflection coefficients.
// They are defined here as nested allpass filters, hence the names allpassn*.
//
// REFERENCES
//   1. https://ccrma.stanford.edu/~jos/pasp/Conventional_Ladder_Filters.html
//      https://ccrma.stanford.edu/~jos/pasp/Nested_Allpass_Filters.html
//   2. Linear Prediction of Speech, Markel and Gray, Springer Verlag, 1976
//
// QUICK GUIDE
//   allpassn  - two-multiply lattice - each section is two multiply-adds
//   allpassnn - normalized form - four multiplies and two adds per section,
//               but coefficients can be time varying and nonlinear without
//               "parametric amplification" (modulation of signal energy).
//   allpassnkl - Kelly-Lochbaum form - four multiplies and two adds per
//               section, but all signals have an immediate physical
//               interpretation as traveling pressure waves, etc.
//   allpassn1m - One-multiply form - one multiply and three adds per section.
//               Normally the most efficient in special-purpose hardware.
//
// TYPICAL USAGE
//   _ : allpassn(N,sv) : _
// where
//   N  = allpass order (number of ladder or lattice sections)
//   sv = (s1,s2,...,sN) = reflection coefficients (between -1 and 1).
//        For allpassnn only, sv is replaced by tv, where sv(i) = sin(tv(i)),
//        where tv(i) may range between -PI and PI.
```

```
//
// two-multiply:
allpassn(0,sv) = _;
allpassn(n,sv) = _ <: ((+ <: (allpassn(n-1,sv)),*(s))~(*(-s))) : _',_ :+
with { s = take(n,sv); };

// power-normalized (reflection coefficients s = sin(t)):
allpassnn(0,tv) = _;
allpassnn(n,tv) = _ <: *(s), (*(c) : (+
       : allpassnn(n-1,tv))~(*(-s))) : _, mem*c : +
with { c=cos(take(n,tv)); s=sin(take(n,tv)); };

// power-normalized with sparse delays dv(n)>1:
allpassnns(0,tv,dmax,dv) = _;
allpassnns(n,tv,dmax,dv) = _ <: *(s), (*(c) : (+ : dl
         : allpassnns(n-1,tv,dmax,dv))~(*(-s))) : _, mem*c : +
with { c=cos(take(n,tv)); s=sin(take(n,tv));
      dl=delay(dmax,(take(n,dv)-1)); };

// Kelly-Lochbaum:
allpassnkl(0,sv) = _;
allpassnkl(n,sv) = _ <: *(s),(*(1+s) : (+
                 : allpassnkl(n-1,sv))~(*(-s))) : _, mem*(1-s) : +
with { s = take(n,sv); };

// one-multiply:
allpassn1m(0,sv) = _;
allpassn1m(n,sv)= _ <: _,_ : ((+:*(s) <: _,_),_ : _,+ : cross
         : allpassn1m(n-1,sv),_)~(*(-1)) : _',_ : +
with {s = take(n,sv); cross = _,_ <: !,_,_,!; };

//===== Digital Filter Sections Specified as Analog Filter Sections =====
//
//------------------------- tf2s, tf2snp --------------------------------
// Second-order direct-form digital filter,
// specified by ANALOG transfer-function polynomials B(s)/A(s),
// and a frequency-scaling parameter. Digitization via the
// bilinear transform is built in.
//
// USAGE: tf2s(b2,b1,b0,a1,a0,w1), where
//
//        b2 s^2 + b1 s + b0
// H(s) = --------------------
//           s^2 + a1 s + a0
//
// and w1 is the desired digital frequency (in radians/second)
// corresponding to analog frequency 1 rad/sec (i.e., s = j).
//
// EXAMPLE: A second-order ANALOG Butterworth lowpass filter,
//         normalized to have cutoff frequency at 1 rad/sec,
//         has transfer function
//
//             1
// H(s) = -----------------
//        s^2 + a1 s + 1
//
// where a1 = sqrt(2). Therefore, a DIGITAL Butterworth lowpass
// cutting off at SR/4 is specified as tf2s(0,0,1,sqrt(2),1,PI*SR/2);
//
// METHOD: Bilinear transform scaled for exact mapping of w1.
// REFERENCE:
//   https://ccrma.stanford.edu/~jos/pasp/Bilinear_Transformation.html
//
tf2s(b2,b1,b0,a1,a0,w1) = tf2(b0d,b1d,b2d,a1d,a2d)
with {
  c   = 1/tan(w1*0.5/SR); // bilinear-transform scale-factor
  csq = c*c;
  d   = a0 + a1 * c + csq;
```

```
  b0d = (b0 + b1 * c + b2 * csq)/d;
  b1d = 2 * (b0 - b2 * csq)/d;
  b2d = (b0 - b1 * c + b2 * csq)/d;
  a1d = 2 * (a0 - csq)/d;
  a2d = (a0 - a1*c + csq)/d;
};

// tf2snp = tf2s but using a protected normalized ladder filter for tf2:
tf2snp(b2,b1,b0,a1,a0,w1) = tf2np(b0d,b1d,b2d,a1d,a2d)
with {
  c   = 1/tan(w1*0.5/SR); // bilinear-transform scale-factor
  csq = c*c;
  d   = a0 + a1 * c + csq;
  b0d = (b0 + b1 * c + b2 * csq)/d;
  b1d = 2 * (b0 - b2 * csq)/d;
  b2d = (b0 - b1 * c + b2 * csq)/d;
  a1d = 2 * (a0 - csq)/d;
  a2d = (a0 - a1*c + csq)/d;
};

//------------------------------ tf3slf -------------------------------
// Analogous to tf2s above, but third order, and using the typical
// low-frequency-matching bilinear-transform constant 2/T ("lf" series)
// instead of the specific-frequency-matching value used in tf2s and tf1s.
// Note the lack of a "w1" argument.
//
tf3slf(b3,b2,b1,b0,a3,a2,a1,a0) = tf3(b0d,b1d,b2d,b3d,a1d,a2d,a3d) with {
  c   = 2.0 * SR; // bilinear-transform scale-factor ("lf" case)
  csq = c*c;
  cc = csq*c;
  // Thank you maxima:
  b3d = (b3*c^3-b2*c^2+b1*c-b0)/d;
  b2d = (-3*b3*c^3+b2*c^2+b1*c-3*b0)/d;
  b1d = (3*b3*c^3+b2*c^2-b1*c-3*b0)/d;
  b0d = (-b3*c^3-b2*c^2-b1*c-b0)/d;
  a3d = (a3*c^3-a2*c^2+a1*c-a0)/d;
  a2d = (-3*a3*c^3+a2*c^2+a1*c-3*a0)/d;
  a1d = (3*a3*c^3+a2*c^2-a1*c-3*a0)/d;
  d = (-a3*c^3-a2*c^2-a1*c-a0);
};

//------------------------------ tf1s --------------------------------
// First-order direct-form digital filter,
// specified by ANALOG transfer-function polynomials B(s)/A(s),
// and a frequency-scaling parameter.
//
// USAGE: tf1s(b1,b0,a0,w1), where
//
//       b1 s + b0
// H(s) = ----------
//          s + a0
//
// and w1 is the desired digital frequency (in radians/second)
// corresponding to analog frequency 1 rad/sec (i.e., s = j).
//
// EXAMPLE: A first-order ANALOG Butterworth lowpass filter,
//         normalized to have cutoff frequency at 1 rad/sec,
//         has transfer function
//
//          1
// H(s) = -------
//        s + 1
//
// so b0 = a0 = 1 and b1 = 0. Therefore, a DIGITAL first-order
// Butterworth lowpass with gain -3dB at SR/4 is specified as
//
// tf1s(0,1,1,PI*SR/2); // digital half-band order 1 Butterworth
//
```

```
// METHOD: Bilinear transform scaled for exact mapping of w1.
// REFERENCE:
//   https://ccrma.stanford.edu/~jos/pasp/Bilinear_Transformation.html
//
tf1s(b1,b0,a0,w1) = tf1(b0d,b1d,a1d)
with {
  c   = 1/tan(w1*0.5/SR); // bilinear-transform scale-factor
  d   = a0 + c;
  b1d = (b0 - b1*c) / d;
  b0d = (b0 + b1*c) / d;
  a1d = (a0 - c) / d;
};

//------------------------------ tf2sb --------------------------------
// Bandpass mapping of tf2s: In addition to a frequency-scaling parameter
// w1 (set to HALF the desired passband width in rad/sec),
// 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 prototpe
// 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. Algebra carried out in maxima and pasted here.
//
tf2sb(b2,b1,b0,a1,a0,w1,wc) =
  iir((b0d/a0d,b1d/a0d,b2d/a0d,b3d/a0d,b4d/a0d),(a1d/a0d,a2d/a0d,a3d/a0d,a4d/a0d)) with {
  T = 1.0/float(SR);
  b0d = (4*b0*w1^2+8*b2*wc^2)*T^2+8*b1*w1*T+16*b2;
  b1d = 4*b2*wc^4*T^4+4*b1*wc^2*w1*T^3-16*b1*w1*T-64*b2;
  b2d = 6*b2*wc^4*T^4+(-8*b0*w1^2-16*b2*wc^2)*T^2+96*b2;
  b3d = 4*b2*wc^4*T^4-4*b1*wc^2*w1*T^3+16*b1*w1*T-64*b2;
  b4d = (b2*wc^4*T^4-2*b1*wc^2*w1*T^3+(4*b0*w1^2+8*b2*wc^2)*T^2-8*b1*w1*T +16*b2)
        + b2*wc^4*T^4+2*b1*wc^2*w1*T^3;
  a0d = wc^4*T^4+2*a1*wc^2*w1*T^3+(4*a0*w1^2+8*wc^2)*T^2+8*a1*w1*T+16;
  a1d = 4*wc^4*T^4+4*a1*wc^2*w1*T^3-16*a1*w1*T-64;
  a2d = 6*wc^4*T^4+(-8*a0*w1^2-16*wc^2)*T^2+96;
  a3d = 4*wc^4*T^4-4*a1*wc^2*w1*T^3+16*a1*w1*T-64;
  a4d = wc^4*T^4-2*a1*wc^2*w1*T^3+(4*a0*w1^2+8*wc^2)*T^2-8*a1*w1*T+16;
};

//------------------------------ tf1sb --------------------------------
// First-to-second-order lowpass-to-bandpass section mapping,
// analogous to tf2sb above.
//
tf1sb(b1,b0,a0,w1,wc) = tf2(b0d/a0d,b1d/a0d,b2d/a0d,a1d/a0d,a2d/a0d) with {
  T = 1.0/float(SR);
  a0d = wc^2*T^2+2*a0*w1*T+4;
  b0d = b1*wc^2*T^2 +2*b0*w1*T+4*b1;
  b1d = 2*b1*wc^2*T^2-8*b1;
  b2d = b1*wc^2*T^2-2*b0*w1*T+4*b1;
  a1d = 2*wc^2*T^2-8;
  a2d = wc^2*T^2-2*a0*w1*T+4;
};

//====================== Simple Resonator Filters ======================

// resonlp = 2nd-order lowpass with corner resonance:
resonlp(fc,Q,gain) = tf2s(b2,b1,b0,a1,a0,wc)
with {
    wc = 2*PI*fc;
    a1 = 1/Q;
    a0 = 1;
    b2 = 0;
    b1 = 0;
    b0 = gain;
};

// resonhp = 2nd-order highpass with corner resonance:
```

```
771 resonhp(fc,Q,gain,x) = gain*x-resonlp(fc,Q,gain,x);
772
773 // resonbp = 2nd-order bandpass
774 resonbp(fc,Q,gain) = tf2s(b2,b1,b0,a1,a0,wc)
775 with {
776     wc = 2*PI*fc;
777     a1 = 1/Q;
778     a0 = 1;
779     b2 = 0;
780     b1 = gain;
781     b0 = 0;
782 };
783
784 //================ Butterworth Lowpass/Highpass Filters ======================
785 // Nth-order Butterworth lowpass or highpass filters
786 //
787 // USAGE:
788 //   _ : lowpass(N,fc) : _
789 //   _ : highpass(N,fc) : _
790 // where
791 //   N = filter order (number of poles) [nonnegative constant integer]
792 //   fc = desired cut-off frequency (-3dB frequency) in Hz
793 // REFERENCE:
794 //  https://ccrma.stanford.edu/~jos/filters/Butterworth_Lowpass_Design.html
795 //  'butter' function in Octave ("[z,p,g] = butter(N,1,'s');")
796 // ACKNOWLEDGMENT
797 //  Generalized recursive formulation initiated by Yann Orlarey.
798
799 lowpass(N,fc) = lowpass0_highpass1(0,N,fc);
800 highpass(N,fc) = lowpass0_highpass1(1,N,fc);
801 lowpass0_highpass1(s,N,fc) = lphpr(s,N,N,fc)
802 with {
803   lphpr(s,0,N,fc) = _;
804   lphpr(s,1,N,fc) = tf1s(s,1-s,1,2*PI*fc);
805   lphpr(s,O,N,fc) = lphpr(s,(O-2),N,fc) : tf2s(s,0,1-s,a1s,1,w1) with {
806     parity = N % 2;
807     S = (O-parity)/2; // current section number
808     a1s = -2*cos(-PI + (1-parity)*PI/(2*N) + (S-1+parity)*PI/N);
809     w1 = 2*PI*fc;
810   };
811 };
812
813 //========== Special Filter-Bank Delay-Equalizing Allpass Filters ===========
814 //
815 // These special allpass filters are needed by filterbank et al. below.
816 // They are equivalent to (lowpass(N,fc) +|- highpass(N,fc))/2, but with
817 // canceling pole-zero pairs removed (which occurs for odd N).
818
819 //-------------------- lowpass_plus|minus_highpass ------------------
820
821 highpass_plus_lowpass(1,fc) = _;
822 highpass_plus_lowpass(3,fc) = tf2s(1,-1,1,1,1,w1) with { w1 = 2*PI*fc; };
823 highpass_minus_lowpass(3,fc) = tf1s(-1,1,1,w1) with { w1 = 2*PI*fc; };
824 highpass_plus_lowpass(5,fc) = tf2s(1,-a11,1,a11,1,w1)
825 with {
826   a11 = 1.618033988749895;
827   w1 = 2*PI*fc;
828 };
829 highpass_minus_lowpass(5,fc) = tf1s(1,-1,1,w1) : tf2s(1,-a12,1,a12,1,w1)
830 with {
831   a12 = 0.618033988749895;
832   w1 = 2*PI*fc;
833 };
834
835 // Catch-all definitions for generality - even order is done:
836
837 highpass_plus_lowpass(N,fc) = switch_odd_even(N%2,N,fc) with {
838   switch_odd_even(0,N,fc) = highpass_plus_lowpass_even(N,fc);
```

```
  switch_odd_even(1,N,fc) = highpass_plus_lowpass_odd(N,fc);
};

highpass_minus_lowpass(N,fc) = switch_odd_even(N%2,N,fc) with {
  switch_odd_even(0,N,fc) = highpass_minus_lowpass_even(N,fc);
  switch_odd_even(1,N,fc) = highpass_minus_lowpass_odd(N,fc);
};

highpass_plus_lowpass_even(N,fc) = highpass(N,fc) + lowpass(N,fc);
highpass_minus_lowpass_even(N,fc) = highpass(N,fc) - lowpass(N,fc);

// FIXME: Rewrite the following, as for orders 3 and 5 above,
//       to eliminate pole-zero cancellations:
highpass_plus_lowpass_odd(N,fc) = highpass(N,fc) + lowpass(N,fc);
highpass_minus_lowpass_odd(N,fc) = highpass(N,fc) - lowpass(N,fc);

//==================== Elliptic (Cauer) Lowpass Filters ==================
// USAGE:
//   _ : lowpass3e(fc) : _
//   _ : lowpass6e(fc) : _
// where fc = -3dB frequency in Hz
//
// REFERENCES:
//   http://en.wikipedia.org/wiki/Elliptic_filter
//   functions 'ncauer' and 'ellip' in Octave

//------------------------------ lowpass3e -----------------------------
// Third-order Elliptic (Cauer) lowpass filter
// DESIGN: For spectral band-slice level display (see octave_analyzer3e):
//   [z,p,g] = ncauer(Rp,Rs,3); % analog zeros, poles, and gain, where
//   Rp = 60 % dB ripple in stopband
//   Rs = 0.2 % dB ripple in passband
//
lowpass3e(fc) = tf2s(b21,b11,b01,a11,a01,w1) : tf1s(0,1,a02,w1)
with {
  a11 = 0.802636764161030; // format long; poly(p(1:2)) % in octave
  a01 = 1.412270893774204;
  a02 = 0.822445908998816; // poly(p(3)) % in octave
  b21 = 0.019809144837789; // poly(z)
  b11 = 0;
  b01 = 1.161516418982696;
  w1 = 2*PI*fc;
};

//------------------------------ lowpass6e -----------------------------
// Sixth-order Elliptic/Cauer lowpass filter
// DESIGN: For spectral band-slice level display (see octave_analyzer6e):
//   [z,p,g] = ncauer(Rp,Rs,6); % analog zeros, poles, and gain, where
//   Rp = 80 % dB ripple in stopband
//   Rs = 0.2 % dB ripple in passband
//
lowpass6e(fc) =
            tf2s(b21,b11,b01,a11,a01,w1) :
            tf2s(b22,b12,b02,a12,a02,w1) :
            tf2s(b23,b13,b03,a13,a03,w1)
with {
  b21 = 0.000099999997055;
  a21 = 1;
  b11 = 0;
  a11 = 0.782413046821645;
  b01 = 0.000433227200555;
  a01 = 0.245291508706160;
  b22 = 1;
  a22 = 1;
  b12 = 0;
  a12 = 0.512478641889141;
  b02 = 7.621731298870603;
  a02 = 0.689621364484675;
```

```
  b23 = 1;
  a23 = 1;
  b13 = 0;
  a13 = 0.168404871113589;
  b03 = 53.536152954556727;
  a03 = 1.069358407707312;
  w1 = 2*PI*fc;
};

//====================== Elliptic Highpass Filters =====================
// USAGE:
//   _ : highpass3e(fc) : _
//   _ : highpass6e(fc) : _
// where fc = -3dB frequency in Hz

//------------------------------ highpass3e -----------------------------
// Third-order Elliptic (Cauer) highpass filter
// DESIGN: Inversion of lowpass3e wrt unit circle in s plane (s <- 1/s)
//
highpass3e(fc) = tf2s(b01/a01,b11/a01,b21/a01,a11/a01,1/a01,w1) :
                 tf1s(1/a02,0,1/a02,w1)
with {
  a11 = 0.802636764161030;
  a01 = 1.412270893774204;
  a02 = 0.822445908998816;
  b21 = 0.019809144837789;
  b11 = 0;
  b01 = 1.161516418982696;
  w1 = 2*PI*fc;
};

//------------------------------ highpass6e -----------------------------
// Sixth-order Elliptic/Cauer highpass filter
// METHOD: Inversion of lowpass3e wrt unit circle in s plane (s <- 1/s)
//
highpass6e(fc) =
             tf2s(b01/a01,b11/a01,b21/a01,a11/a01,1/a01,w1) :
             tf2s(b02/a02,b12/a02,b22/a02,a12/a02,1/a02,w1) :
             tf2s(b03/a03,b13/a03,b23/a03,a13/a03,1/a03,w1)
with {
  b21 = 0.000099999997055;
  a21 = 1;
  b11 = 0;
  a11 = 0.782413046821645;
  b01 = 0.000433227200555;
  a01 = 0.245291508706160;
  b22 = 1;
  a22 = 1;
  b12 = 0;
  a12 = 0.512478641889141;
  b02 = 7.621731298870603;
  a02 = 0.689621364484675;
  b23 = 1;
  a23 = 1;
  b13 = 0;
  a13 = 0.168404871113589;
  b03 = 53.536152954556727;
  a03 = 1.069358407707312;
  w1 = 2*PI*fc;
};

//================== Butterworth Bandpass/Bandstop Filters ======================
// Order 2*Nh Butterworth bandpass filter made using the transformation
// s <- s + wc^2/s on lowpass(Nh), where wc is the desired bandpass center
// frequency. The lowpass(Nh) cutoff w1 is half the desired bandpass width.
// A notch-like "bandstop" filter is similarly made from highpass(Nh).
//
// USAGE:
```

```
//   _ : bandpass(Nh,fl,fu) : _
//   _ : bandstop(Nh,fl,fu) : _
// where
//   Nh = HALF the desired bandpass/bandstop order (which is therefore even)
//   fl = lower -3dB frequency in Hz
//   fu = upper -3dB frequency in Hz
// Thus, the passband (stopband) width is fu-fl,
//       and its center frequency is (fl+fu)/2.
//
// REFERENCE:
//   http://cnx.org/content/m16913/latest/
//
bandpass(Nh,fl,fu) = bandpass0_bandstop1(0,Nh,fl,fu);
bandstop(Nh,fl,fu) = bandpass0_bandstop1(1,Nh,fl,fu);
bandpass0_bandstop1(s,Nh,fl,fu) = bpbsr(s,Nh,Nh,fl,fu)
with {
  wl = 2*PI*fl; // digital (z-plane) lower passband edge
  wu = 2*PI*fu; // digital (z-plane) upper passband edge

  c = 2.0*SR; // bilinear transform scaling used in tf2sb, tf1sb
  wla = c*tan(wl/c); // analog (s-splane) lower cutoff
  wua = c*tan(wu/c); // analog (s-splane) upper cutoff

  wc = sqrt(wla*wua); // s-plane center frequency
  w1 = wua - wc^2/wua; // s-plane lowpass prototype cutoff

  bpbsr(s,0,Nh,fl,fu) = _;
  bpbsr(s,1,Nh,fl,fu) = tf1sb(s,1-s,1,w1,wc);
  bpbsr(s,0,Nh,fl,fu) = bpbsr(s,0-2,Nh,fl,fu) : tf2sb(s,0,(1-s),a1s,1,w1,wc)
  with {
    parity = Nh % 2;
    S = (0-parity)/2; // current section number
    a1s = -2*cos(-PI + (1-parity)*PI/(2*Nh) + (S-1+parity)*PI/Nh);
  };
};

//====================== Elliptic Bandpass Filters ============================

//------------------------------ bandpass6e ----------------------------
// Order 12 elliptic bandpass filter analogous to bandpass(6) above.
//
bandpass6e(fl,fu) = tf2sb(b21,b11,b01,a11,a01,w1,wc) : tf1sb(0,1,a02,w1,wc)
with {
  a11 = 0.802636764161030; // In octave: format long; poly(p(1:2))
  a01 = 1.412270893774204;
  a02 = 0.822445908998816; // poly(p(3))
  b21 = 0.019809144837789; // poly(z)
  b11 = 0;
  b01 = 1.161516418982696;

  wl = 2*PI*fl; // digital (z-plane) lower passband edge
  wu = 2*PI*fu; // digital (z-plane) upper passband edge

  c = 2.0*SR; // bilinear transform scaling used in tf2sb, tf1sb
  wla = c*tan(wl/c); // analog (s-splane) lower cutoff
  wua = c*tan(wu/c); // analog (s-splane) upper cutoff

  wc = sqrt(wla*wua); // s-plane center frequency
  w1 = wua - wc^2/wua; // s-plane lowpass cutoff
};

//------------------------------ bandpass12e -----------------------------

bandpass12e(fl,fu) =
            tf2sb(b21,b11,b01,a11,a01,w1,wc) :
            tf2sb(b22,b12,b02,a12,a02,w1,wc) :
            tf2sb(b23,b13,b03,a13,a03,w1,wc)
with { // octave script output:
```

```
  b21 = 0.000099999997055;
  a21 = 1;
  b11 = 0;
  a11 = 0.782413046821645;
  b01 = 0.000433227200555;
  a01 = 0.245291508706160;
  b22 = 1;
  a22 = 1;
  b12 = 0;
  a12 = 0.512478641889141;
  b02 = 7.621731298870603;
  a02 = 0.689621364484675;
  b23 = 1;
  a23 = 1;
  b13 = 0;
  a13 = 0.168404871113589;
  b03 = 53.536152954556727;
  a03 = 1.069358407707312;

  wl = 2*PI*fl; // digital (z-plane) lower passband edge
  wu = 2*PI*fu; // digital (z-plane) upper passband edge

  c = 2.0*SR; // bilinear transform scaling used in tf2sb, tf1sb
  wla = c*tan(wl/c); // analog (s-splane) lower cutoff
  wua = c*tan(wu/c); // analog (s-splane) upper cutoff

  wc = sqrt(wla*wua); // s-plane center frequency
  w1 = wua - wc^2/wua; // s-plane lowpass cutoff
};

//================= Parametric Equalizers (Shelf, Peaking) ==================
// REFERENCES
// - http://en.wikipedia.org/wiki/Equalization
// - filterbank (below, here in filter.lib)
// - http://www.musicdsp.org/files/Audio-EQ-Cookbook.txt
// - Digital Audio Signal Processing, Udo Zolzer, Wiley, 1999, p. 124
// - https://ccrma.stanford.edu/~jos/filters/Low_High_Shelving_Filters.html
// - https://ccrma.stanford.edu/~jos/filters/Peaking_Equalizers.html
// - maxmsp.lib in the Faust distribution
// - bandfilter.dsp in the faust2pd distribution

//------------------------------ low_shelf -----------------------------------
// First-order "low shelf" filter (gain boost|cut between dc and some frequency)
//
// USAGE:
//   _ : lowshelf(N,L0,fx) : _
// where
//   N = filter order 1, 3, 5, ... (odd only).
//  L0 = desired level (dB) between dc and fx (boost L0>0 or cut L0<0)
//  fx = -3dB frequency of lowpass band (L0>0) or upper band (L0<0)
//       (see "SHELF SHAPE" below).
//
// The gain at SR/2 is constrained to be 1.
// The generalization to arbitrary odd orders is based on the well known
// fact that odd-order Butterworth band-splits are allpass-complementary
// (see filterbank documentation below for references).
//
// SHELF SHAPE
//  The magnitude frequency response is approximately piecewise-linear
//  on a log-log plot ("BODE PLOT"). The Bode "stick diagram" approximation
//  L(lf) is easy to state in dB versus dB-frequency lf = dB(f):
// L0 > 0:
//  L(lf) = L0, f between 0 and fx = 1st corner frequency;
//  L(lf) = L0 - N * (lf - lfx), f between fx and f2 = 2nd corner frequency;
//  L(lf) = 0, lf > lf2.
//    lf2 = lfx + L0/N = dB-frequency at which level gets back to 0 dB.
//    See lowshelf_other_freq() below.
// L0 < 0:
```

```
// L(lf) = L0, f between 0 and f1 = 1st corner frequency;
// L(lf) = - N * (lfx - lf), f between f1 and lfx = 2nd corner frequency;
// L(lf) = 0, lf > lfx.
//   lf1 = lfx + L0/N = dB-frequency at which level goes up from L0.
//   See lowshelf_other_freq() below.

lowshelf(N,L0,fx) = filterbank(N,(fx)) : _, *(db2linear(L0)) :> _;
// Special cases and optimization:
low_shelf = lowshelf(3); // default = 3rd order Butterworth
low_shelf1(L0,fx,x) = x + (db2linear(L0)-1)*lowpass(1,fx,x); // optimized
low_shelf1_l(G0,fx,x) = x + (G0-1)*lowpass(1,fx,x); // optimized

lowshelf_other_freq(N, L0, fx) = db2linear(linear2db(fx) + L0/N); // convenience

//------------------------------ high_shelf ------------------------------------
// First-order "high shelf" filter (gain boost|cut above some frequency)
//
// USAGE:
//   _ : highshelf(N,Lpi,fx) : _
// where
//   N = filter order 1, 3, 5, ... (odd only).
//  Lpi = desired level (dB) between fx and SR/2 (boost Lpi>0 or cut Lpi<0)
//   fx = -3dB frequency of highpass band (L0>0) or lower band (L0<0)
//       (Use highshelf_other_freq() below to find the other one.)
//
// The gain at dc is constrained to be 1.
// See lowshelf documentation above regarding SHELF SHAPE.

highshelf(N,Lpi,fx) = filterbank(N,(fx)) : *(db2linear(Lpi)), _ :> _;
// Special cases and optimization:
high_shelf = highshelf(3); // default = 3rd order Butterworth
high_shelf1(Lpi,fx,x) = x + (db2linear(Lpi)-1)*highpass(1,fx,x); // optimized
high_shelf1_l(Gpi,fx,x) = x + (Gpi-1)*highpass(1,fx,x); //optimized

// shelf transitions between frequency fx and this one:
highshelf_other_freq(N, Lpi, fx) = db2linear(linear2db(fx) - Lpi/N);

//-------------------------------- peak_eq -------------------------------------
// Second order "peaking equalizer" section
// (gain boost or cut near some frequency)
//   Also called a "parametric equalizer" section
// USAGE: _ : peak_eq(Lfx,fx,B) : _;
// where
//   Lfx = level (dB) at fx (boost Lfx>0 or cut Lfx<0)
//    fx = peak frequency (Hz)
//     B = bandwidth (B) of peak in Hz
//
peak_eq(Lfx,fx,B) = tf2s(1,b1s,1,a1s,1,wx) with {
  T = float(1.0/SR);
  Bw = B*T/sin(wx*T); // prewarp s-bandwidth for more accuracy in z-plane
  a1 = PI*Bw;
  b1 = g*a1;
  g = db2linear(abs(Lfx));
  b1s = select2(Lfx>0,a1,b1); // When Lfx>0, pole dominates bandwidth
  a1s = select2(Lfx>0,b1,a1); // When Lfx<0, zero dominates
  wx = 2*PI*fx;
};

//------------------------------ peak_eq_cq --------------------------------
// Constant-Q second order peaking equalizer section
// USAGE: _ : peak_eq_cq(Lfx,fx,Q) : _;
// where
//   Lfx = level (dB) at fx
//    fx = boost or cut frequency (Hz)
//     Q = "Quality factor" = fx/B where B = bandwidth of peak in Hz
//
peak_eq_cq(Lfx,fx,Q) = peak_eq(Lfx,fx,fx/Q);
```

```
//------------------------------- peak_eq_rm ---------------------------------
// Regalia-Mitra second order peaking equalizer section
// USAGE: _ : peak_eq_rm(Lfx,fx,tanPiBT) : _;
// where
//    Lfx = level (dB) at fx
//     fx = boost or cut frequency (Hz)
// tanPiBT = tan(PI*B/SR), where B = -3dB bandwidth (Hz) when 10^(Lfx/20) = 0
//        ~ PI*B/SR for narrow bandwidths B
//
// REFERENCE:
//   P.A. Regalia, S.K. Mitra, and P.P. Vaidyanathan,
//   "The Digital All-Pass Filter: A Versatile Signal Processing Building Block"
//   Proceedings of the IEEE, 76(1):19-37, Jan. 1988. (See pp. 29-30.)
//
peak_eq_rm(Lfx,fx,tanPiBT) = _ <: _,A,_ : +,- : *(0.5),*(K/2.0) : + with {
  A = tf2(k2, k1*(1+k2), 1, k1*(1+k2), k2) <: _,_; // allpass
  k1 = 0.0 - cos(2.0*PI*fx/SR);
  k2 = (1.0 - tanPiBT)/(1.0 + tanPiBT);
  K = db2linear(Lfx);
};

//-------------------------- parametric_eq_demo -----------------------------
// USAGE:
//   _ : parametric_eq_demo: _ ;
parametric_eq_demo = // input_signal :
    low_shelf(LL,FL) :
        peak_eq(LP,FP,BP) :
    high_shelf(LH,FH)
// Recommended:
//      : mth_octave_spectral_level_demo(2) // half-octave spectrum analyzer
with {
  eq_group(x) = hgroup("[0] PARAMETRIC EQ SECTIONS
               [tooltip: See Faust's filter.lib for info and pointers]",x);

  ls_group(x) = eq_group(vgroup("[1] Low Shelf",x));
  LL = ls_group(hslider("[0] Low Boost|Cut [unit:dB] [style:knob]
               [tooltip: Amount of low-frequency boost or cut in decibels]",
               0,-40,40,0.1));
  FL = ls_group(hslider("[1] Transition Frequency [unit:Hz] [style:knob] [scale:log]
               [tooltip: Transition-frequency from boost (cut) to unity gain]",
               200,1,5000,1));

  pq_group(x) = eq_group(vgroup("[2] Peaking Equalizer
               [tooltip: Parametric Equalizer sections from filter.lib]",x));
  LP = pq_group(hslider("[0] Peak Boost|Cut [unit:dB] [style:knob]
               [tooltip: Amount of local boost or cut in decibels]",
               0,-40,40,0.1));
  FP = pq_group(hslider("[1] Peak Frequency [unit:PK] [style:knob]
       [tooltip: Peak Frequency in Piano Key (PK) units (A440 = 49PK)]",
               49,1,100,1)) : smooth(0.999) : pianokey2hz
        with { pianokey2hz(x) = 440.0*pow(2.0, (x-49.0)/12); };

  Q = pq_group(hslider("[2] Peak Q [style:knob] [scale:log]
             [tooltip: Quality factor (Q) of the peak = center-frequency/bandwidth]",
             40,1,1000,0.1));

  BP = FP/Q;

  hs_group(x) = eq_group(vgroup("[3] High Shelf
               [tooltip: A high shelf provides a boost or cut
                        above some frequency]",x));
  LH = hs_group(hslider("[0] High Boost|Cut [unit:dB] [style:knob]
               [tooltip: Amount of high-frequency boost or cut in decibels]",
               0,-40,40,.1));
  FH = hs_group(hslider("[1] Transition Frequency [unit:Hz] [style:knob] [scale:log]
               [tooltip: Transition-frequency from boost (cut) to unity gain]",
               8000,20,10000,1));
};
```

```
1247
1248 //------------------------- spectral_tilt -------------------------------
1249 // Spectral tilt filter, providing an arbitrary spectral rolloff factor
1250 // alpha in (-1,1), where
1251 //  -1 corresponds to one pole (-6 dB per octave), and
1252 //  +1 corresponds to one zero (+6 dB per octave).
1253 // In other words, alpha is the slope of the ln magnitude versus ln frequency.
1254 // For a "pinking filter" (e.g., to generate 1/f noise from white noise),
1255 // set alpha to -1/2.
1256 //
1257 // USAGE:
1258 //   _ : spectral_tilt(N,f0,bw,alpha) : _
1259 // where
1260 //     N = desired integer filter order (fixed at compile time)
1261 //    f0 = lower frequency limit for desired roll-off band
1262 //    bw = bandwidth of desired roll-off band
1263 // alpha = slope of roll-off desired in nepers per neper
1264 //         (ln mag / ln radian freq)
1265 //
1266 // EXAMPLE:
1267 //    spectral_tilt_demo below
1268 //
1269 // REFERENCE:
1270 //    http://arxiv.org/abs/1606.06154
1271 //
1272 spectral_tilt(N,f0,bw,alpha) = seq(i,N,sec(i)) with {
1273   sec(i) = g * tf1s(b1,b0,a0,1) with {
1274     g = a0/b0; // unity dc-gain scaling
1275     b1 = 1.0;
1276     b0 = mzh(i);
1277     a0 = mph(i);
1278     mzh(i) = prewarp(mz(i),SR,w0); // prewarping for bilinear transform
1279     mph(i) = prewarp(mp(i),SR,w0);
1280     prewarp(w,SR,wp) = wp * tan(w*T/2) / tan(wp*T/2) with { T = 1/SR; };
1281     mz(i) = w0 * r ^ (-alpha+i); // minus zero i in s plane
1282     mp(i) = w0 * r ^ i; // minus pole i in s plane
1283     w0 = 2 * PI * f0; // radian frequency of first pole
1284     f1 = f0 + bw; // upper band limit
1285     r = (f1/f0)^(1.0/float(N-1)); // pole ratio (2 => octave spacing)
1286   };
1287 };
1288
1289 //------------------------- spectral_tilt_demo ------------------------------
1290 // USAGE:
1291 //   _ : spectral_tilt_demo(N) : _ ;
1292 // where
1293 //   N = filter order (integer)
1294 // All other parameters interactive
1295 //
1296 spectral_tilt_demo(N) = spectral_tilt(N,f0,bw,alpha) with {
1297     alpha = hslider("[1] Slope of Spectral Tilt across Band",-1/2,-1,1,0.001);
1298     f0 = hslider("[2] Band Start Frequency [unit:Hz]",100,20,10000,1);
1299     bw = hslider("[3] Band Width [unit:Hz]",5000,100,10000,1);
1300 };
1301
1302 //======================== Lagrange Interpolation ========================
1303
1304 //------------------------- fdelaylti, fdelayltv -------------------------
1305 // Fractional delay line using Lagrange interpolation
1306 // USAGE: _ : fdelaylt[i|v](order, maxdelay, delay, inputsignal) : _
1307 // where order=1,2,3,... is the order of the Lagrange interpolation polynomial.
1308 // fdelaylti is most efficient, but designed for constant/slowly-varying delay.
1309 // fdelayltv is more expensive and more robust when the delay varies rapidly.
1310 //
1311 // NOTE: The requested delay should not be less than (N-1)/2.
1312 //
1313 // The first-order case (linear interpolation) is equivalent to
1314 // fdelay in music.lib (delay d in [0,1])
```

```
//
// REFERENCES:
//
// https://ccrma.stanford.edu/~jos/pasp/Lagrange_Interpolation.html
//
// Timo I. Laakso et al., "Splitting the Unit Delay - Tools for Fractional
//        Delay Filter Design", IEEE Signal Processing Magazine,
//        vol. 13, no. 1, pp. 30-60, Jan 1996.
//
// Philippe Depalle and Stephan Tassart, "Fractional Delay Lines using
//        Lagrange Interpolators", ICMC Proceedings, pp. 341-343, 1996.

fdelaylti(N,n,d,x) = delay(n,id,x) <: seq(i,N,section(i)) : !,_
with {
  o = (N-1.00001)/2; // offset to ~center FIR interpolator
  dmo = d - o; // assumed nonnegative [d > (N-1)/2]
  id = int(dmo);
  fd = o + frac(dmo);
  section(i,x,y) = (x-x') * c(i) <: _,+(y);
  c(i) = (i - fd)/(i+1);
};

fdelayltv(N,n,d,x) = sum(i, N+1, delay(n,id+i,x) * h(N,fd,i))
with {
  o = (N-1.00001)/2; // ~center FIR interpolator
  dmo = d - o; // assumed >=0 [d > (N-1)/2]
  id = int(dmo);
  fd = o + frac(dmo);
  h(N,d,n) = facs1(N,d,n) * facs2(N,d,n);
  facs1(N,d,n) = select2(n,1,prod(k,max(1,n),select2(k<n,1,fac(d,n,k))));
  facs2(N,d,n) = select2(n<N,1,prod(l,max(1,N-n),fac(d,n,l+n+1)));
  fac(d,n,k) = (d-k)/((n-k)+(n==k));
  // Explicit formula for Lagrange interpolation coefficients:
  // h_d(n) = \prod_{\stackrel{k=0}{k\neq n}}^N \frac{d-k}{n-k}, n=0:N
};

// Backward compatibility:
fdelay1 = fdelayltv(1);
fdelay2 = fdelayltv(2);
fdelay3 = fdelayltv(3);
fdelay4 = fdelayltv(4);
fdelay5 = fdelayltv(5);

//====================== Thiran Allpass Interpolation ====================
// Reference:
// https://ccrma.stanford.edu/~jos/pasp/Thiran_Allpass_Interpolators.html
//
//---------------- fdelay1a, fdelay2a, fdelay3a, fdelay4a -------------
// Delay lines interpolated using Thiran allpass interpolation
// USAGE: fdelayNa(maxdelay, delay, inputsignal)
//       (exactly like fdelay in music.lib)
// where N=1,2,3, or 4 is the order of the Thiran interpolation filter,
// and the delay argument is at least N - 1/2.
//
// (Move the following and similar notes above to filter-lib-doc.txt?)
//
// NOTE: The interpolated delay should not be less than N - 1/2.
//          (The allpass delay ranges from N - 1/2 to N + 1/2.)
//          This constraint can be alleviated by altering the code,
//          but be aware that allpass filters approach zero delay
//          by means of pole-zero cancellations.
//          The delay range [N-1/2,N+1/2] is not optimal. What is?
//
// NOTE: Delay arguments too small will produce an UNSTABLE allpass!
//
// NOTE: Because allpass interpolation is recursive, it is not as robust
//      as Lagrange interpolation under time-varying conditions.
//      (You may hear clicks when changing the delay rapidly.)
```

```
1383 //
1384 // first-order allpass interpolation, delay d in [0.5,1.5]
1385 fdelay1a(n,d,x) = delay(n,id,x) : tf1(eta,1,eta)
1386 with {
1387   o = 0.49999; // offset to make life easy for allpass
1388   dmo = d - o; // assumed nonnegative
1389   id = int(dmo);
1390   fd = o + frac(dmo);
1391   eta = (1-fd)/(1+fd); // allpass coefficient
1392 };
1393
1394 // second-order allpass delay in [1.5,2.5]
1395 fdelay2a(n,d,x) = delay(n,id,x) : tf2(a2,a1,1,a1,a2)
1396 with {
1397   o = 1.49999;
1398   dmo = d - o; // delay range is [order-1/2, order+1/2]
1399   id = int(dmo);
1400   fd = o + frac(dmo);
1401   a1o2 = (2-fd)/(1+fd); // share some terms (the compiler does this anyway)
1402   a1 = 2*a1o2;
1403   a2 = a1o2*(1-fd)/(2+fd);
1404 };
1405
1406 // third-order allpass delay in [2.5,3.5]
1407 // delay d should be at least 2.5
1408 fdelay3a(n,d,x) = delay(n,id,x) : iir((a3,a2,a1,1),(a1,a2,a3))
1409 with {
1410   o = 2.49999;
1411   dmo = d - o;
1412   id = int(dmo);
1413   fd = o + frac(dmo);
1414   a1o3 = (3-fd)/(1+fd);
1415   a2o3 = a1o3*(2-fd)/(2+fd);
1416   a1 = 3*a1o3;
1417   a2 = 3*a2o3;
1418   a3 = a2o3*(1-fd)/(3+fd);
1419 };
1420
1421 // fourth-order allpass delay in [3.5,4.5]
1422 // delay d should be at least 3.5
1423 fdelay4a(n,d,x) = delay(n,id,x) : iir((a4,a3,a2,a1,1),(a1,a2,a3,a4))
1424 with {
1425   o = 3.49999;
1426   dmo = d - o;
1427   id = int(dmo);
1428   fd = o + frac(dmo);
1429   a1o4 = (4-fd)/(1+fd);
1430   a2o6 = a1o4*(3-fd)/(2+fd);
1431   a3o4 = a2o6*(2-fd)/(3+fd);
1432   a1 = 4*a1o4;
1433   a2 = 6*a2o6;
1434   a3 = 4*a3o4;
1435   a4 = a3o4*(1-fd)/(4+fd);
1436 };
1437
1438 //================ Mth-Octave Filter-Banks and Spectrum-Analyzers ============
1439 // Mth-octave filter-banks and spectrum-analyzers split the input signal into a
1440 // bank of parallel signals, one for each spectral band. The parameters are
1441 //
1442 //   M = number of band-slices per octave (>1)
1443 //   N = total number of bands (>2)
1444 //   ftop = upper bandlimit of the Mth-octave bands (<SR/2)
1445 //
1446 // In addition to the Mth-octave output signals, there is a highpass signal
1447 // containing frequencies from ftop to SR/2, and a "dc band" lowpass signal
1448 // containing frequencies from 0 (dc) up to the start of the Mth-octave bands.
1449 // Thus, the N output signals are
1450 //
```

```
//   highpass(ftop), MthOctaveBands(M,N-2,ftop), dcBand(ftop*2^(-M*(N-1)))
//
// A FILTER-BANK is defined here as a signal bandsplitter having the
// property that summing its output signals gives an allpass-filtered
// version of the filter-bank input signal. A more conventional term for
// this is an "allpass-complementary filter bank". If the allpass filter
// is a pure delay (and possible scaling), the filter bank is said to be
// a "perfect-reconstruction filter bank" (see Vaidyanathan-1993 cited
// below for details). A "graphic equalizer", in which band signals
// are scaled by gains and summed, should be based on a filter bank.
//
// A SPECTRUM-ANALYZER is defined here as any band-split whose bands span
// the relevant spectrum, but whose band-signals do not
// necessarily sum to the original signal, either exactly or to within an
// allpass filtering. Spectrum analyzer outputs are normally at least nearly
// "power complementary", i.e., the power spectra of the individual bands
// sum to the original power spectrum (to within some negligible tolerance).
//
// The filter-banks below are implemented as Butterworth or Elliptic
// spectrum-analyzers followed by delay equalizers that make them
// allpass-complementary.
//
// INCREASING CHANNEL ISOLATION
//   Go to higher filter orders - see Regalia et al. or Vaidyanathan (cited
//   below) regarding the construction of more aggressive recursive
//   filter-banks using elliptic or Chebyshev prototype filters.
//
// REFERENCES
// - "Tree-structured complementary filter banks using all-pass sections",
//   Regalia et al., IEEE Trans. Circuits & Systems, CAS-34:1470-1484, Dec. 1987
// - "Multirate Systems and Filter Banks", P. Vaidyanathan, Prentice-Hall, 1993
// - Elementary filter theory: https://ccrma.stanford.edu/~jos/filters/
//
//------------------------- mth_octave_analyzer ----------------------------
//
// USAGE
//  _ : mth_octave_analyzer(O,M,ftop,N) : par(i,N,_); // Oth-order Butterworth
//  _ : mth_octave_analyzer6e(M,ftop,N) : par(i,N,_); // 6th-order elliptic
//
// where
//   O = order of filter used to split each frequency band into two
//   M = number of band-slices per octave
//   ftop = highest band-split crossover frequency (e.g., 20 kHz)
//   N = total number of bands (including dc and Nyquist)
//
// ACKNOWLEDGMENT
//  Recursive band-splitting formulation improved by Yann Orlarey.

mth_octave_analyzer6e(M,ftop,N) = _ <: bsplit(N-1) with {
  fc(n) = ftop * 2^(float(n-N+1)/float(M)); // -3dB crossover frequencies
  lp(n) = lowpass6e(fc(n)); // 6th-order elliptic - see other choices above
  hp(n) = highpass6e(fc(n)); // (search for lowpass* and highpass*)
  bsplit(0) = _;
  bsplit(i) = hp(i), (lp(i) <: bsplit(i-1));
};

// Butterworth analyzers may be cascaded with allpass
// delay-equalizers to make (allpass-complementary) filter banks:

mth_octave_analyzer(O,M,ftop,N) = _ <: bsplit(N-1) with {
  fc(n) = ftop * 2^(float(n-N+1)/float(M));
  lp(n) = lowpass(O,fc(n)); // Order O Butterworth
  hp(n) = highpass(O,fc(n));
  bsplit(0) = _;
  bsplit(i) = hp(i), (lp(i) <: bsplit(i-1));
};

mth_octave_analyzer3(M,ftop,N) = mth_octave_analyzer(3,M,ftop,N);
```

```
mth_octave_analyzer5(M,ftop,N) = mth_octave_analyzer(5,M,ftop,N);
mth_octave_analyzer_default = mth_octave_analyzer6e; // default analyzer

//----------------------- mth_octave_filterbank -------------------------
// Allpass-complementary filter banks based on Butterworth band-splitting.
// For Butterworth band-splits, the needed delay equalizer is easily found.

mth_octave_filterbank(O,M,ftop,N) =
    mth_octave_analyzer(O,M,ftop,N) :
    delayeq(N) with {
  fc(n) = ftop * 2^(float(n-N+1)/float(M)); // -3dB crossover frequencies
  ap(n) = highpass_plus_lowpass(O,fc(n)); // delay-equalizing allpass
  delayeq(N) = par(i,N-2,apchain(i+1)), _, _;
  apchain(i) = seq(j,N-1-i,ap(j+1));
};

// dc-inverted version. This reduces the delay-equalizer order for odd O.
// Negating the input signal makes the dc band noninverting
//   and all higher bands sign-inverted (if preferred).
mth_octave_filterbank_alt(O,M,ftop,N) =
    mth_octave_analyzer(O,M,ftop,N) : delayeqi(O,N) with {
  fc(n) = ftop * 2^(float(n-N+1)/float(M)); // -3dB crossover frequencies
  ap(n) = highpass_minus_lowpass(O,fc(n)); // half the order of 'plus' case
  delayeqi(N) = par(i,N-2,apchain(i+1)), _, *(-1.0);
  apchain(i) = seq(j,N-1-i,ap(j+1));
};

// Note that even-order cases require complex coefficients.
// See Vaidyanathan 1993 and papers cited there for more info.

mth_octave_filterbank3(M,ftop,N) = mth_octave_filterbank_alt(3,M,ftop,N);
mth_octave_filterbank5(M,ftop,N) = mth_octave_filterbank(5,M,ftop,N);

mth_octave_filterbank_default = mth_octave_filterbank5;

//======================= Mth-Octave Spectral Level ========================
// Spectral Level: Display (in bar graphs) the average signal level in each
//               spectral band.
//
//----------------------- mth_octave_spectral_level -------------------------
// USAGE: _ : mth_octave_spectral_level(M,ftop,NBands,tau,dB_offset);
// where
//    M = bands per octave
//    ftop = lower edge frequency of top band
//    NBands = number of passbands (including highpass and dc bands),
//    tau = spectral display averaging-time (time constant) in seconds,
//    dB_offset = constant dB offset in all band level meters.
//
mth_octave_spectral_level6e(M,ftop,N,tau,dB_offset) = _<:
    _,mth_octave_analyzer6e(M,ftop,N) :
    _,(display:>_):attach with {
  display = par(i,N,dbmeter(i));
  dbmeter(i) = abs : smooth(tau2pole(tau)) : linear2db : +(dB_offset) :
     meter(N-i-1);
    meter(i) = speclevel_group(vbargraph("[%2i] [unit:dB]
     [tooltip: Spectral Band Level in dB]", -50, 10));
  O = int(((N-2)/M)+0.4999);
  speclevel_group(x) = hgroup("[0] CONSTANT-Q SPECTRUM ANALYZER (6E), %N bands spanning LP,
        %O octaves below %ftop Hz, HP
     [tooltip: See Faust's filter.lib for documentation and references]", x);
};

mth_octave_spectral_level_default = mth_octave_spectral_level6e;
spectral_level = mth_octave_spectral_level(2,10000,20); // simple default

//---------------------- mth_octave_spectral_level_demo ----------------------
// Demonstrate mth_octave_spectral_level in a standalone GUI.
//
```

```
// USAGE: _ : mth_octave_spectral_level_demo(BandsPerOctave);

mth_octave_spectral_level_demo(M) =
  mth_octave_spectral_level_default(M,ftop,N,tau,dB_offset)
with {
  ftop = 16000;
  Noct = 10; // number of octaves down from ftop
  // Lowest band-edge is at ftop*2^(-Noct+2) = 62.5 Hz when ftop=16 kHz:
  N = int(Noct*M); // without 'int()', segmentation fault observed for M=1.67
  ctl_group(x) = hgroup("[1] SPECTRUM ANALYZER CONTROLS", x);
  tau = ctl_group(hslider("[0] Level Averaging Time [unit:ms] [scale:log]
        [tooltip: band-level averaging time in milliseconds]",
        100,1,10000,1)) * 0.001;
  dB_offset = ctl_group(hslider("[1] Level dB Offset [unit:dB]
        [tooltip: Level offset in decibels]",
        50,0,100,1));
};

spectral_level_demo = mth_octave_spectral_level_demo(1.5); // 2/3 octave

//---------------- (third|half)_octave_(analyzer|filterbank) -----------------

// Named special cases of mth_octave_* with defaults filled in:

third_octave_analyzer(N) = mth_octave_analyzer_default(3,10000,N);
third_octave_filterbank(N) = mth_octave_filterbank_default(3,10000,N);
// Third-Octave Filter-Banks have been used in audio for over a century.
// See, e.g.,
//   Acoustics [the book], by L. L. Beranek
//   Amer. Inst. Physics for the Acoustical Soc. America,
//   http://asa.aip.org/publications.html, 1986 (1st ed.1954)

// Third-octave bands across the audio spectrum are too wide for current
// typical computer screens, so half-octave bands are the default:
half_octave_analyzer(N) = mth_octave_analyzer_default(2,10000,N);
half_octave_filterbank(N) = mth_octave_filterbank_default(2,10000,N);

octave_filterbank(N) = mth_octave_filterbank_default(1,10000,N);
octave_analyzer(N) = mth_octave_analyzer_default(1,10000,N);

//========================== Filter-Bank Demos ==============================
// Graphic Equalizer: Each filter-bank output signal routes through a fader.
//
// USAGE: _ : mth_octave_filterbank_demo(M) : _
// where
//   M = number of bands per octave

mth_octave_filterbank_demo(M) = bp1(bp,mthoctavefilterbankdemo) with {
   bp1 = component("effect.lib").bypass1;
   mofb_group(x) = vgroup("CONSTANT-Q FILTER BANK (Butterworth dyadic tree)
     [tooltip: See Faust's filter.lib for documentation and references]", x);
   bypass_group(x) = mofb_group(hgroup("[0]", x));
   slider_group(x) = mofb_group(hgroup("[1]", x));
   N = 10*M; // total number of bands (highpass band, octave-bands, dc band)
   ftop = 10000;
   mthoctavefilterbankdemo = chan;
   chan = mth_octave_filterbank_default(M,ftop,N) :
         sum(i,N,(*(db2linear(fader(N-i)))));
   fader(i) = slider_group(vslider("[%2i] [unit:dB]
            [tooltip: Bandpass filter gain in dB]", -10, -70, 10, 0.1)) :
            smooth(0.999);
   bp = bypass_group(checkbox("[0] Bypass
            [tooltip: When this is checked, the filter-bank has no effect]"));
};

filterbank_demo = mth_octave_filterbank_demo(1); // octave-bands = default

//=========== Arbritary-Crossover Filter-Banks and Spectrum Analyzers ========
```

```
// These are similar to the Mth-octave filter-banks above, except that the
// band-split frequencies are passed explicitly as arguments.
//
// USAGE:
//    _ : filterbank (O,freqs) : par(i,N,_); // Butterworth band-splits
//    _ : filterbanki(O,freqs) : par(i,N,_); // Inverted-dc version
//    _ : analyzer  (O,freqs) : par(i,N,_); // No delay equalizer
//
// where
//   O    = band-split filter order (ODD integer required for filterbank[i])
//   freqs = (fc1,fc2,...,fcNs) [in numerically ascending order], where
//          Ns=N-1 is the number of octave band-splits
//          (total number of bands N=Ns+1).
//
// If frequencies are listed explicitly as arguments, enclose them in parens:
//
//    _ : filterbank(3,(fc1,fc2)) : _,_,_
//
// ACKNOWLEDGMENT
//    Technique for processing a variable number of signal arguments due
//    to Yann Orlarey (as is the entire Faust framework!)
//
//------------------------------ analyzer ---------------------------------------
analyzer(O,lfreqs) = _ <: bsplit(nb) with
{
   nb = count(lfreqs);
   fc(n) = take(n, lfreqs);
   lp(n) = lowpass(O,fc(n));
   hp(n) = highpass(O,fc(n));
   bsplit(0) = _;
   bsplit(i) = hp(i), (lp(i) <: bsplit(i-1));
};

//------------------------------ filterbank ---------------------------------------
filterbank(O,lfreqs) = analyzer(O,lfreqs) : delayeq(nb) with
{
   nb = count(lfreqs);
   fc(n) = take(n, lfreqs);
   ap(n) = highpass_plus_lowpass(O,fc(n));
   delayeq(1) = _,_; // par(i,0,...) does not fly
   delayeq(nb) = par(i,nb-1,apchain(nb-1-i)),_,_;
   apchain(0) = _;
   apchain(i) = ap(i) : apchain(i-1);
};

//------------------------------ filterbanki ---------------------------------------
filterbanki(O,lfreqs) = _ <: bsplit(nb) with
{
   fc(n) = take(n, lfreqs);
   lp(n) = lowpass(O,fc(n));
   hp(n) = highpass(O,fc(n));
   ap(n) = highpass_minus_lowpass(O,fc(n));
   bsplit(0) = *(-1.0);
   bsplit(i) = (hp(i) : delayeq(i-1)), (lp(i) <: bsplit(i-1));
   delayeq(0) = _; // moving the *(-1) here inverts all outputs BUT dc
   delayeq(i) = ap(i) : delayeq(i-1);
};
```

Listing 4: `music.lib`

```
////////////////////////////////////////////////////////////////////////////////
// WARNING: Deprecated Library!!
// Read the README file in /libraries for more information
////////////////////////////////////////////////////////////////////////////////
```

```
5
6  /************************************************************************
7   ************************************************************************
8      FAUST library file
9      Copyright (C) 2003-2012 GRAME, Centre National de Creation Musicale
10     --------------------------------------------------------------------
11     This program is free software; you can redistribute it and/or modify
12     it under the terms of the GNU Lesser General Public License as
13     published by the Free Software Foundation; either version 2.1 of the
14     License, or (at your option) any later version.
15
16     This program is distributed in the hope that it will be useful,
17     but WITHOUT ANY WARRANTY; without even the implied warranty of
18     MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
19     GNU Lesser General Public License for more details.
20
21     You should have received a copy of the GNU Lesser General Public
22     License along with the GNU C Library; if not, write to the Free
23     Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA
24     02111-1307 USA.
25
26     EXCEPTION TO THE LGPL LICENSE : As a special exception, you may create a
27     larger FAUST program which directly or indirectly imports this library
28     file and still distribute the compiled code generated by the FAUST
29     compiler, or a modified version of this compiled code, under your own
30     copyright and license. This EXCEPTION TO THE LGPL LICENSE explicitly
31     grants you the right to freely choose the license for the resulting
32     compiled code. In particular the resulting compiled code has no obligation
33     to be LGPL or GPL. For example you are free to choose a commercial or
34     closed source license or any other license if you decide so.
35
36   ************************************************************************
37   ************************************************************************/
38
39 declare name "Music Library";
40 declare author "GRAME";
41 declare copyright "GRAME";
42 declare version "1.0";
43 declare license "LGPL with exception";
44 declare deprecated "This library is deprecated and is not maintained anymore. It will be
       removed in August 2017.";
45
46 import("math.lib");
47
48 //-----------------------------------------------
49 //                DELAY LINE
50 //-----------------------------------------------
51 frac(n)               = n-int(n);
52 index(n)              = &(n-1) ~ +(1);             // n = 2**i
53 //delay(n,d,x) = rwtable(n, 0.0, index(n), x, (index(n)-int(d)) & (n-1));
54 delay(n,d,x)   = x@(int(d)&(n-1));
55 fdelay(n,d,x)  = delay(n,int(d),x)*(1 - frac(d)) + delay(n,int(d)+1,x)*frac(d);
56
57
58 delay1s(d)     = delay(65536,d);
59 delay2s(d)     = delay(131072,d);
60 delay5s(d)     = delay(262144,d);
61 delay10s(d)    = delay(524288,d);
62 delay21s(d)    = delay(1048576,d);
63 delay43s(d)    = delay(2097152,d);
64
65 fdelay1s(d)    = fdelay(65536,d);
66 fdelay2s(d)    = fdelay(131072,d);
67 fdelay5s(d)    = fdelay(262144,d);
68 fdelay10s(d)   = fdelay(524288,d);
69 fdelay21s(d)   = fdelay(1048576,d);
70 fdelay43s(d)   = fdelay(2097152,d);
71
```

```
72  millisec   = SR/1000.0;
73
74  time1s  = hslider("time", 0, 0, 1000, 0.1)*millisec;
75  time2s  = hslider("time", 0, 0, 2000, 0.1)*millisec;
76  time5s  = hslider("time", 0, 0, 5000, 0.1)*millisec;
77  time10s = hslider("time", 0, 0, 10000, 0.1)*millisec;
78  time21s = hslider("time", 0, 0, 21000, 0.1)*millisec;
79  time43s = hslider("time", 0, 0, 43000, 0.1)*millisec;
80
81
82  echo1s = vgroup("echo 1000", +~(delay(65536, int(hslider("millisecond", 0, 0, 1000, 0.10)*
        millisec)-1) * (hslider("feedback", 0, 0, 100, 0.1)/100.0)));
83  echo2s = vgroup("echo 2000", +~(delay(131072, int(hslider("millisecond", 0, 0, 2000, 0.25)*
        millisec)-1) * (hslider("feedback", 0, 0, 100, 0.1)/100.0)));
84  echo5s = vgroup("echo 5000", +~(delay(262144, int(hslider("millisecond", 0, 0, 5000, 0.50)*
        millisec)-1) * (hslider("feedback", 0, 0, 100, 0.1)/100.0)));
85  echo10s = vgroup("echo 10000", +~(delay(524288, int(hslider("millisecond", 0, 0, 10000,
        1.00)*millisec)-1) * (hslider("feedback", 0, 0, 100, 0.1)/100.0)));
86  echo21s = vgroup("echo 21000", +~(delay(1048576, int(hslider("millisecond", 0, 0, 21000,
        1.00)*millisec)-1) * (hslider("feedback", 0, 0, 100, 0.1)/100.0)));
87  echo43s = vgroup("echo 43000", +~(delay(2097152, int(hslider("millisecond", 0, 0, 43000,
        1.00)*millisec)-1) * (hslider("feedback", 0, 0, 100, 0.1)/100.0)));
88
89
90  //-------------------------sdelay(N,it,dt)---------------------------
91  // s(mooth)delay : a mono delay that doesn't click and doesn't
92  // transpose when the delay time is changed. It takes 4 input signals
93  // and produces a delayed output signal
94  //
95  // USAGE :  ... : sdelay(N,it,dt) : ...
96  //
97  // Where :
98  //  <N>  = maximal delay in samples (must be a constant power of 2, for example 65536)
99  //  <it> = interpolation time (in samples) for example 1024
100 //  <dt> = delay time (in samples)
101 //  <  > = input signal we want to delay
102 //-----------------------------------------------------------------------
103
104 sdelay(N, it, dt) = ctrl(it,dt),_ : ddi(N)
105
106     with {
107
108         //-------------------------ddi(N,i,d0,d1)-----------------------------
109         //  DDI (Double Delay with Interpolation) : the input signal is sent to two
110         //  delay lines. The outputs of these delay lines are crossfaded with
111         //  an interpolation stage. By acting on this interpolation value one
112         //  can move smoothly from one delay to another. When <i> is 0 we can
113         //  freely change the delay time <d1> of line 1, when it is 1 we can freely change
114         //  the delay time <d0> of line 0.
115         //
116         //  <N>  = maximal delay in samples (must be a power of 2, for example 65536)
117         //  <i>  = interpolation value between 0 and 1 used to crossfade the outputs of the
118         //         two delay lines (0.0: first delay line, 1.0: second delay line)
119         //  <d0> = delay time of delay line 0 in samples between 0 and <N>-1
120         //  <d1> = delay time of delay line 1 in samples between 0 and <N>-1
121         //  <  > = the input signal we want to delay
122         //-----------------------------------------------------------------------
123         ddi(N, i, d0, d1) = _ <: delay(N,d0), delay(N,d1) : interpolate(i);
124
125
126         //-----------------------------ctrl(it,dt)-----------------------------------
127         //  Control logic for a Double Delay with Interpolation according to two
128         //
129         //  USAGE : ctrl(it,dt)
130         //  where :
131         //  <it> an interpolation time (in samples, for example 256)
132         //  <dt> a delay time (in samples)
133         //
```

```
134         //  ctrl produces 3 outputs : an interpolation value <i> and two delay
135         //  times <d0> and <d1>. These signals are used to control a ddi (Double Delay with
                Interpolation).
136         //  The principle is to detect changes in the input delay time dt, then to
137         //  change the delay time of the delay line currently unused and then by a
138         //  smooth crossfade to remove the first delay line and activate the second one.
139         //
140         //  The control logic has an internal state controlled by 4 elements
141         //  <v> : the interpolation variation (0, 1/it, -1/it)
142         //  <i> : the interpolation value (between 0 and 1)
143         //  <d0>: the delay time of line 0
144         //  <d1>: the delay time of line 1
145         //
146         //  Please note that the last stage (!,_,_,_) cut <v> because it is only
147         //  used internally.
148         //--------------------------------------------------------------------------
149         ctrl(it, dt) = \(v,ip,d0,d1).( (nv, nip, nd0, nd1)
150             with {
151
152                 // interpolation variation
153                 nv = if (v!=0.0,                        // if variation we are interpolating
154                        if( (ip>0.0) & (ip<1.0), v , 0), //     should we continue or not ?
155                     if ((ip==0.0) & (dt!=d0), 1.0/it,  // if true xfade from dl0 to dl1
156                     if ((ip==1.0) & (dt!=d1), -1.0/it,  // if true xfade from dl1 to dl0
157                     0)));                               // nothing to change
158                 // interpolation value
159                 nip = ip+nv : min(1.0) : max(0.0);
160
161                 // update delay time of line 0 if needed
162                 nd0 = if ((ip >= 1.0) & (d1!=dt), dt, d0);
163
164                 // update delay time of line 0 if needed
165                 nd1 = if ((ip <= 0.0) & (d0!=dt), dt, d1);
166
167             } ) ~ (_,_,_,_) : (!,_,_,_);
168     };
169
170
171 //-----------------------------------------------
172 //         Tempo, beats and pulses
173 //-----------------------------------------------
174
175 tempo(t)   = (60*SR)/t;          // tempo(t) -> samples
176
177 period(p)  = %(int(p))~+(1);     // signal en dent de scie de periode p
178 pulse(t)   = period(t)==0;       // pulse (10000...) de periode p
179 pulsen(n,t) = period(t)<n;       // pulse (1110000...) de taille n et de periode p
180 beat(t)    = pulse(tempo(t));    // pulse au tempo t
181
182
183 //-----------------------------------------------
184 //  conversions between db and linear values
185 //-----------------------------------------------
186
187 db2linear(x)   = pow(10, x/20.0);
188 linear2db(x)   = 20*log10(x);
189
190
191 //================================================
192 //         Random and Noise generators
193 //================================================
194
195
196 //-----------------------------------------------
197 //         noise : Noise generator
198 //-----------------------------------------------
199
200 random     = +(12345) ~ *(1103515245); // "linear congruential"
```

```
RANDMAX    = 2147483647.0; // = 2^31-1 = MAX_SIGNED_INT in 32 bits

noise      = random / RANDMAX;


//-----------------------------------------------
// Generates multiple decorrelated random numbers
// in parallel. Expects n>0.
//-----------------------------------------------

multirandom(n) = randomize(n) ~_
with {
    randomize (1)  = +(12345) : *(1103515245);
    randomize (n)  = randomize(1) <: randomize(n-1),_;
};


//-----------------------------------------------
// Generates multiple decorrelated noises
// in parallel. Expects n>0.
//-----------------------------------------------

multinoise(n) = multirandom(n) : par(i,n,/(RANDMAX))
with {
    RANDMAX = 2147483647.0;
};


//-----------------------------------------------

noises(N,i) = multinoise(N) : selector(i,N);


//-----------------------------------------------
//         osc(freq) : Sinusoidal Oscillator
//-----------------------------------------------

tablesize  = 1 << 16;
samplingfreq  = SR;

time       = (+(1)~_ ) - 1;         // 0,1,2,3,...
sinwaveform    = float(time)*(2.0*PI)/float(tablesize) : sin;
coswaveform    = float(time)*(2.0*PI)/float(tablesize) : cos;

decimal(x) = x - floor(x);
phase(freq)    = freq/float(samplingfreq) : (+ : decimal) ~ _ : *(float(tablesize));
oscsin(freq)   = rdtable(tablesize, sinwaveform, int(phase(freq)) );
osccos(freq)   = rdtable(tablesize, coswaveform, int(phase(freq)) );
oscp(freq,p)   = oscsin(freq) * cos(p) + osccos(freq) * sin(p);
osc    = oscsin;
osci(freq) = s1 + d * (s2 - s1)
       with {
           i = int(phase(freq));
           d = decimal(phase(freq));
           s1 = rdtable(tablesize+1,sinwaveform,i);
           s2 = rdtable(tablesize+1,sinwaveform,i+1);};


//-----------------------------------------------
//         ADSR envelop
//-----------------------------------------------

// a,d,s,r = attack (sec), decay (sec), sustain (percentage of t), release (sec)
// t      = trigger signal ( >0 for attack, then release is when t back to 0)

adsr(a,d,s,r,t) = env ~ (_,_) : (!,_) // the 2 'state' signals are fed back
with {
    env (p2,y) =
```

```
        (t>0) & (p2|(y>=1)),       // p2 = decay-sustain phase
        (y + p1*u - (p2&(y>s))*v*y - p3*w*y) // y = envelop signal
    *((p3==0)|(y>=eps)) // cut off tails to prevent denormals
    with {
    p1 = (p2==0) & (t>0) & (y<1);     // p1 = attack phase
    p3 = (t<=0) & (y>0);              // p3 = release phase
    // #samples in attack, decay, release, must be >0
    na = SR*a+(a==0.0); nd = SR*d+(d==0.0); nr = SR*r+(r==0.0);
    // correct zero sustain level
    z = s+(s==0.0)*db2linear(-60);
    // attack, decay and (-60dB) release rates
    u = 1/na; v = 1-pow(z, 1/nd); w = 1-1/pow(z*db2linear(60), 1/nr);
    // values below this threshold are considered zero in the release phase
    eps = db2linear(-120);
    };
};


//-----------------------------------------------
//         Spatialisation
//-----------------------------------------------

panner(c) = _ <: *(1-c), *(c);

bus2 = _,_;
bus3 = _,_,_;
bus4 = _,_,_,_;
bus5 = _,_,_,_,_;
bus6 = _,_,_,_,_,_;
bus7 = _,_,_,_,_,_,_;
bus8 = _,_,_,_,_,_,_,_;

gain2(g) = *(g),*(g);
gain3(g) = *(g),*(g),*(g);
gain4(g) = *(g),*(g),*(g),*(g);
gain5(g) = *(g),*(g),*(g),*(g),*(g);
gain6(g) = *(g),*(g),*(g),*(g),*(g),*(g);
gain7(g) = *(g),*(g),*(g),*(g),*(g),*(g),*(g);
gain8(g) = *(g),*(g),*(g),*(g),*(g),*(g),*(g),*(g);


//-----------------------------------------------------
//
//                   GMEM SPAT
// n-outputs spatializer
// implementation of L. Pottier
//
//-----------------------------------------------------
//
// n = number of outputs
// r = rotation (between 0 et 1)
//  d = distance of the source (between 0 et 1)
//
//-----------------------------------------------------
spat(n,a,d) = _ <: par(i, n, *( scaler(i, n, a, d) : smooth(0.9999) ))
    with {
        scaler(i,n,a,d) = (d/2.0+0.5)
                        * sqrt( max(0.0, 1.0 - abs(fmod(a+0.5+float(n-i)/n, 1.0) - 0.5) * n * d
                              ) );
        smooth(c) = *(1-c) : +~*(c);
    };


//--------------- Second Order Generic Transfert Function -------------------------
// TF2(b0,b1,b2,a1,a2)
//
//-------------------------------------------------------------------------------
```

```
TF2(b0,b1,b2,a1,a2) = sub ~ conv2(a1,a2) : conv3(b0,b1,b2)
    with {
        conv3(k0,k1,k2,x)  = k0*x + k1*x' + k2*x'';
        conv2(k0,k1,x)     = k0*x + k1*x';
        sub(x,y)           = y-x;
    };


/************************** Break Point Functions *************************

bpf is an environment (a group of related definitions) that can be used to
create break-point functions. It contains three functions :
  - start(x,y) to start a break-point function
  - end(x,y) to end a break-point function
  - point(x,y) to add intermediate points to a break-point function

A minimal break-point function must contain at least a start and an end point :

  f = bpf.start(x0,y0) : bpf.end(x1,y1);

A more involved break-point function can contains any number of intermediate
points

  f = bpf.start(x0,y0) : bpf.point(x1,y1) : bpf.point(x2,y2) : bpf.end(x3,y3);

In any case the x_{i} must be in increasing order (for all i, x_{i} < x_{i+1})

For example the following definition :

  f = bpf.start(x0,y0) : ... : bpf.point(xi,yi) : ... : bpf.end(xn,yn);

implements a break-point function f such that :

  f(x) = y_{0} when x < x_{0}
  f(x) = y_{n} when x > x_{n}
  f(x) = y_{i} + (y_{i+1}-y_{i})*(x-x_{i})/(x_{i+1}-x_{i}) when x_{i} <= x and x < x_{i+1}

*****************************************************************************/

bpf = environment
{
  // Start a break-point function
  start(x0,y0) = \(x).(x0,y0,x,y0);

  // Add a break-point
  point(x1,y1) = \(x0,y0,x,y).(x1, y1, x , if (x < x0, y, if (x < x1, y0 + (x-x0)*(y1-y0)/(
      x1-x0), y1)));

  // End a break-point function
  end  (x1,y1) = \(x0,y0,x,y).(if (x < x0, y, if (x < x1, y0 + (x-x0)*(y1-y0)/(x1-x0), y1)));

  // definition of if
  if (c,t,e) = select2(c,e,t);
};


//----------------------------------Stereoize------------------------------
// Transform an arbitrary processor p into a stereo processor with 2 inputs
// and 2 outputs.
//-------------------------------------------------------------------------
stereoize(p) = S(inputs(p), outputs(p))
    with {
      // degenerated processor with no outputs
        S(n,0) = !,! : 0,0;      // just in case, probably a rare case

      // processors with no inputs
        S(0,1) = !,! : p <: _,_;  // add two fake inputs and split output
        S(0,2) = !,! : p;
```

```
403         S(0,n) = !,! : p,p :> _,_; // we are sure this will work if n is odd
404
405       // processors with one input
406         S(1,1) = p,p;             // add two fake inputs and split output
407         S(1,n) = p,p :> _,_;      // we are sure this will work if n is odd
408
409       // processors with two inputs
410         S(2,1) = p <: _,_;        // split the output
411         S(2,2) = p;               // nothing to do, p is already stereo
412
413       // processors with inputs > 2 and outputs > 2
414         S(n,m) = _,_ <: p,p :> _,_; // we are sure this works if n or p are odd
415     };
416
417
418 //----------------------------------Recursivize-----------------------------
419 // Create a recursion from two arbitrary processors p and q
420 //--------------------------------------------------------------------------
421 recursivize(p,q) = (_,_,_,_ :> stereoize(p)) ~ stereoize(q);
422
423
424 //----------------------------------Automat---------------------------------
425 // Record and replay to the values the input signal in a loop
426 //
427 // USAGE: hslider(...) : automat(360, 15, 0.0)
428 //--------------------------------------------------------------------------
429
430 automat(bps, size, init, input) = rwtable(size+1, init, windex, input, rindex)
431     with {
432         clock  = beat(bps);
433         rindex  = int(clock) : (+ : %(size)) ~ _;  // each clock read the next entry of the
                table
434         windex  = if (timeToRenew, rindex, size);  // we ignore input unless it is time to
                renew
435         if(cond,thn,els) = select2(cond,els,thn);
436         timeToRenew    = int(clock) & (inputHasMoved | (input <= init));
437         inputHasMoved  = abs(input-input') : countfrom(int(clock)') : >(0);
438         countfrom(reset) = (+ : if(reset, 0, _)) ~ _;
439     };
440
441
442 //----------------------------------bsmooth---------------------------------
443 // bsmooth : (block smooth) linear interpolation during a block of samples
444 //
445 // USAGE: hslider(...) : bsmooth
446 //--------------------------------------------------------------------------
447
448 bsmooth(c) = +(i) ~ _
449     with {
450         i = (c-c@n)/n;
451         n = min(4096, max(1, fvariable(int count, <math.h>)));
452     };
453
454
455 //----------------------------------chebychev-------------------------------
456 // chebychev(n) : chebychev transformation of order n
457 // USAGE: _ : chebychev(3) : _
458 //
459 //
460 // Semantics:
461 //  T[0](x) = 1,
462 //  T[1](x) = x,
463 //  T[n](x) = 2x*T[n-1](x) - T[n-2](x)
464 //
465 //  see : http://en.wikipedia.org/wiki/Chebyshev_polynomial
466 //--------------------------------------------------------------------------
467
468 chebychev(0) = !:1;
```

```
chebychev(1) = _;
chebychev(n) = _ <: *(2)*chebychev(n-1)-chebychev(n-2);


//--------------------------------chebychevpoly-------------------------------
//  chebychevpoly((c0,c1,...,cn)) : linear combination of the first Chebyshev polynomials
//  USAGE: _ : chebychevpoly((0.1,0.8,0.1)) : _
//
//
// Semantics:
//  chebychevpoly((c0,c1,...,cn)) = Sum of chebychev(i)*ci
//  see : http://www.csounds.com/manual/html/chebyshevpoly.html
//----------------------------------------------------------------------------

chebychevpoly(lcoef) = _ <: L(0,lcoef) :> _
    with {
        L(n,(c,cs)) = chebychev(n)*c, L(n+1,cs);
        L(n,c)      = chebychev(n)*c;
    };
```

Listing 5: `math.lib`

```
////////////////////////////////////////////////////////////////////////////////////
// WARNING: Deprecated Library!!
// Read the README file in /libraries for more information
////////////////////////////////////////////////////////////////////////////////////

/************************************************************************
 ************************************************************************
    FAUST library file
    Copyright (C) 2003-2012 GRAME, Centre National de Creation Musicale
    --------------------------------------------------------------------
    This program is free software; you can redistribute it and/or modify
    it under the terms of the GNU Lesser General Public License as
    published by the Free Software Foundation; either version 2.1 of the
    License, or (at your option) any later version.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
    GNU Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Public
    License along with the GNU C Library; if not, write to the Free
    Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA
    02111-1307 USA.

    EXCEPTION TO THE LGPL LICENSE : As a special exception, you may create a
    larger FAUST program which directly or indirectly imports this library
    file and still distribute the compiled code generated by the FAUST
    compiler, or a modified version of this compiled code, under your own
    copyright and license. This EXCEPTION TO THE LGPL LICENSE explicitly
    grants you the right to freely choose the license for the resulting
    compiled code. In particular the resulting compiled code has no obligation
    to be LGPL or GPL. For example you are free to choose a commercial or
    closed source license or any other license if you decide so.

 ************************************************************************
 ************************************************************************/

declare name "Math Library";
declare author "GRAME";
declare copyright "GRAME";
declare version "1.0";
```

```
declare license "LGPL with exception";
declare deprecated "This library is deprecated and is not maintained anymore. It will be
     removed in August 2017.";

//----------------------Mathematic library for Faust----------------------------
//
// Implementation as Faust foreign functions of math.h functions that are not
// part of Faust's primitives. Defines also various constants and several utilities
//
// ### History
// + 07/08/2015 [YO]  documentation comments
// + 20/06/2014 [SL]  added FTZ function
// + 20/06/2014 [SL]  added FTZ function
// + 22/06/2013 [YO]  added float|double|quad variants of some foreign functions
// + 28/06/2005 [YO]  postfixed functions with 'f' to force float version instead of double
// + 28/06/2005 [YO]  removed 'modf' because it requires a pointer as argument
//-----------------------------------------------------------------------------


//---------------------------------- SR -------------------------------------
// Current sampling rate (between 1Hz and 192000Hz). Constant during
// program execution.
//
// ### Usage:
//   `SR:_`
//-----------------------------------------------------------------------------
SR         = min(192000.0, max(1.0, fconstant(int fSamplingFreq, <math.h>)));


//---------------------------------- BS -------------------------------------
// Current block-size. Can change during the execution
//
// ### Usage:
//   `BS:_`
//-----------------------------------------------------------------------------
BS         = fvariable(int count, <math.h>);


//---------------------------------- PI -------------------------------------
// Constant PI in double precision
//
// ### Usage:
//   `PI:_`
//-----------------------------------------------------------------------------
PI         = 3.1415926535897932385;


//---------------------------------- FTZ ------------------------------------
// Flush to zero : force samples under the "maximum subnormal number"
// to be zero. Usually not needed in C++ because the architecture
// file take care of this, but can be useful in javascript for instance.
//
// ### Usage:
//   `_:ftz:_`
//
// see : <http://docs.oracle.com/cd/E19957-01/806-3568/ncg_math.html>
//-----------------------------------------------------------------------------
FTZ(x)     = x * (abs(x) > 1.17549435e-38);


//---------------------------------- neg ------------------------------------
// Invert the sign (-x) of a signal.
//
// ### Usage:
//   `_:neg:_`
//
//-----------------------------------------------------------------------------
neg(x)     = -x;
```

```
110
111
112 //---------------------------------- inv -----------------------------------------
113 // Compute the inverse (1/x) of the input signal
114 //
115 // ### Usage:
116 //    `_:inv:_`
117 //
118 //--------------------------------------------------------------------------------
119 inv(x)     = 1/x;
120
121
122 //---------------------------------- cbrt ----------------------------------------
123 // Computes the cube root of of the input signal.
124 //
125 // ### Usage:
126 //    `_:cbrt:_`
127 //--------------------------------------------------------------------------------
128 cbrt       = ffunction(float cbrtf|cbrt|cbrtl (float), <math.h>,"");
129
130
131 //---------------------------------- hypot(x,y) ----------------------------------
132 // Computes the euclidian distance of the two input signals
133 // sqrt(x*x+y*y) without undue overflow or underflow.
134 //
135 // ### Usage:
136 //    `hypot(x,y):_`
137 //    `_,_:hypot:_`
138 //--------------------------------------------------------------------------------
139 hypot      = ffunction(float hypotf|hypot|hypotl (float, float), <math.h>,"");
140
141
142 //---------------------------------- ldexp ---------------------------------------
143 // Takes two input signals: x and n, and multiplies x by 2 to the power n.
144 //
145 // ### Usage:
146 //    `_,_:ldexp:_`
147 //--------------------------------------------------------------------------------
148 ldexp      = ffunction(float ldexpf|ldexp|ldexpl (float, int), <math.h>,"");
149
150
151 //---------------------------------- scalb ---------------------------------------
152 // Takes two input signals: x and n, and multiplies x by 2 to the power n.
153 //
154 // ### Usage:
155 //    `_,_:scalb:_`
156 //--------------------------------------------------------------------------------
157 scalb      = ffunction(float scalbnf|scalbn|scalbnl (float, int), <math.h>,"");
158
159
160 //---------------------------------- log1p(x) ------------------------------------
161 // Computes log(1 + x) without undue loss of accuracy when x is nearly zero.
162 //
163 // ### Usage:
164 // +  `log1p(x):_`
165 // +  `_:log1p:_`
166 //--------------------------------------------------------------------------------
167 log1p      = ffunction(float log1pf|log1p|log1pl (float), <math.h>,"");
168
169
170 //---------------------------------- logb ----------------------------------------
171 // Return exponent of the input signal as a floating-point number
172 //
173 // ### Usage:
174 //    `_:logb:_`
175 //--------------------------------------------------------------------------------
176 logb       = ffunction(float logbf|logb|logbl (float), <math.h>,"");
177
```

```
178
179 //-------------------------------- ilogb ------------------------------------
180 // Return exponent of the input signal as an integer number
181 //
182 // ### Usage:
183 //    `_:ilogb:_`
184 //---------------------------------------------------------------------------
185 ilogb      = ffunction(int ilogbf|ilogb|ilogbl (float), <math.h>,"");
186
187
188 //-------------------------------- expm1 ------------------------------------
189 // Return exponent of the input signal minus 1 with better precision.
190 //
191 // ### Usage:
192 //    `_:expm1:_`
193 //---------------------------------------------------------------------------
194 expm1      = ffunction(float expm1f|expm1|expm1l (float), <math.h>,"");
195
196
197 //-------------------------------- acosh ------------------------------------
198 // Computes the principle value of the inverse hyperbolic cosine
199 // of the input signal.
200 //
201 // ### Usage:
202 //    `_:acosh:_`
203 //---------------------------------------------------------------------------
204 acosh      = ffunction(float acoshf|acosh|acoshl (float), <math.h>, "");
205
206
207 //-------------------------------- asinh(x) ---------------------------------
208 // Computes the inverse hyperbolic sine of the input signal.
209 //
210 // ### Usage:
211 //    `_:asinh:_`
212 //---------------------------------------------------------------------------
213 asinh      = ffunction(float asinhf|asinh|asinhl (float), <math.h>, "");
214
215
216 //-------------------------------- atanh(x) ---------------------------------
217 // Computes the inverse hyperbolic tangent of the input signal.
218 //
219 // ### Usage:
220 //    `_:atanh:_`
221 //---------------------------------------------------------------------------
222 atanh      = ffunction(float atanhf|atanh|atanhl (float), <math.h>, "");
223
224
225 //-------------------------------- sinh -------------------------------------
226 // Computes the hyperbolic sine of the input signal.
227 //
228 // ### Usage:
229 //    `_:sinh:_`
230 //---------------------------------------------------------------------------
231 sinh       = ffunction(float sinhf|sinh|sinhl (float), <math.h>, "");
232
233
234 //-------------------------------- cosh -------------------------------------
235 // Computes the hyperbolic cosine of the input signal.
236 //
237 // ### Usage:
238 //    `_:cosh:_`
239 //---------------------------------------------------------------------------
240 cosh       = ffunction(float coshf|cosh|coshl (float), <math.h>, "");
241
242
243 //-------------------------------- tanh -------------------------------------
244 // Computes the hyperbolic tangent of the input signal.
245 //
```

```
// ### Usage:
//    `_:tanh:_`
//------------------------------------------------------------------------------
tanh       = ffunction(float tanhf|tanh|tanhl (float), <math.h>,"");


//---------------------------------- erf -----------------------------------------
// Computes the error function of the input signal.
//
// ### Usage:
//    `_:erf:_`
//------------------------------------------------------------------------------
erf        = ffunction(float erff|erf|erfl(float), <math.h>,"");


//---------------------------------- erf -----------------------------------------
// Computes the complementary error function of the input signal.
//
// ### Usage:
//    `_:erfc:_`
//------------------------------------------------------------------------------
erfc       = ffunction(float erfcf|erfc|erfcl(float), <math.h>,"");


//---------------------------------- gamma -----------------------------------------
// Computes the gamma function of the input signal.
//
// ### Usage:
//    `_:gamma:_`
//------------------------------------------------------------------------------
gamma      = ffunction(float tgammaf|tgamma|tgammal(float), <math.h>,"");


//---------------------------------- lgamma -----------------------------------------
// Calculates the natural logorithm of the absolute value of
// the gamma function of the input signal.
//
// ### Usage:
//    `_:lgamma:_`
//------------------------------------------------------------------------------
lgamma     = ffunction(float lgammaf|lgamma|lgammal(float), <math.h>,"");


//---------------------------------- J0 -----------------------------------------
// Computes the Bessel function of the first kind of order 0
// of the input signal.
//
// ### Usage:
//    `_:J0:_`
//------------------------------------------------------------------------------
J0         = ffunction(float j0(float), <math.h>,"");

//---------------------------------- J1 -----------------------------------------
// Computes the Bessel function of the first kind of order 1
// of the input signal.
//
// ### Usage:
//    `_:J1:_`
//------------------------------------------------------------------------------
J1         = ffunction(float j1(float), <math.h>,"");

//---------------------------------- Jn -----------------------------------------
// Computes the Bessel function of the first kind of order n
// (first input signal) of the second input signal.
//
// ### Usage:
//    `_,_:Jn:_`
//------------------------------------------------------------------------------
```

```
Jn         = ffunction(float jn(int, float), <math.h>,"");


//---------------------------------- Y0 ------------------------------------
// Computes the linearly independent Bessel function of the second kind
// of order 0 of the input signal.
//
// ### Usage:
//    `_:Y0:_`
//--------------------------------------------------------------------------
Y0         = ffunction(float y0(float), <math.h>,"");

//---------------------------------- Y1 ------------------------------------
// Computes the linearly independent Bessel function of the second kind
// of order 1 of the input signal.
//
// ### Usage:
//    `_:Y0:_`
//--------------------------------------------------------------------------
Y1         = ffunction(float y1(float), <math.h>,"");

//---------------------------------- Yn ------------------------------------
// Computes the linearly independent Bessel function of the second kind
// of order n (first input signal) of the second input signal.
//
// ### Usage:
//    `_,_:Yn:_`
//--------------------------------------------------------------------------
Yn         = ffunction(float yn(int, float), <math.h>,"");


// -- Miscellaneous Functions

fabs = abs;
fmax = max;
fmin = min;

//--------------- isnan(x) ----------------
// return non-zero if and only if x is a NaN,
//
// ### Usage:
//    `_:isnan:_`
//
// #### Where:
// + x = signal to analyse
//
//-----------------------------------------
isnan      = ffunction(int isnan (float),<math.h>,"");
nextafter  = ffunction(float nextafter(float, float),<math.h>,"");


//--------------------------------- count(l) ---------------------------------
// Count the number of elements of list l
//
// ### Usage:
//    `count ((10,20,30,40))  -> 4`
//
// #### Where:
// + l = list of elements
//
//----------------------------------------------------------------------------
count ((xs, xxs)) = 1 + count(xxs);
count (xx) = 1;


//------------------------------ take(e,l) ----------------------------------
// Take an element from a list
//
```

```
// ### Usage:
//    `take (3,(10,20,30,40)) -> 30`
//
// #### Where:
// + p = position (starting at 1)
// + l = list of elements
//
//-------------------------------------------------------------------------
take (1, (xs, xxs))   = xs;
take (1, xs)          = xs;
take (nn, (xs, xxs))  = take (nn-1, xxs);


//---------------------------- subseq(l, p, n) -------------------------------
// Extract a part of a list
//
// ### Usage:
// + `subseq((10,20,30,40,50,60), 1, 3) -> (20,30,40)`
// + `subseq((10,20,30,40,50,60), 4, 1) -> 50`
//
// #### Where:
// + l = list
// + p = start point (0: begin of list)
// + n = number of elements
//
// #### Note:
// Faust doesn't have proper lists. Lists are simulated with parallel
// compositions and there is no empty list
//
//-------------------------------------------------------------------------
subseq((head, tail), 0, 1)   = head;
subseq((head, tail), 0, n)   = head, subseq(tail, 0, n-1);
subseq((head, tail), p, n)   = subseq(tail, p-1, n);
subseq(head, 0, n)           = head;


//---------------------------- interpolate(i) -------------------------------
// linear interpolation between two signals
//
// ### Usage:
//    `_,_:interpolate(i):_`
//
// #### Where:
// + i = interpolation control between 0 and 1 (0: first input; 1: second input)
//
//-------------------------------------------------------------------------
interpolate(i) = *(1.0-i),*(i) : +;


//---------------------------- if(c,t,e)) -----------------------------------
// if-then-else implemented with a select2
//
// ### Usage:
// +  `if(c, then, else):_`
// +  `_,_:if(c):_`
//
// #### Where:
// + c = condition
// + t = signal selected while c is true
// + e = signal selected while c is false
//
//-------------------------------------------------------------------------
if(cond,thn,els) = select2(cond,els,thn);


//---------------------------- countdown(n,trig) ----------------------------
// Starts counting down from n included to 0. While trig is 1 the output is n.
// The countdown starts with the transition of trig from 1 to 0. At the end
```

```
// of the countdown the output value will remain at 0 until the next trig.
//
// ### Usage:
// + `countdown(n,trig):_`
// + `_:countdown(n):_`
// + `_,_:countdown:_`
//
// #### Where
// + n : the starting point of the countdown
// + trig : the trigger signal (1: start at n; 0: decrease until 0)
//------------------------------------------------------------------------------
countdown(count, trig) = \(c).(if(trig>0, count, max(0, c-1))) ~_;


//---------------------------- countup(n,trig) --------------------------------
// Starts counting up from 0 to n included. While trig is 1 the output is 0.
// The countup starts with the transition of trig from 1 to 0. At the end
// of the countup the output value will remain at n until the next trig.
//
// ### Usage:
// + `countup(n,trig):_`
// + `_:countup(n):_`
// + `_,_:countup:_`
//
// #### Where
// + n : the starting point of the countup
// + trig : the trigger signal (1: start at 0; 0: increase until n)
//------------------------------------------------------------------------------
countup(count, trig)  = \(c).(if(trig>0, 0, min(count, c+1))) ~_;


//-------------------------------- bus(n) --------------------------------------
// n parallel cables
//
// ### Usage:
//    `bus(4) -> _,_,_,_`
//
// #### Where:
// + n = is an integer known at compile time that indicates the number of parallel cables.
//
//------------------------------------------------------------------------------
bus(2) = _,_; // avoids a lot of "bus(1)" labels in block diagrams
bus(n) = par(i, n, _);


//---------------------------- selector(i,n) ----------------------------------
// Selects the ith input among n at compile time
//
// ### Usage:
// `_,_,_,_:selector(2,4):_` selects the 3rd input among 4
//
// #### Where:
// + i = input to select (int, numbered from 0, known at compile time)
// + n = number of inputs (int, known at compile time, n > i)
//
//------------------------------------------------------------------------------
selector(i,n) = par(j, n, S(i, j)) with { S(i,i) = _; S(i,j) = !; };


//---------------------------- selectn(N,i) -----------------------------------
// Selects the ith input among N at run time
//
// ### Usage:
// `_,_,_,_:selectn(4,2):_` selects the 3rd input among 4
//
// #### Where:
// + N = number of inputs (int, known at compile time, N > 0)
// + i = input to select (int, numbered from 0)
```

```
//
// #### Example test program:
// `N=64; process = par(n,N, (par(i,N,i) : selectn(N,n)));`
//-----------------------------------------------------------------------------
selectn(N,i) = S(N,0)
    with {
       S(1,offset) = _;
       S(n,offset) = S(left, offset), S(right, offset+left) : select2(i >= offset+left)
            with {
                right = int(n/2);
                left  = n-right;
            };
    };

//-------------------------- interleave(row,col) ------------------------------
// interleave row*col cables from column order to row order.
// input : x(0), x(1), x(2) ..., x(row*col-1)
// output: x(0+0*row), x(0+1*row), x(0+2*row), ..., x(1+0*row), x(1+1*row), x(1+2*row), ...
//
// ### Usage:
//    `_,_,_,_,_,_:interleave(3,2):_,_,_,_,_,_`
//
// #### Where:
// + row = the number of row (int, known at compile time)
// + column = the number of column (int, known at compile time)
//
//-----------------------------------------------------------------------------
interleave(row,col) = bus(row*col) <: par(r, row, par(c, col, selector(r+c*row,row*col)));


//------------------------------ butterfly(n) ---------------------------------
// Addition (first half) then substraction (second half) of interleaved signals.
//
// ### Usage:
//    `_,_,_,_:butterfly(4):_,_,_,_`
//
// #### Where:
// + n = size of the butterfly (n is int, even and known at compile time)
//
//-----------------------------------------------------------------------------
butterfly(n) = bus(n) <: interleave(n/2,2), interleave(n/2,2) : par(i, n/2, +), par(i, n/2,
     -);


//------------------------------ hadamard(n) ----------------------------------
// hadamard matrix function of size n = 2^k
//
// ### Usage:
//    `_,_,_,_:hadamard(4):_,_,_,_`
//
// #### Where:
// + n = 2^k, size of the matrix (int, must be known at compile time)
//
// #### Note:
// Implementation contributed by Remy Muller.
//
//-----------------------------------------------------------------------------
hadamard(2) = butterfly(2);
hadamard(n) = butterfly(n) : (hadamard(n/2) , hadamard(n/2));


//------------------------------ dot(n) ---------------------------------------
// Dot product for two vectors of size n
//
// ### Usage:
//    `_,_,_,_,_,_:dot(3):_`
//
// #### Where:
```

```
// + n = size of the vectors (int, must be known at compile time)
//
//-------------------------------------------------------------------------------
dot(n) = interleave(n,2) : par(i,n,*) :> _;


//--------------------------------- cross(n) ----------------------------------
// cross n signals : (x1,x2,..,xn) -> (xn,..,x2,x1)
//
// ### Usage:
//    `_,_,_:cross(3):_,_,_`
//
// #### Where:
// + n = number of signals (int, must be known at compile time)
//
//-------------------------------------------------------------------------------
// cross n cables : (x1,x2,..,xn) -> (xn,..,x2,x1)
cross(n) = bus(n) <: par(i,n,selector(n-i-1,n));
```

Listing 6: `stdfaust.lib`

```
//################################## stdfaust.lib ########################################
// The purpose of this library is to give access to all the Faust standard libraries
// through a series of environment.
//########################################################################################

an = library("analyzers.lib");
ba = library("basics.lib");
co = library("compressors.lib");
de = library("delays.lib");
dm = library("demos.lib");
dx = library("dx7.lib");
en = library("envelopes.lib");
fi = library("filters.lib");
ho = library("hoa.lib");
ma = library("maths.lib");
ef = library("misceffects.lib");
os = library("oscillators.lib");
no = library("noises.lib");
pf = library("phaflangers.lib");
pm = library("physmodels.lib");
re = library("reverbs.lib");
ro = library("routes.lib");
sp = library("spats.lib");
si = library("signals.lib");
sy = library("synths.lib");
ve = library("vaeffects.lib");
sf = library("all.lib");
```