Virtual Electric Guitars and Effects Using Faust and Octave

Julius Smith
CCRMA, Stanford University

LAC-2008

March 1, 2008

Outline

- Overview and Demo
- Extended Karplus Strong (EKS) Elements
- Overdrive
- Amplifier Feedback
- Coupled Strings
- Wah Pedal
- Faust Libraries:
  - `filter.lib`
  - `effect.lib`
  - `osc.lib`

Why Resurrect These Old Algorithms Now?

(Extended Karplus Strong, Sullivan extensions, early waveguide)

- Some patents have expired
- Useful free methods are not in wide use
- Reference implementations in a modern framework

Why Resurrect These Old Algorithms Now?

(Extended Karplus Strong, Sullivan extensions, early waveguide)

- Some patents have expired
- Useful free methods are not in wide use
- Reference implementations in a modern framework

Why Resurrect These Old Algorithms Now?

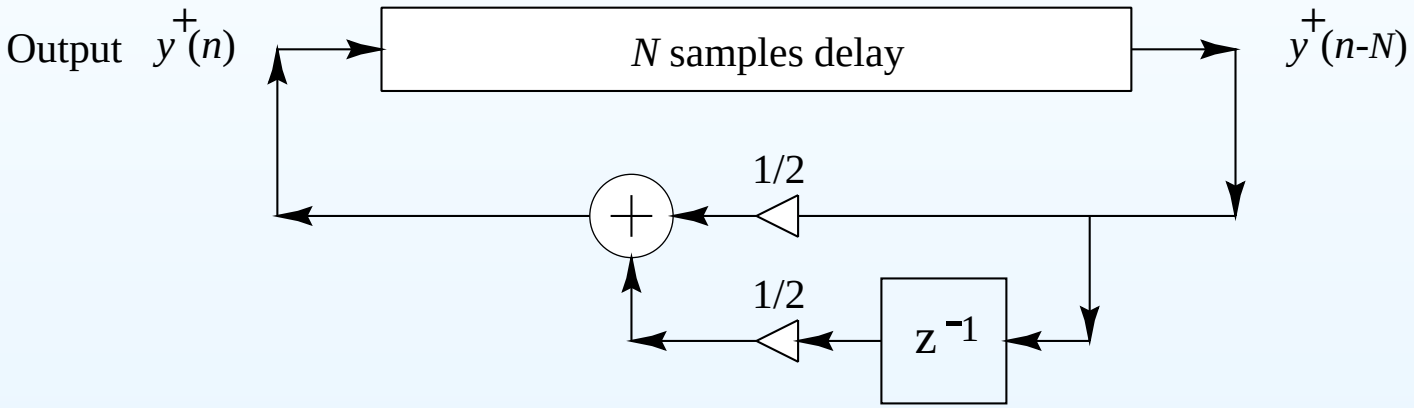
(Extended Karplus Strong, Sullivan extensions, early waveguide)

- Some patents have expired
- Useful free methods are not in wide use
- Reference implementations in a modern framework

Extended Karplus-Strong (EKS) Algorithm

Karplus-Strong (KS) Algorithm (1983)

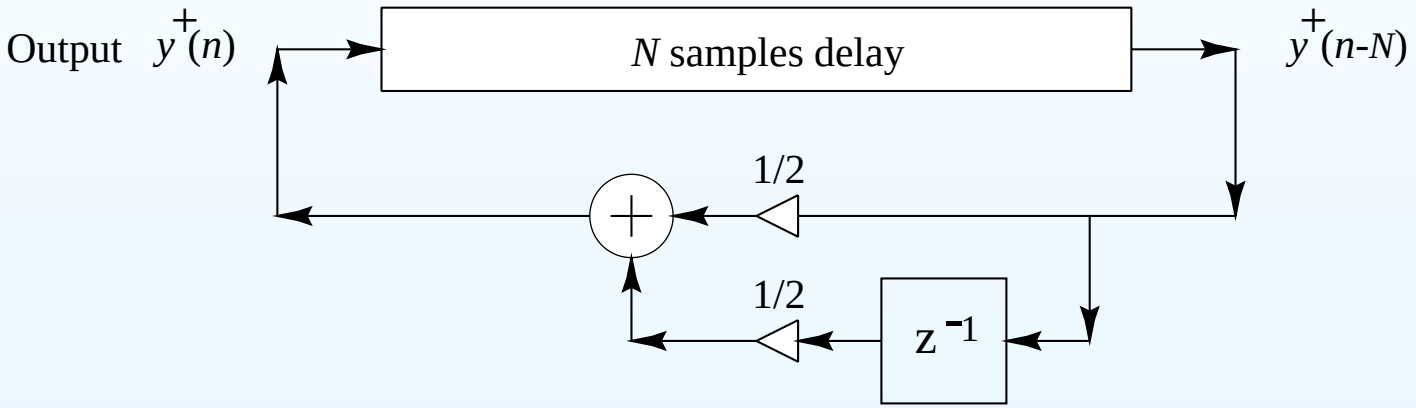
Digital Filter Interpretation:



- Discovered as “self-modifying wavetable synthesis”
- Wavetable is preferably initialized with random numbers
- Patents now expired

Karplus-Strong (KS) Algorithm (1983)

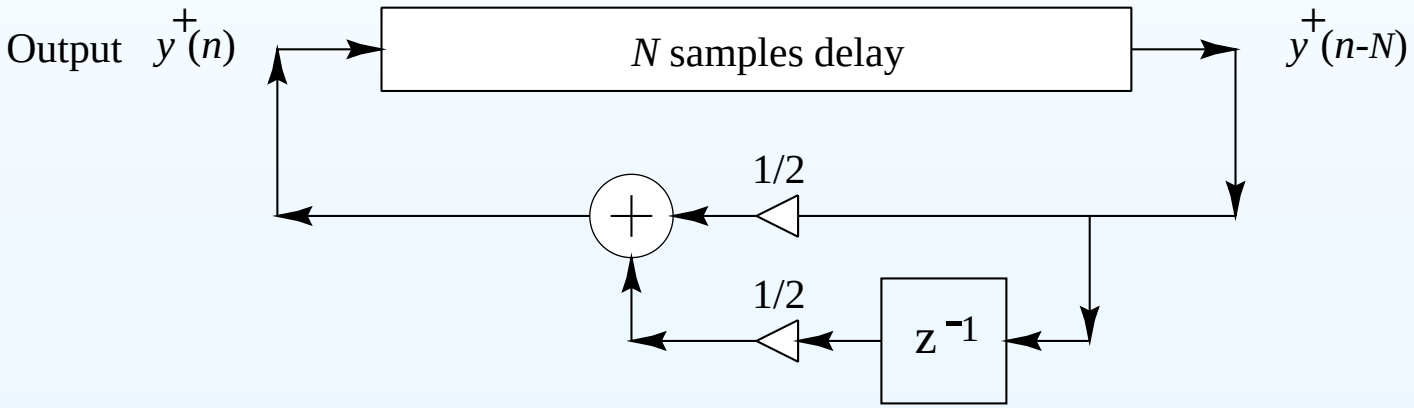
Digital Filter Interpretation:



- Discovered as “self-modifying wavetable synthesis”
- Wavetable is preferably initialized with random numbers
- Patents now expired

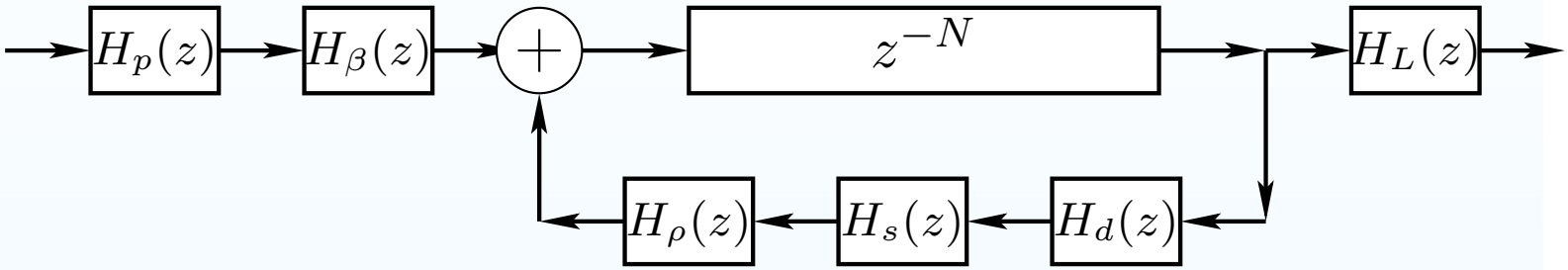
Karplus-Strong (KS) Algorithm (1983)

Digital Filter Interpretation:



- Discovered as “self-modifying wavetable synthesis”
- Wavetable is preferably initialized with random numbers
- Patents now expired

EKS Algorithms



$$N = \text{pitch period } (2\times \text{ string length}) \text{ in samples}$$

$$H_p(z) = \frac{1-p}{1-p\,z^{-1}} = \text{pick-direction lowpass filter}$$

$$H_\beta(z) = 1 - z^{-\beta N} = \textbf{pick-position comb filter}, \beta \in (0,1)$$

$$H_d(z) = \textbf{string-damping filter} \text{ (one/two poles/zeros typical)}$$

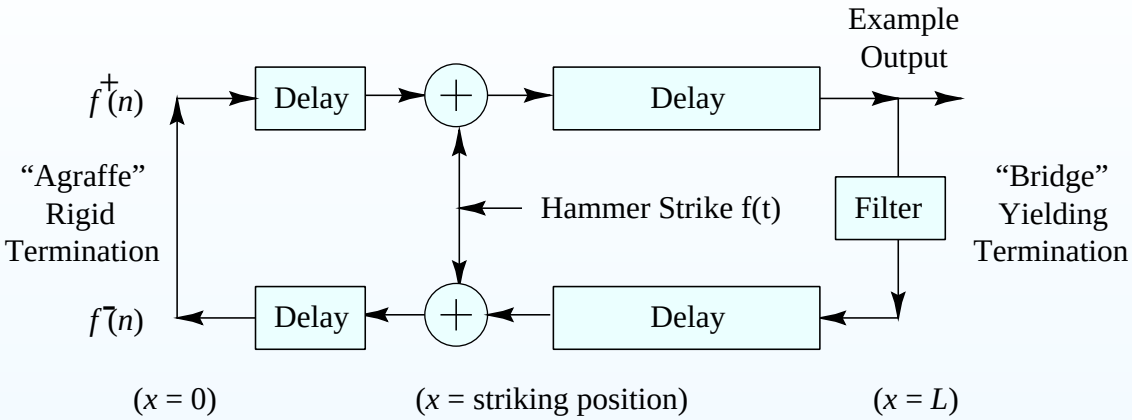
$$H_s(z) = \text{string-stiffness allpass filter (several poles and zeros)}$$

$$H_\rho(z) = \frac{\rho(N) - z^{-1}}{1-\rho(N)\,z^{-1}} = \text{first-order } \textbf{string-tuning} \text{ allpass filter}$$

$$H_L(z) = \frac{1-R_L}{1-R_L\,z^{-1}} = \textbf{dynamic-level lowpass filter}$$

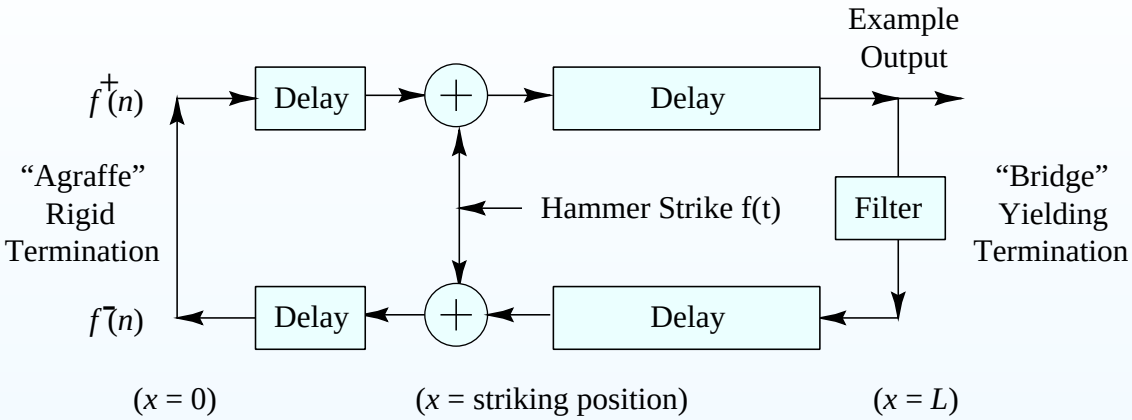
Pick Position Comb Filter

String Model Excited Externally at One Point



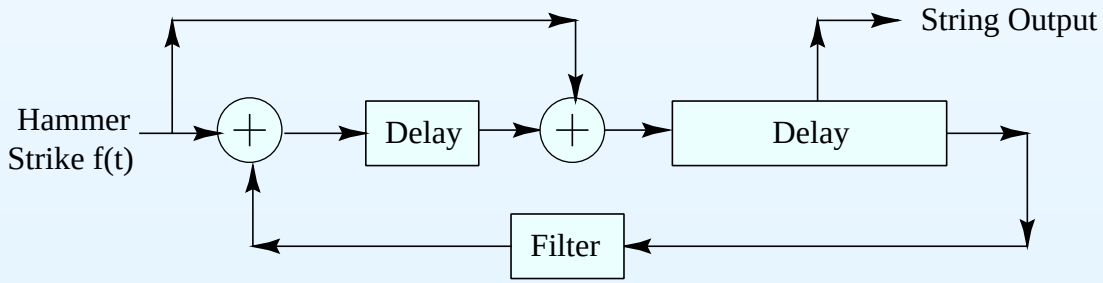
"Waveguide Formulation"

String Model Excited Externally at One Point

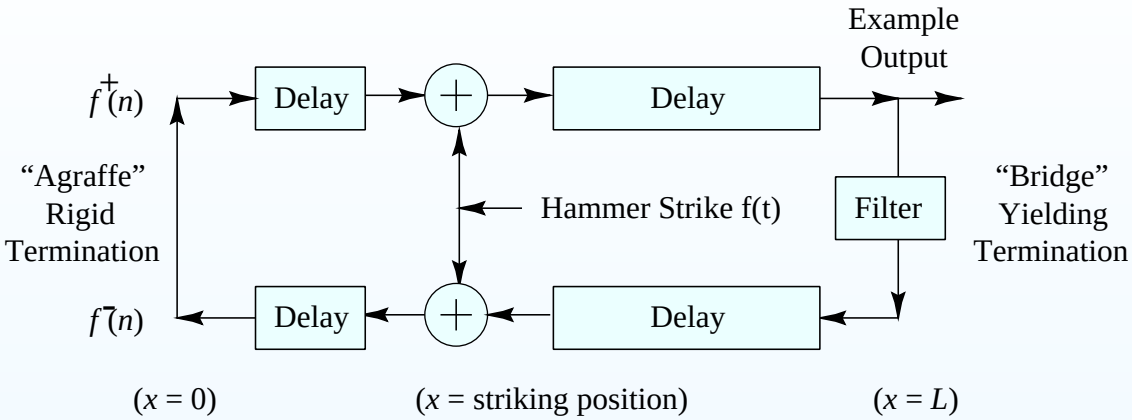


"Waveguide Formulation"

Equivalent System by Delay Consolidation:

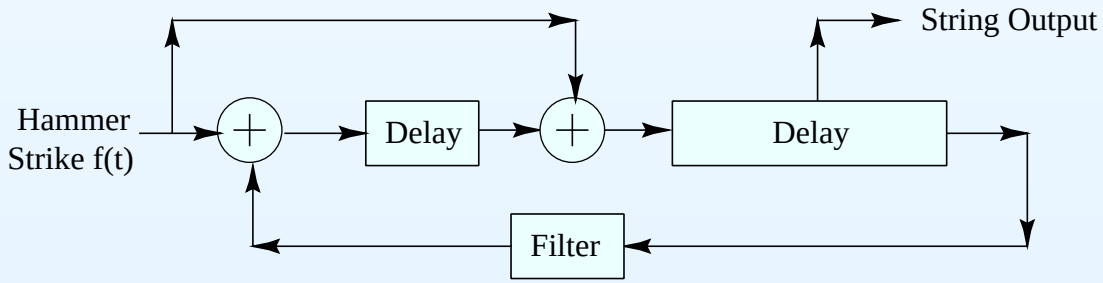


String Model Excited Externally at One Point



"Waveguide Formulation"

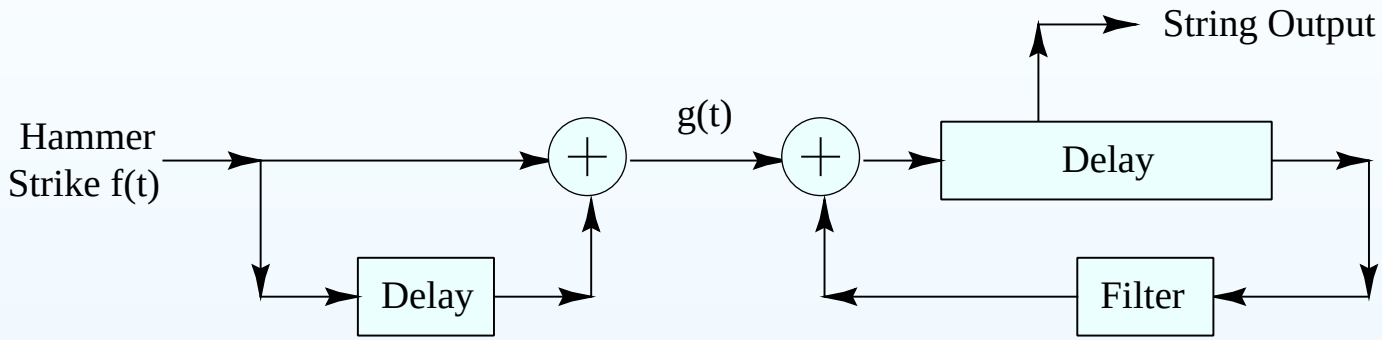
Equivalent System by Delay Consolidation:



Finally, we "pull out" the comb-filter component:

Pick-Position Comb Filter

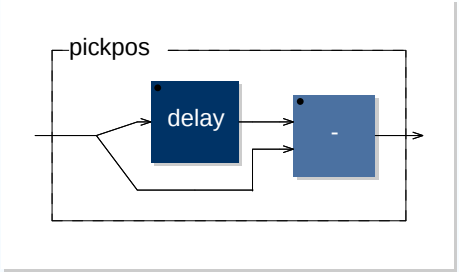
Equivalent System: Comb Filter Factored Out



$$H(z) = z^{-N}\frac{1+z^{-2M}}{1-z^{-(2M+2N)}} = \left(1+z^{-2M}\right)\frac{z^{-N}}{1-z^{-(2M+2N)}}$$

- *Excitation Position* controlled by left delay-line length
- *Fundamental Frequency* controlled by right delay-line length
- Derived originally (1982) by transfer-function factorization

Pick-Position Comb Filter in Faust




```
beta = hslider("pick_position",
                0.13, 0.02, 0.5, 0.01);

P = SR/freq; // fundamental period in samples
Pmax = 4096; // maximum P (delay-line allocation)
ppdel = beta*P; // pick-position delay

ffcombfilter(maxdel,del,g) =
              _ <: delay(maxdel,del) : *(g) : + ;

pickposfilter = ffcombfilter(Pmax,ppdel,-1);
```

EKS Damping Filter

EKS Damping Filter

Original:

$$H_d(z) = (1 - S) + Sz^{-1}, \quad S \in \left[\frac{1}{2}, 1\right]$$

Later: Symmetric FIR (delay always one sample):

$$H_d(z) = h_1 + h_0 z^{-1} + h_1 z^{-2} = z^{-1}\left[h_0 + h_1(z + z^{-1})\right].$$

Faust Implementation:

```
t60 = hslider("decaytime T60", 4, 0, 10, 0.01);
B = hslider("brightness", 0.5, 0, 1, 0.01); // 0-1

rho = pow(0.001,1.0/(freq*t60));
h0 = (1.0 + B)/2;
h1 = (1.0 - B)/4;
dampingfilter(x) = rho * (h0 * x' + h1*(x+x''));
```

EKS Tuning Filter

Original EKS Tuning Filter

$$H_\eta(z) = -\frac{\eta - z^{-1}}{1 - \eta z^{-1}}$$

where the phase delay $\Delta_\eta(\omega)$ has dc limit

$$\Delta_\eta(0) = \frac{1+\eta}{1-\eta}$$

$$\Rightarrow \quad \eta \approx \frac{\Delta - 1}{\Delta + 1}$$

where $\Delta =$ desired delay

Faust Implementation:

```
fdelay1a(n,d,x)  = delay(n,id,x) : tf1(eta,1,eta);
```

Tuning by Lagrange Interpolation (HUT)

Transfer function:

$$H_\Delta(z) = \sum_{n=0}^{N} h_\Delta(n) z^{-n}$$

Coefficient formula:

$$h_\Delta(n) = \prod_{\substack{k=0 \\ k \neq n}}^{N} \frac{\Delta - k}{n - k}$$

Useful `maxima` function:

```
hd(n,d,N) :=
  product(if k=n then 1 else (d-k)/(n-k),k,0,N);
```

Lagrange Interpolation in Faust

Second-order case:

```
fdelay2(n,d,x) = delay(n,id,x)*(1-fd)*(2-fd)/2
               + delay(n,id+1,x)*(2-fd)*fd
               + delay(n,id+2,x)*(fd-1)*fd/2
with {
  o = 0.49999; // center delay-range in polynomial
  dmo = d - o; // assumed nonnegative
  id = int(dmo);
  fd = o + frac(dmo);
};
```

More robust than allpass under rapidly time-varying conditions

Testing Lagrange Interpolation in Faust

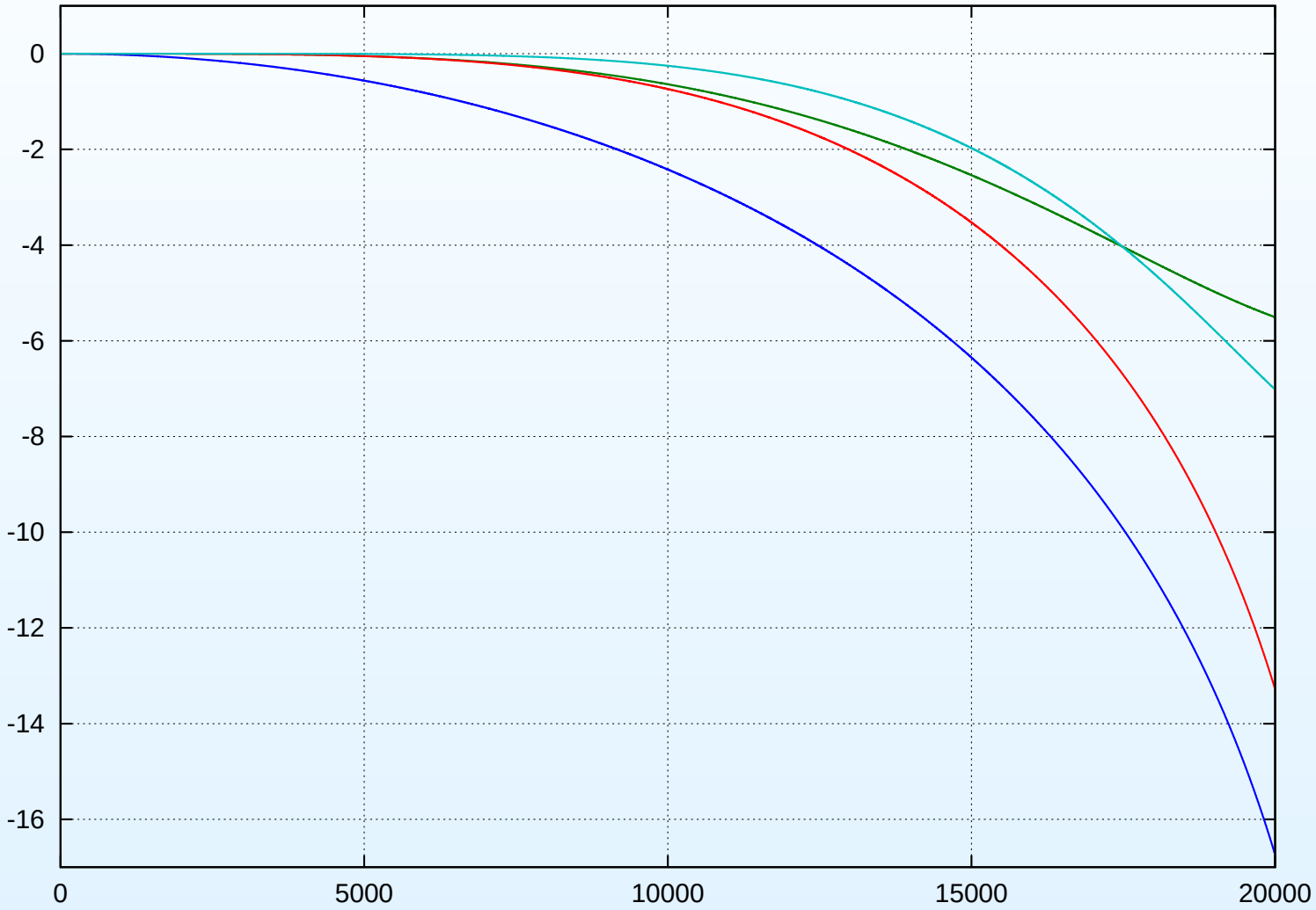

```
> cat tlagrange1-4.dsp
```

```
import("filter.lib");
N = 16;  d = 1.5;  impulse = 1-1';
process  = impulse <: (fdelay1(N,d),
                       fdelay2(N,d),
                       fdelay3(N,d),
                       fdelay4(N,d));
```

```
> faust2octave tlagrange1-4.dsp
octave:1> \
 plot(20*log10(abs((fft(faustout,1024)(1:512,:)))))
```

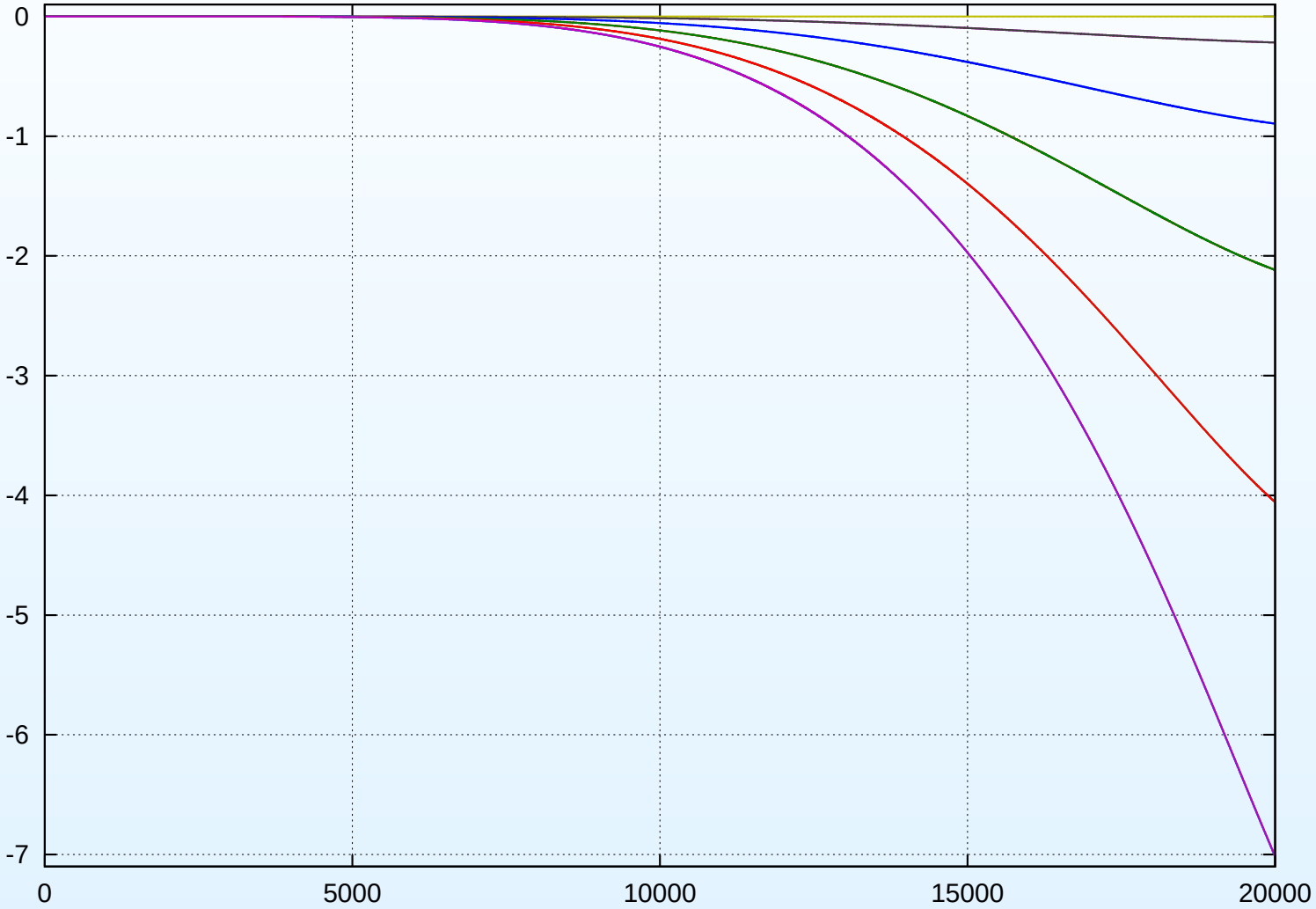
Lagrange Interpolation Orders 1-4

Amplitude Responses for requested delay = 1.5 samples:



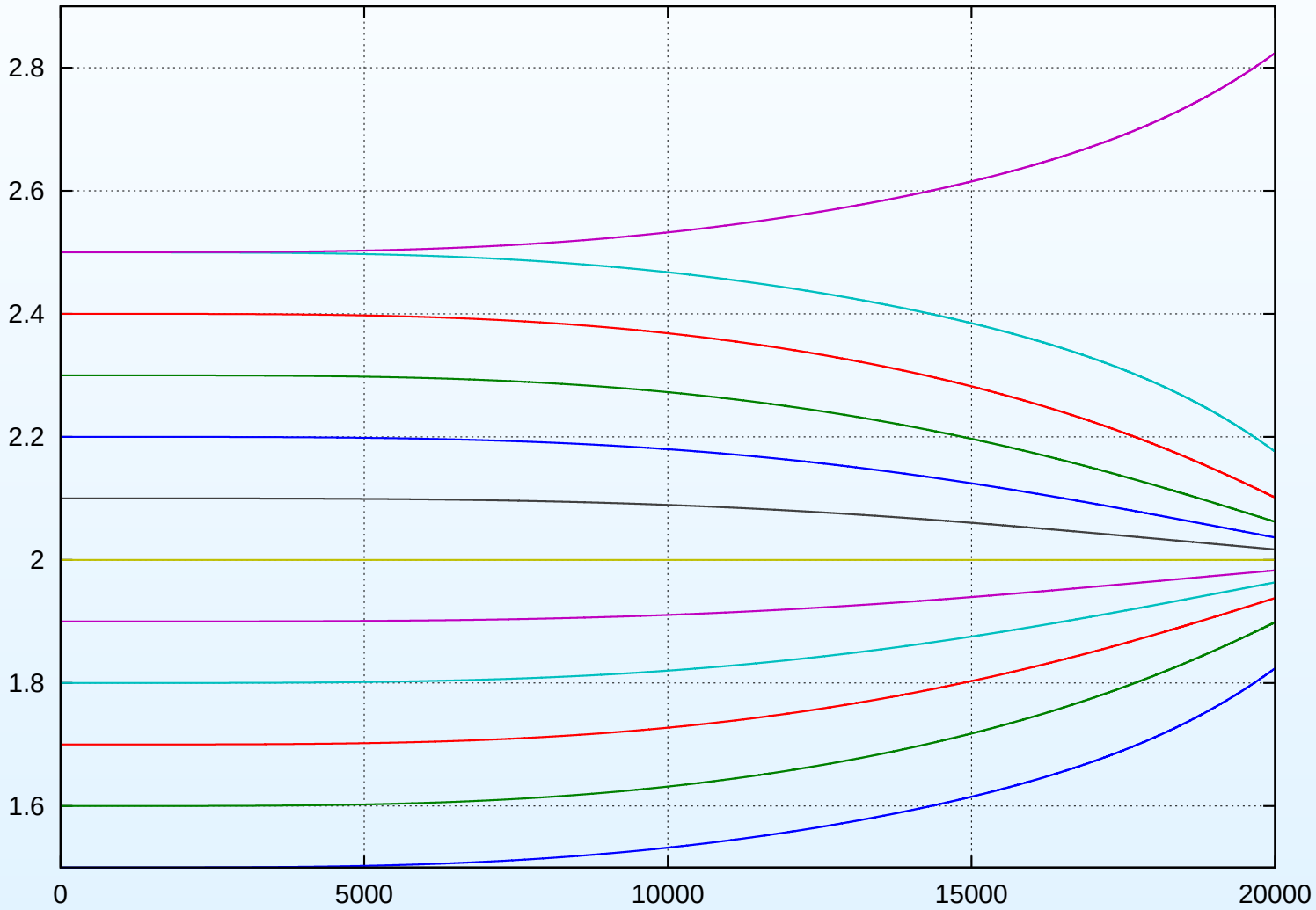
Lagrange Interpolation Order 4

Amplitude Responses for a range of requested delays [1.5 : 0.1 : 2.5]:



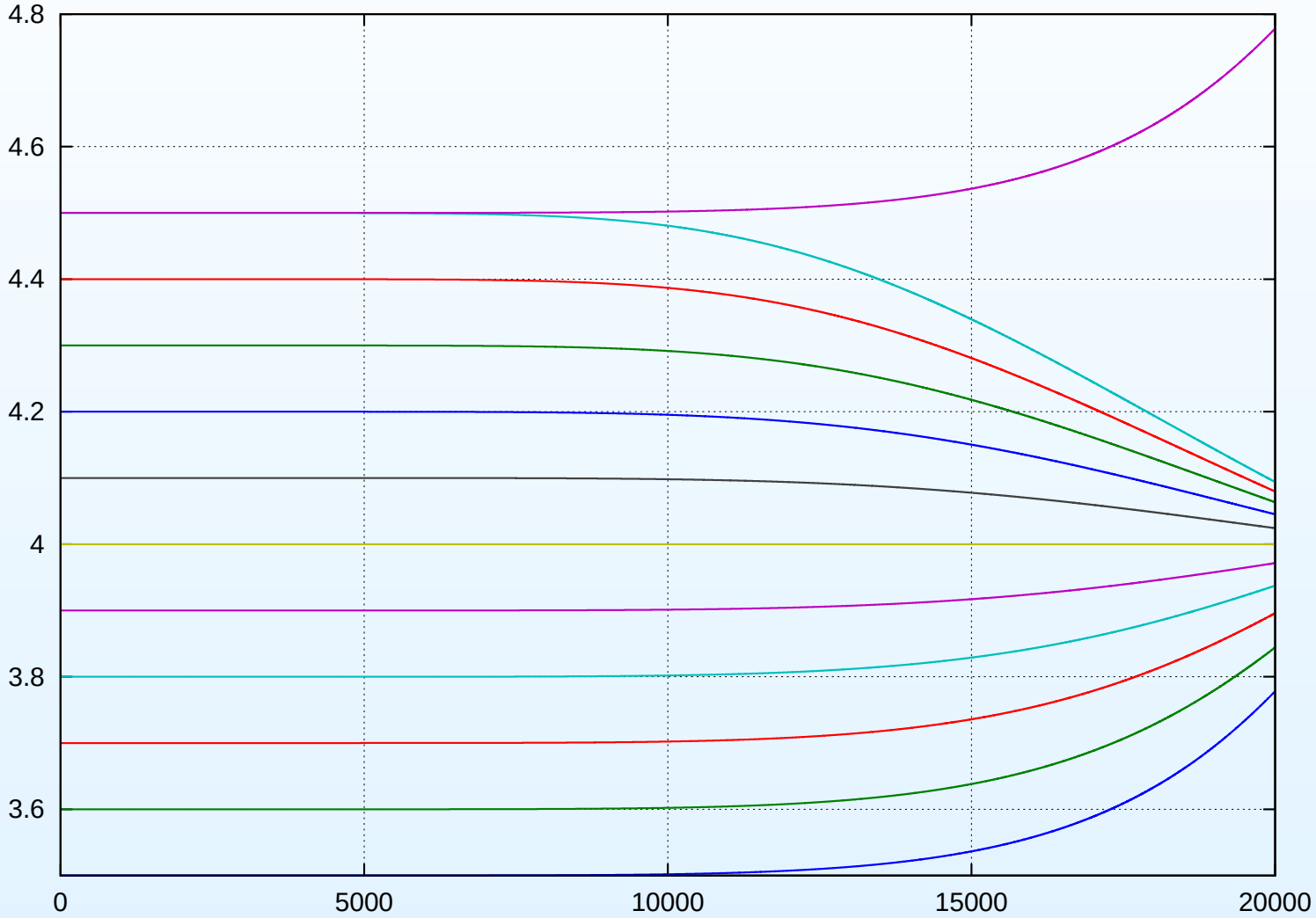
Lagrange Interpolation Order 4

Phase Delays for a range of requested delays [1.5 : 0.1 : 2.5, 2.5001]:



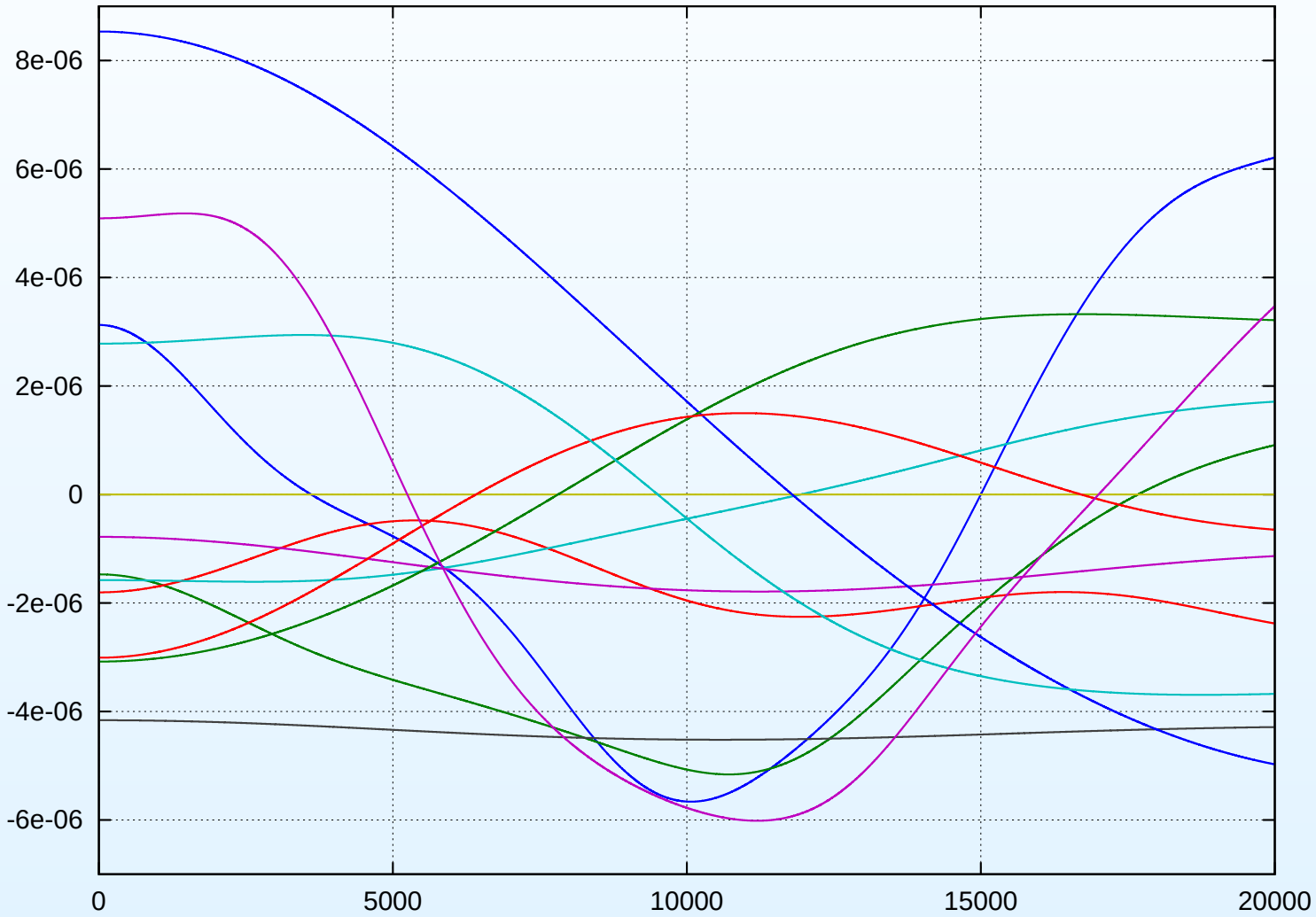
Thiran Allpass Interpolation Order 4

Phase Delays for a range of requested delays [1.5 : 0.1 : 2.5, 2.5001]:



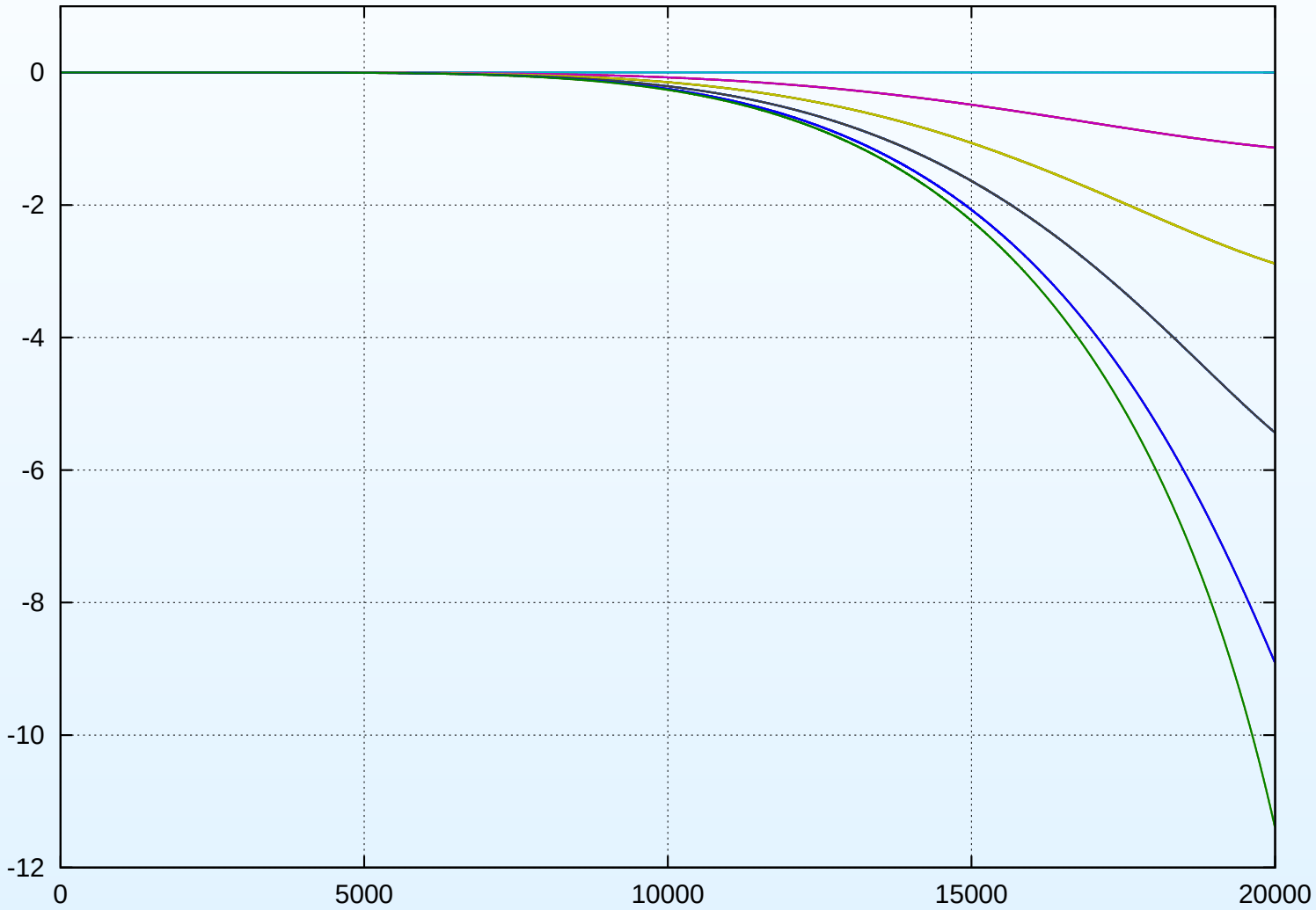
Thiran Allpass Interpolation Order 4

Amplitude Responses for a range of requested delays [1.5 : 0.1 : 2.5]:



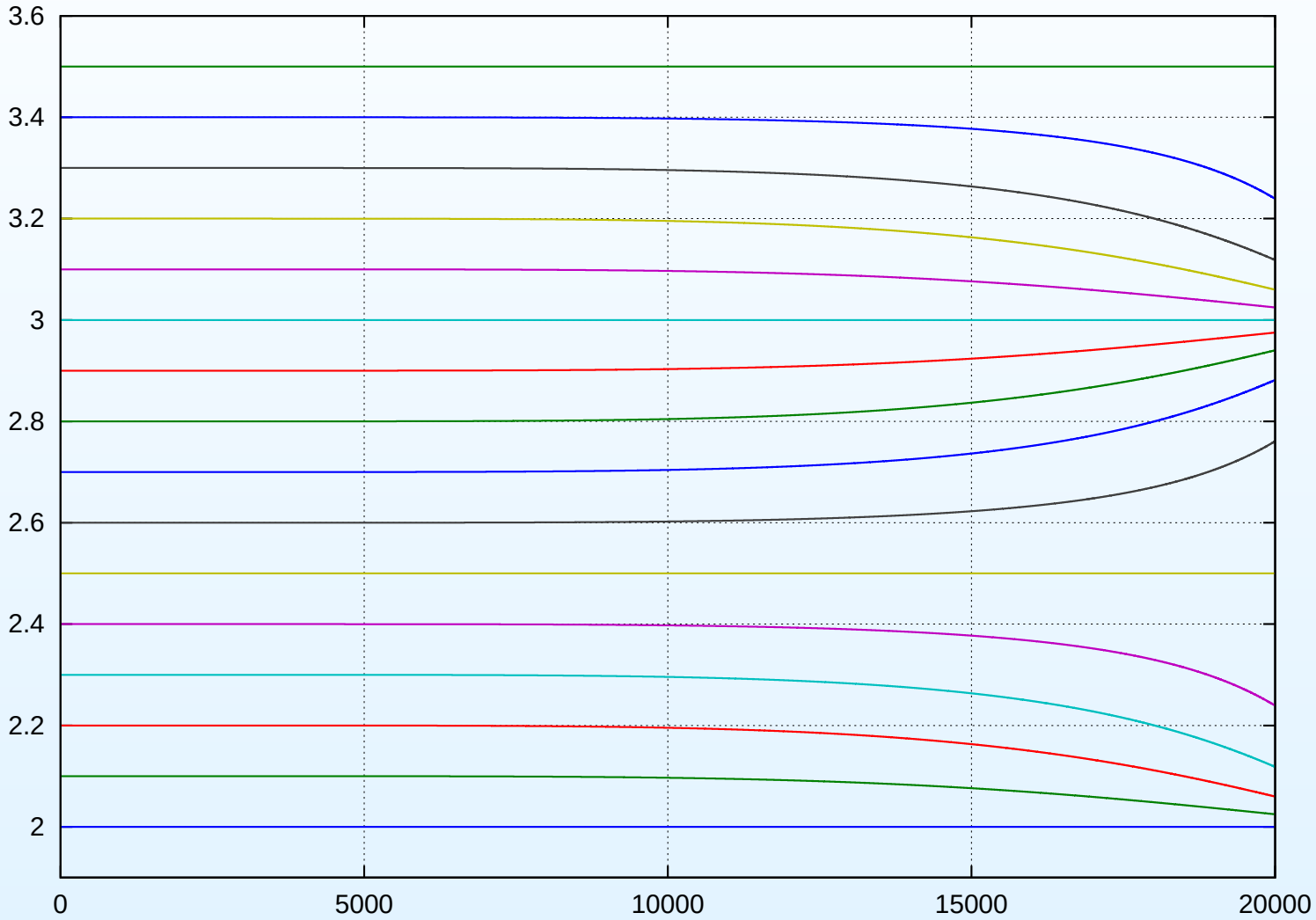
Lagrange Interpolation Order 5

Amplitude Responses for a range of requested delays [2 : 0.1 : 3.5]:



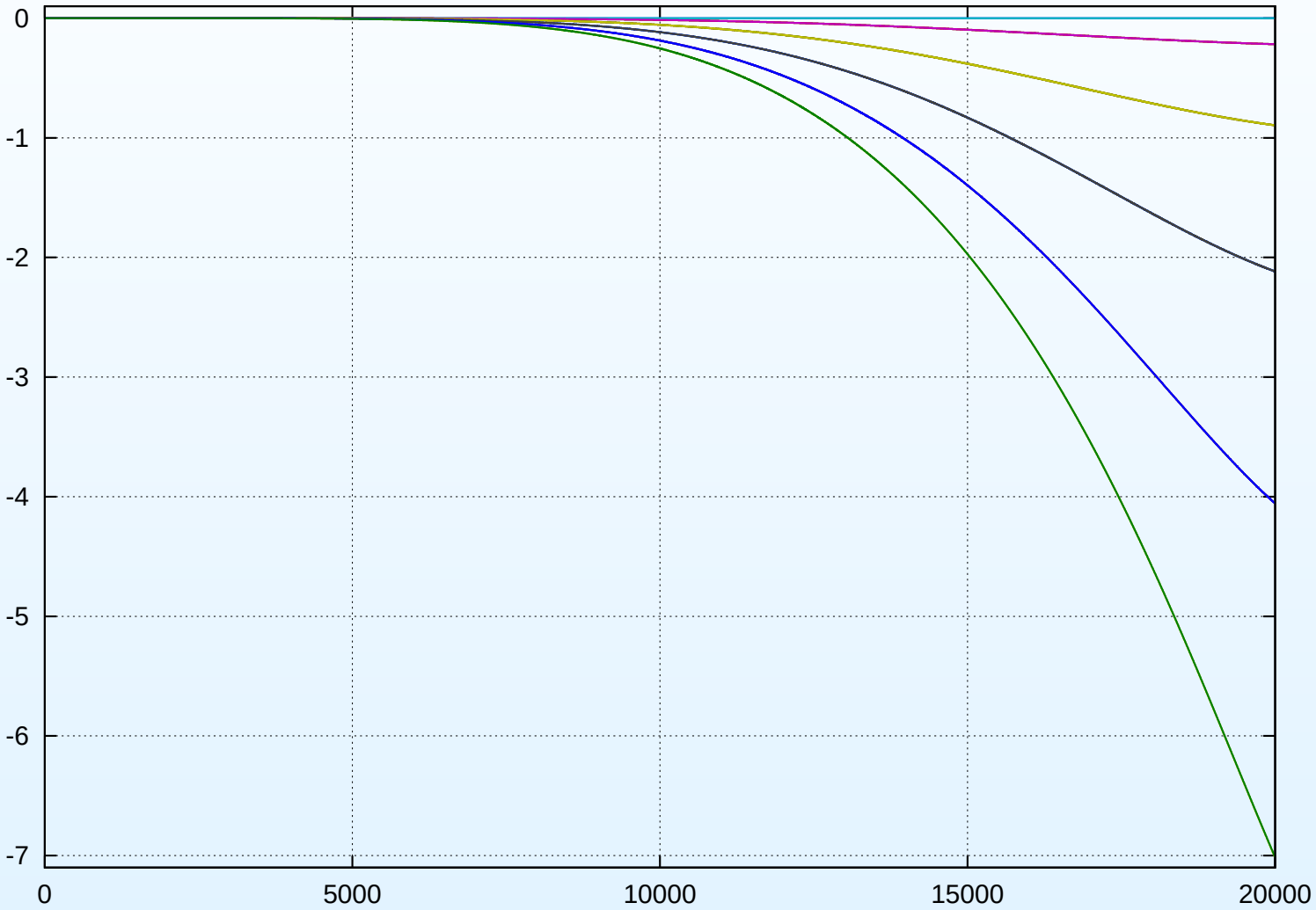
Lagrange Interpolation Order 5

Phase Delays for a range of requested delays [2 : 0.1 : 3.5]:



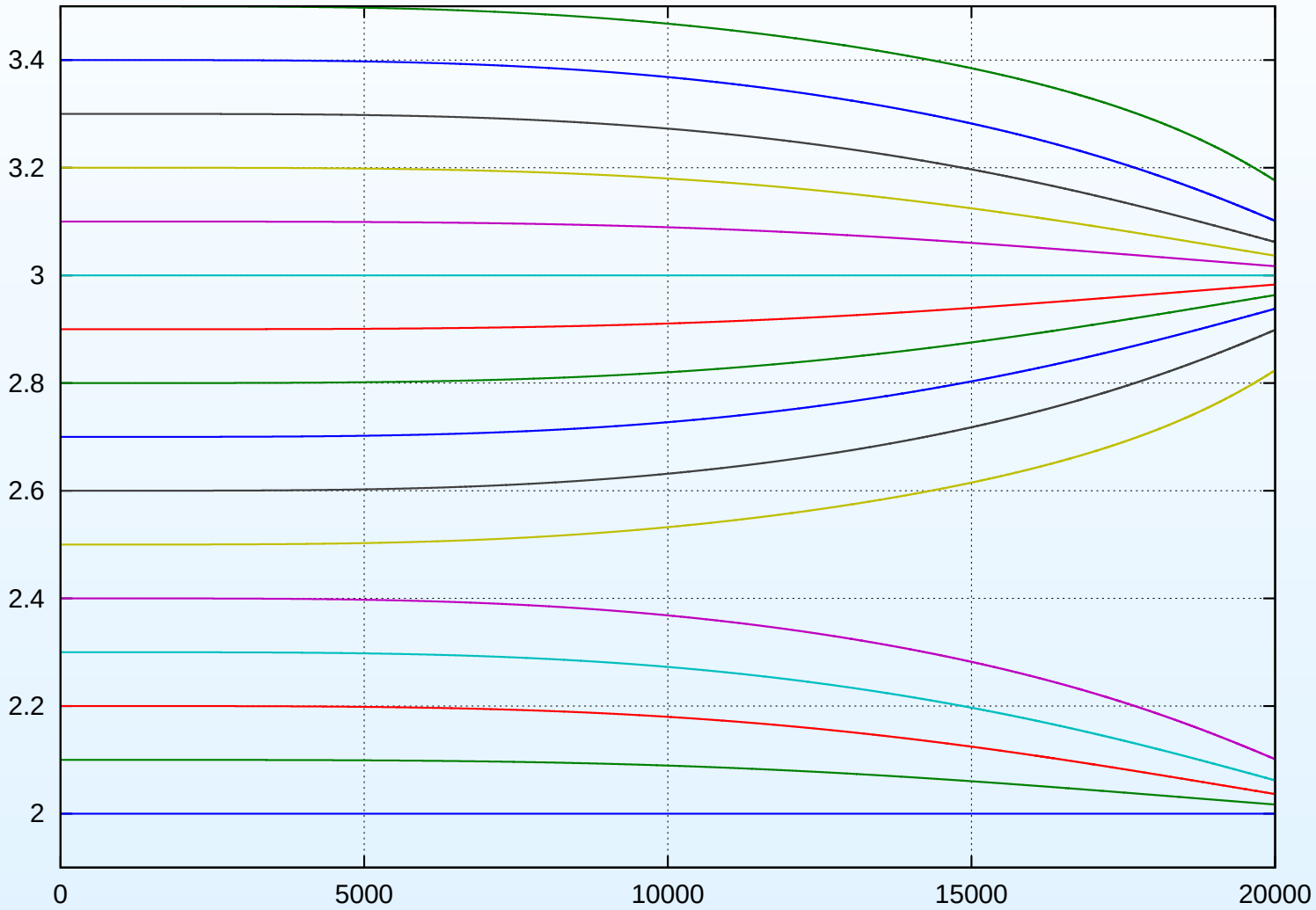
Lagrange Interpolation Order 4

Amplitude Responses for a range of requested delays [2 : 0.1 : 3.5]:



Lagrange Interpolation Order 4

Phase Delays for a range of requested delays [2 : 0.1 : 3.5]:



EKS Dynamic Level Filter

Dynamic Level Filter

In real strings, the *spectral centroid* typically rises as plucking/striking becomes more energetic.
The EKS dynamic-level lowpass filter

$$H_{L,\omega_1}(z) \;=\; \frac{1 - R_L(\omega_1)}{1 - R_L(\omega_1)z^{-1}}$$

qualitatively models this phenomenon

- "Spectral modeling filter"
- Not needed in a true physical model
- Useful with noise-burst string excitation (KS, EKS)

Dynamic Level Filter

In real strings, the *spectral centroid* typically rises as plucking/striking becomes more energetic.
The EKS dynamic-level lowpass filter

$$H_{L,\omega_1}(z) \;=\; \frac{1 - R_L(\omega_1)}{1 - R_L(\omega_1)z^{-1}}$$

qualitatively models this phenomenon

- "Spectral modeling filter"
- Not needed in a true physical model
- Useful with noise-burst string excitation (KS, EKS)

Dynamic Level Filter

In real strings, the *spectral centroid* typically rises as plucking/striking becomes more energetic.
The EKS dynamic-level lowpass filter

$$H_{L,\omega_1}(z) \;=\; \frac{1 - R_L(\omega_1)}{1 - R_L(\omega_1)z^{-1}}$$

qualitatively models this phenomenon

- "Spectral modeling filter"
- Not needed in a true physical model
- Useful with noise-burst string excitation (KS, EKS)

Dynamic Level Filter

In real strings, the *spectral centroid* typically rises as plucking/striking becomes more energetic.
The EKS dynamic-level lowpass filter

$$H_{L,\omega_1}(z) \;=\; \frac{1 - R_L(\omega_1)}{1 - R_L(\omega_1)z^{-1}}$$

qualitatively models this phenomenon

- "Spectral modeling filter"
- Not needed in a true physical model
- Useful with noise-burst string excitation (KS, EKS)

DLF Derivation

$$\boxed{H_{L,\omega_1}(z) = L \cdot L_0(L) + (1-L) \cdot H_1(z)}$$

where $H_1(z) \approx$ *analog* 1st-order Butterworth lowpass:

$$H_1(s) \stackrel{\Delta}{=} \frac{\omega_1}{s+\omega_1}$$

- Unity dc gain
- -3 dB gain at $s = j\omega_1$
- $\omega_1 = 2\pi f_1 =$ fundamental frequency (rad/sec)
- Rolls off -6 dB/octave for $\omega \gg \omega_1$.
- L ("dynamic level") = gain at Nyquist limit
- $L_0 =$ low-frequency attenuation:

$$L_0(L) = L^{1/3}$$

Bilinear Transform

$$s = \alpha \frac{1 - z^{-1}}{1 + z^{-1}}$$

$\Rightarrow$

$$H_1(z) = \frac{\omega_1}{\alpha \frac{1-z^{-1}}{1+z^{-1}} + \omega_1} = \frac{\omega_1}{\alpha + \omega_1} \cdot \frac{1 + z^{-1}}{1 - pz^{-1}}$$

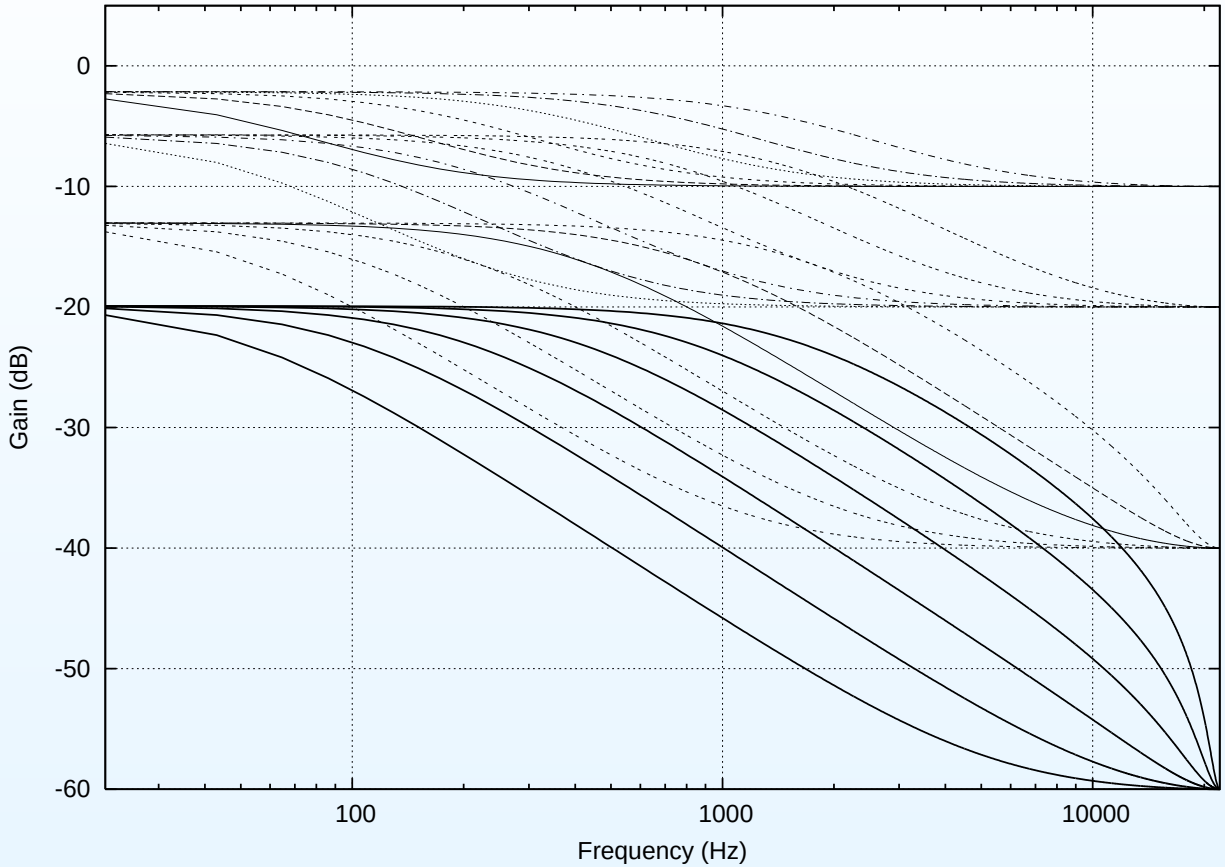
where

$$p = \frac{\alpha - \omega_1}{\alpha + \omega_1}$$

To map analog break frequency exactly:

$$\boxed{\alpha = \frac{\omega_1}{\tan\left(\frac{\omega_1 T}{2}\right)}}$$

Dynamic Level Filter for Various Levels



- $L = -60$ dB, -40 dB, -20 dB, and -10 dB
- DC level = 1/3 Nyquist-limit level in dB

Dynamic Level Lowpass in Faust

```
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);
};
```

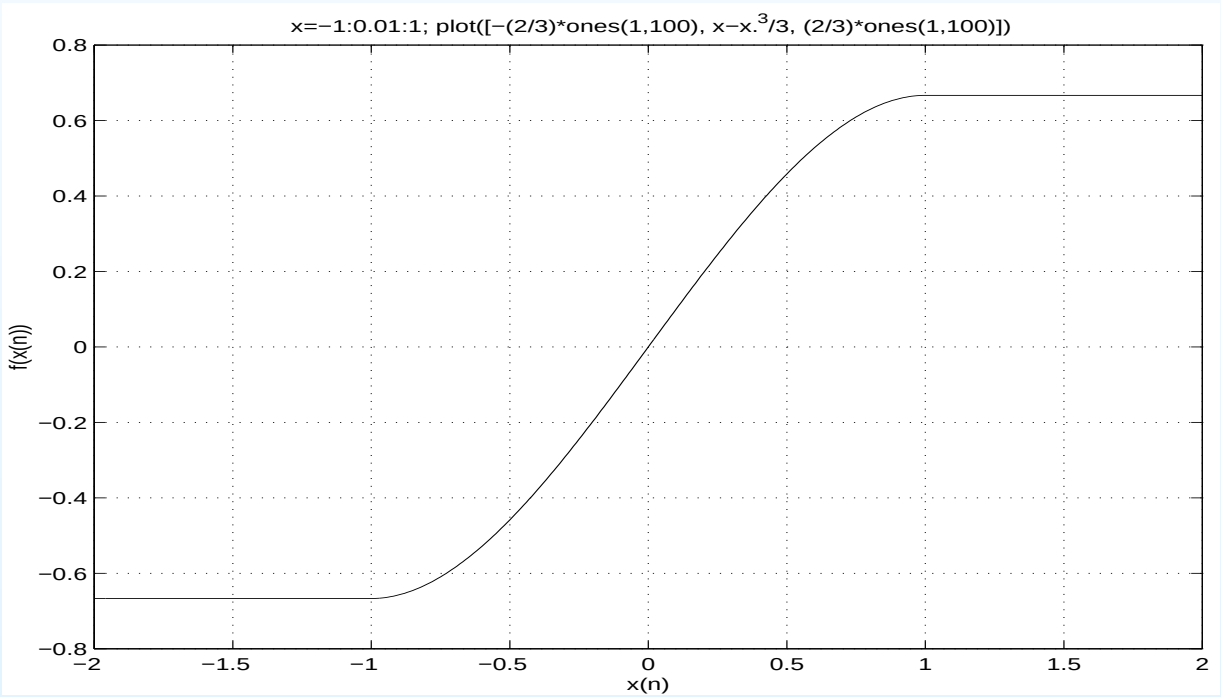
To intensify the effect, N_d units can be used in series, with the desired Nyquist-limit level divided by N_d for each section:

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

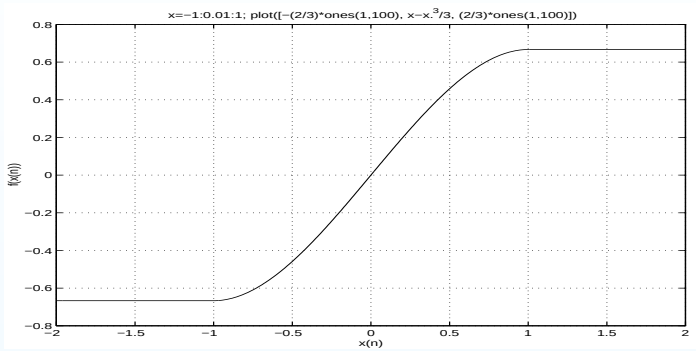
Overdrive and Feedback

Cubic Nonlinearity

$$f(x) = \begin{cases} -\frac{2}{3}, & x \leq -1 \\ x - \frac{x^3}{3}, & -1 \leq x \leq 1 \\ \frac{2}{3}, & x \geq 1 \end{cases}$$

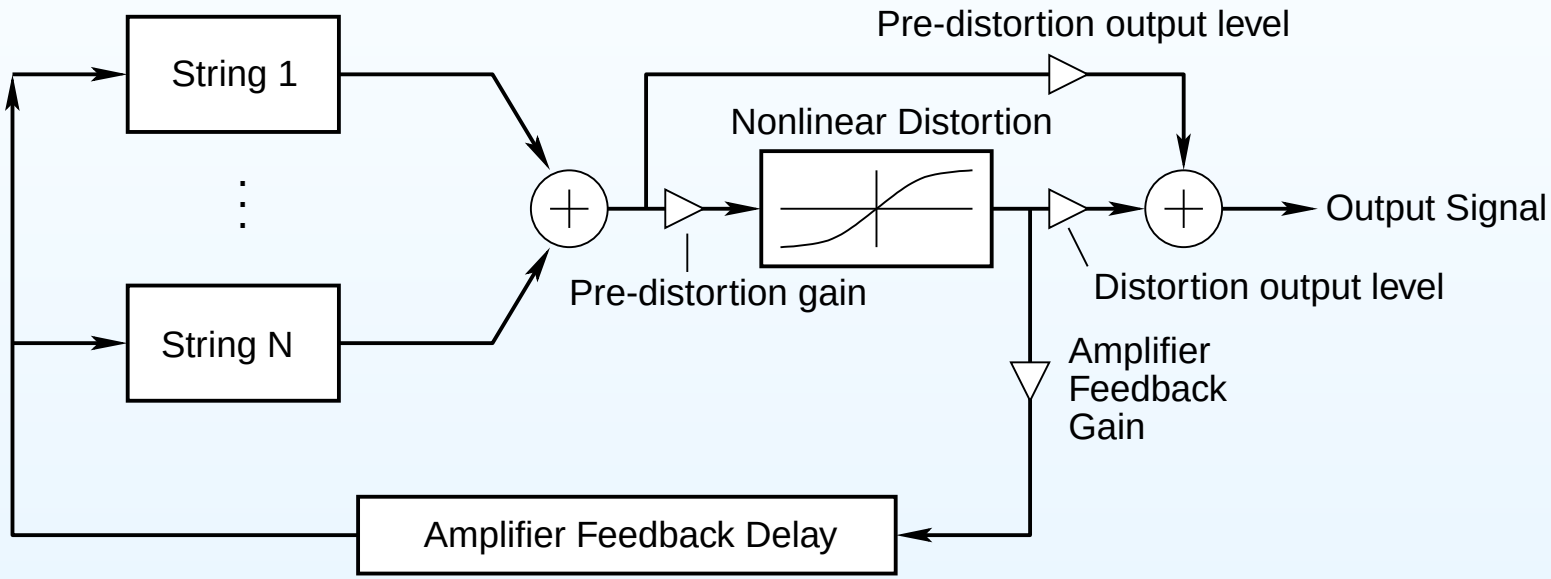


Clipping Cubic Nonlinearity in Faust (`effect.lib`)




```
cubicnl(drive,offset) =
    +(offset) : *(pregain) : clip(-1,1)
    : cubic : dcblocker
with {
    pregain = pow(10.0,2*drive);
    clip(lo,hi) = min(hi) : max(lo);
    cubic(x) = x - x*x*x/3;
};
```

Amplifier Overdrive and Feedback



Distortion output signal often further filtered by an *amplifier cabinet filter*, representing speaker cabinet, driver responses, etc.

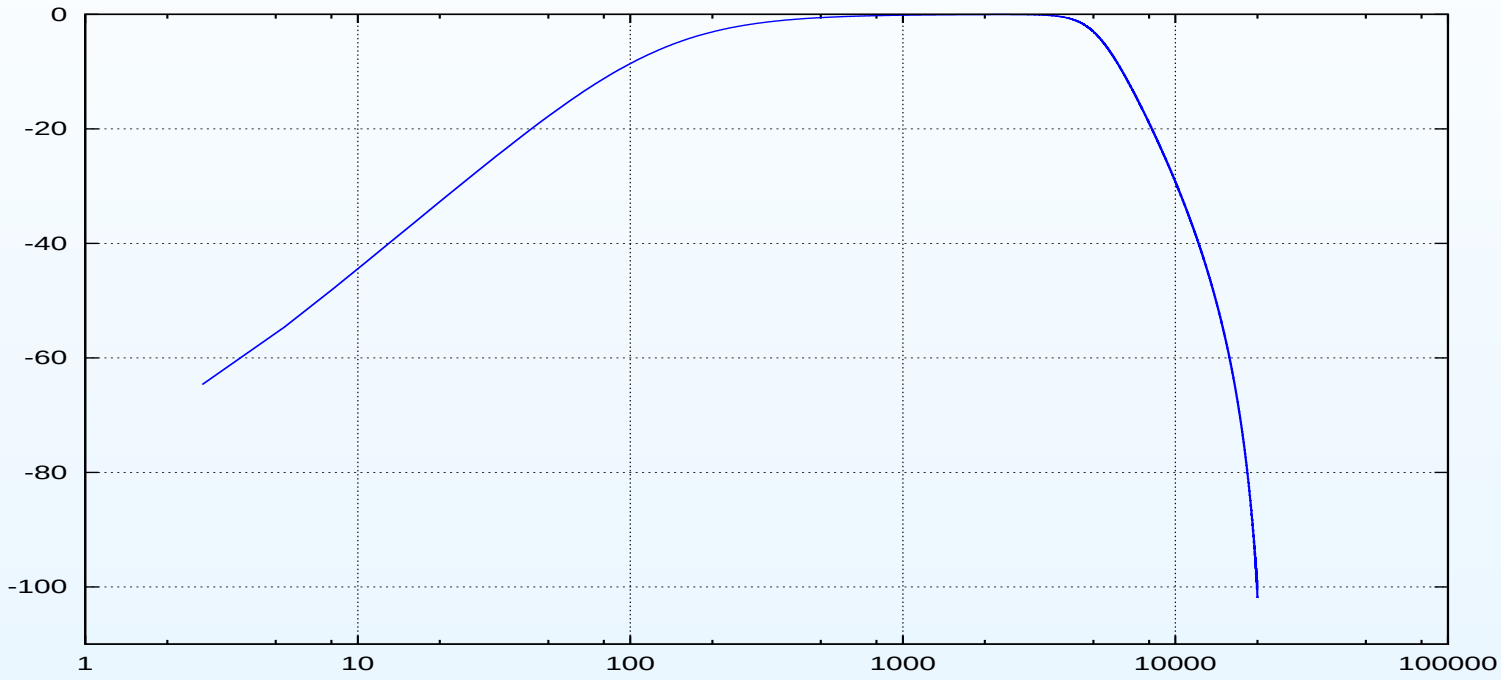
Distortion and Amplifier Feedback in Faust (`freeax.dsp`)

```
D  = hslider("distortion drive",0.629,0,10,0.01);

ampfbgain = hslider("Amp feedback gain",
                    0.001,0,0.01,0.001);
ampfbfres = hslider("Amp feedback 1st resonance Hz",
                    100,40,2000,1);

ampfbdelay = int(SR/ampfbfres); // no interpolation
ampfb = *(ampfbgain) : delay(4096,ampfbdelay);
...
process = filtered_excitation : (+ : stringloop
  : cubicnl(D,0) : speakerbp(sf1,sf2))
  ~ ampfb : *(level) <: _,_ : widthdelay : panner;
```

Speaker Bandpass

Speaker Bandpass

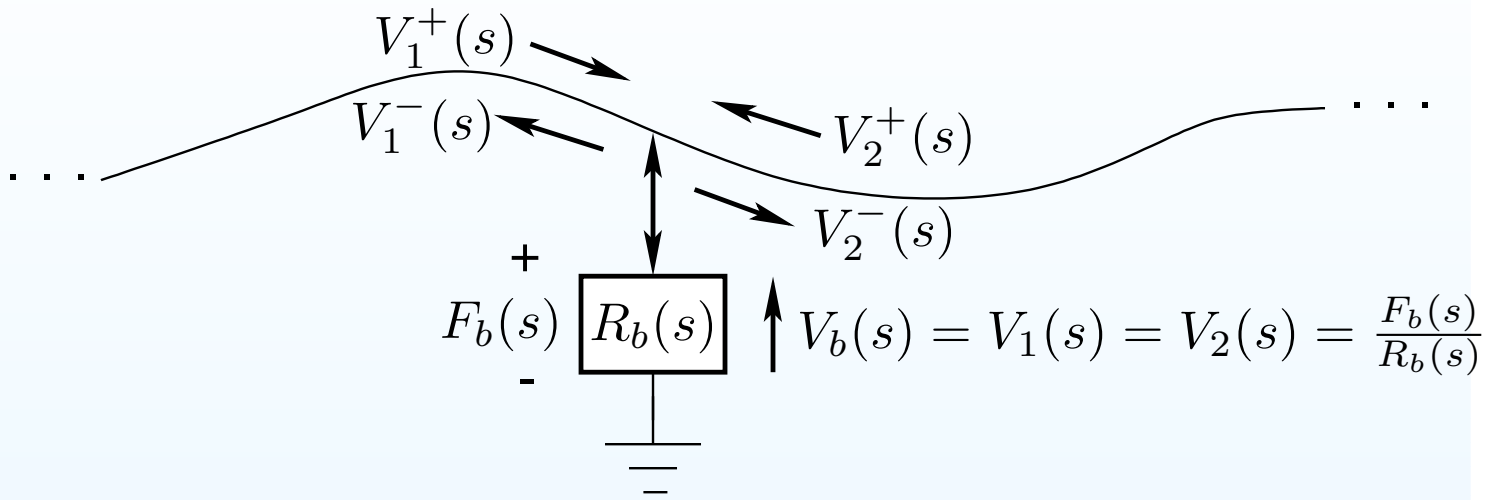
General “spectral window” of a Celestion 12¨ speaker:




```
import("filter.lib");
dcblockerat(130) : dcblockerat(130) : lowpass4(5000);
```

Coupled Strings

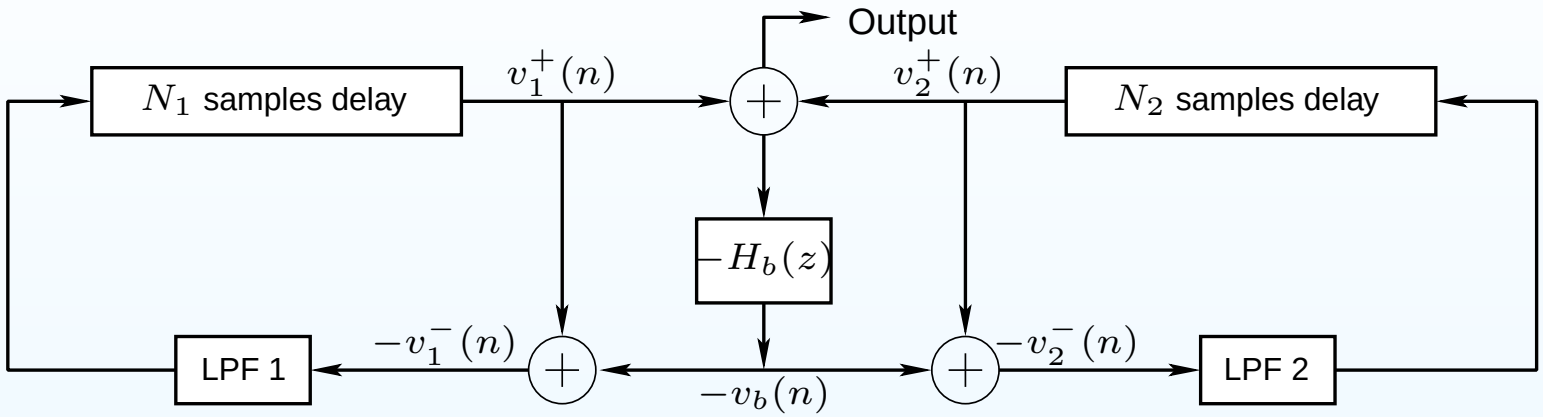
Two Ideal Coupled Strings at a Common Bridge Impedance



$$
\begin{aligned}
v_1^-(t) &= v_b(t) - v_1^+(t) \\
v_2^-(t) &= v_b(t) - v_2^+(t)
\end{aligned}
$$

$$
\begin{aligned}
V_b(s) &= H_b(s)[R_1 V_1^+(s) + R_2 V_2^+(s)], \text{ where} \\
R_i &= \text{wave impedance of string } i, \text{ and} \\
H_b(s) &\triangleq \frac{2}{R_b(s) + R_1 + R_2}
\end{aligned}
$$

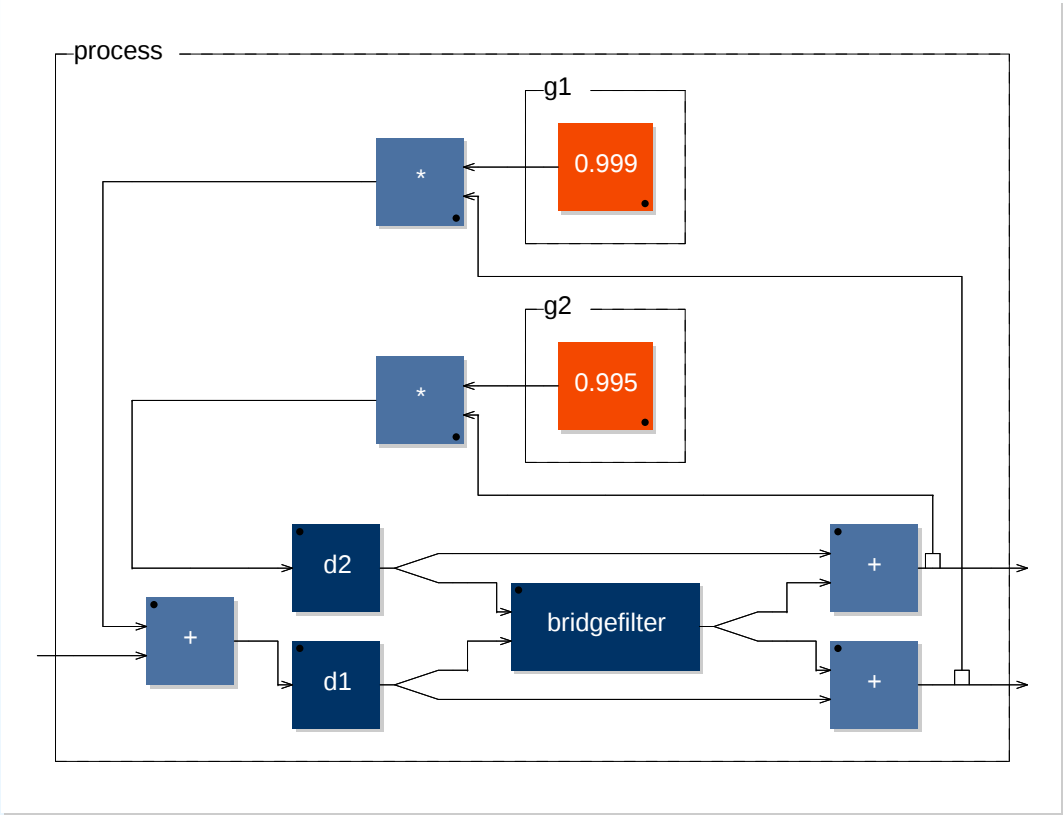
Digital Waveguide Model



Faust Code

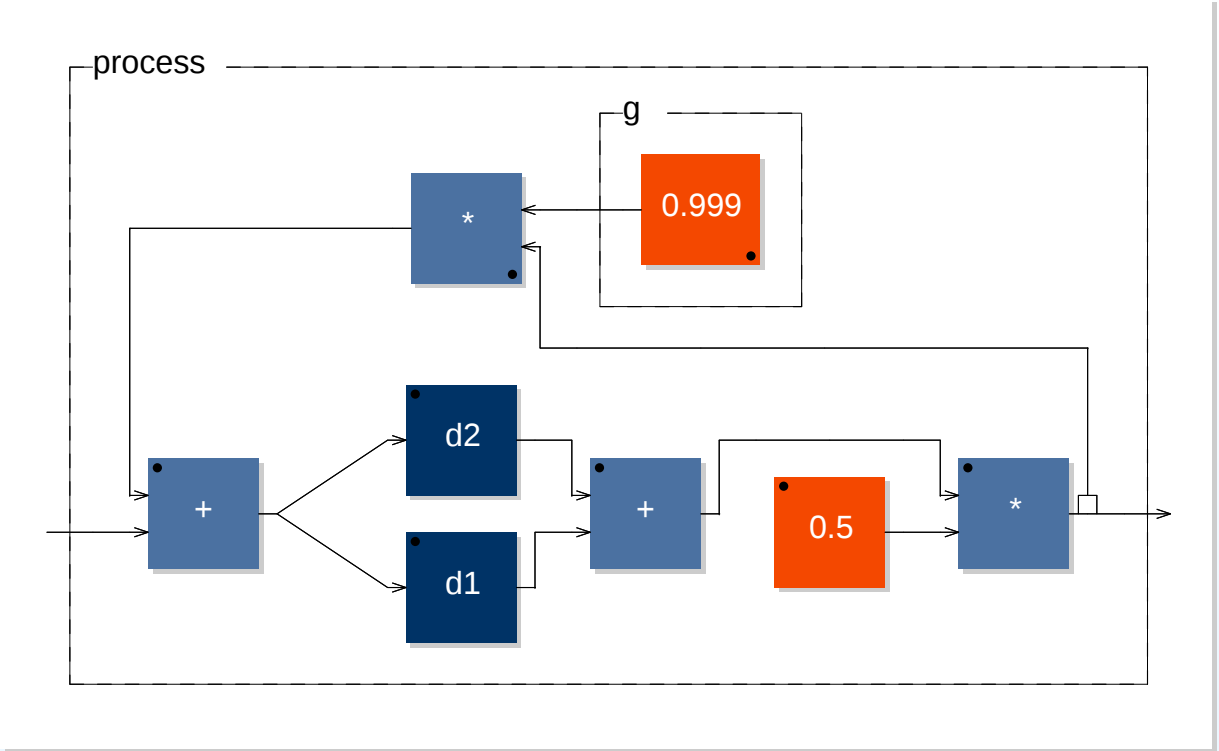
```
// Detuning of coupled planes of vibration:
detune = hslider("detuning percent", 1, 0, 10, 0.1);

d1 = fdelay1(Pmax, P-2); // P = period in samples
d2 = delay(Pmax, P*(1.0 - 0.01*detune)-2);
stringloop = coupledstrings(d1,d2);
```

Coupled Strings in Faust




```
d1 = fdelay1(Pmax, P-2); // P = period in samples
d2 = delay(Pmax, P*(1.0 - 0.01*detune)-2);
stringloop = ( _,+ : ((d2 <: _,_),(d1 <: _,_))
     : (_, (bridgefilter <: _,_) ,_)
     : +,+) ~ (*(g2),*(g1)) ;
```

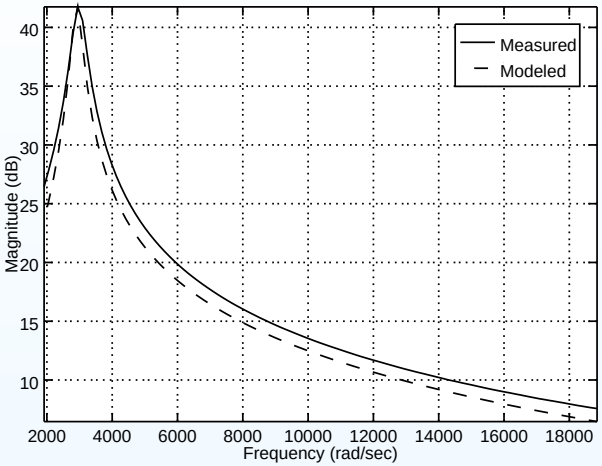
Simplified String Coupling



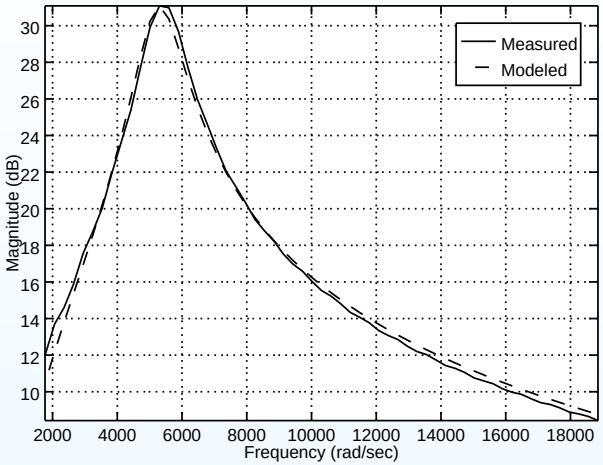

```
stringloop = (+ <: d1,d2 : + : *(0.5))
           ~ dampingfilter;
```

Wah Pedal

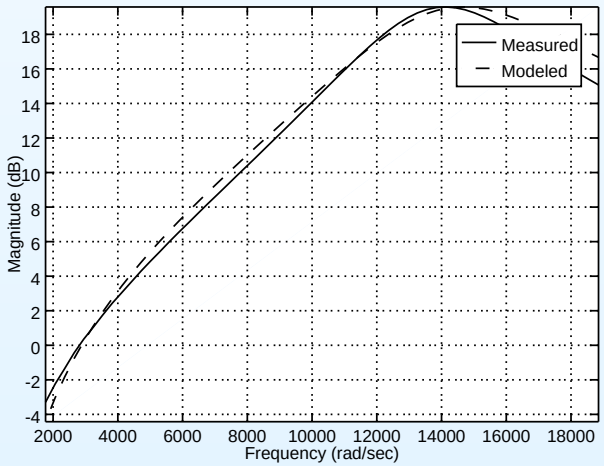
CryBaby Measurements at Three Pedal Angles



(a) Rocked Back Full



(b) Middle Angle

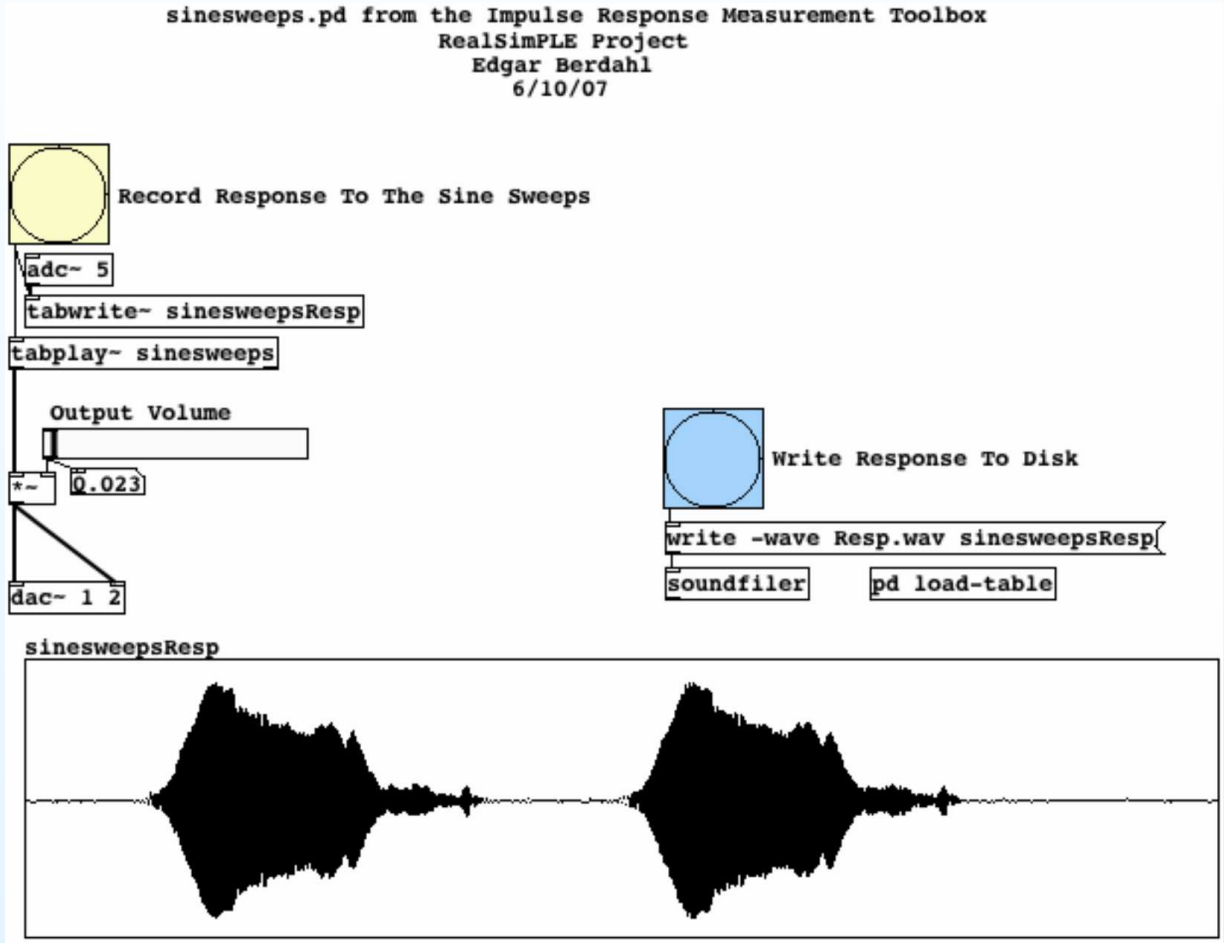


(c) Rocked Forward Full

RealSimple Transfer Function Measurement Toolbox

Measures acoustic impulse responses using Pd and Octave:

`http://ccrma.stanford.edu/realsimple/imp_meas/`



`sinesweeps.pd`

Octave Code for CryBaby Q and Resonance Measurement

```
f1 = 40; f2 = 10000;     % used in filename
f1z = 300; f2z = 3000;   % zoom-in range
del = [3000 2000 2000]; % system delay (samples)
dur = [2048 1024 1024]; % impulse-response duration
dir = sprintf('wah-2sec-%dHz-%dkHz',f1,f2/1000);
Q = zeros(1,3);
wp = zeros(1,3);
for i=1:3
  ifn = sprintf('%s/wah%dImpResp.wav',dir,i-1);
  [wahir,fs] = wavread(ifn);
  wi = wahir(del(i)+1:del(i)+dur(i));
  [Qi,wpi,Hp,Hd,w] = invfreqsmethod(wi,f1z,f2z,fs);
  Q(i) = Qi; wp(i) = wpi;
  disp('PAUSING - RETURN to continue'); pause;
end
Q                  % print out estimated Q values
fp = wp/(2*pi)  % and estimated pole frequencies
```

CryBaby Q and Resonance Measurement Results

Estimated Q values:

$$Q = [9.4, 4.0, 1.9]$$

Estimated pole frequencies:

$$f_p = [464, 838, 2252] \quad \text{Hz}$$

Analog CryBaby model ($\omega_p = 2\pi f_p$):

$$H(s) = \frac{s - \xi}{\left(\frac{s}{\omega_p}\right)^2 + \frac{2}{Q}\left(\frac{s}{\omega_p}\right) + 1}$$

Digital model derived using $z = \exp(sT) \approx 1 + sT$
(low-frequency resonance assumed)

Octave function `invfreqsmethod`

```
function [Q,wp] = invfreqsmethod(h,f1,f2,fs);
%
% INVFREQSMETHOD - use invfreqs to estimate Q and resonance
%                  frequency from a resonator impulse response
% USAGE:
%       [Q,wp] = invfreqsmethod(h,f1,f2,fs);
% where
%
% h  = impulse response (power of 2 length preferred)
% f1 = lowest frequency of interest (Hz)
% f2 = highest frequency of interest (Hz)
% fs = sampling rate (Hz)
% Q  = estimated resonator Quality factor
% wp = estimated pole frequency (rad/sec)
```

Octave function `invfreqsmethod`, continued

```
h = minphaseir(h);
H = fft(h);
...
[Bh,Ah] = invfreqs(H(zoom),w,2,2,1 ./w.^2);

% Denominator to canonical form
%   A(s) = s^2 + (wp/Q) s + wp^2:

Ahn = Ah/Ah(1);
wp = sqrt(Ahn(3));
Q = wp/Ahn(2);
...
```

Octave function `minphaseir`

```
function [hmp] = minphaseir(h)
%
% MINPHASEIR - Convert a real impulse response to its
%              minimum phase counterpart
% USAGE:
%       [hmp] = minphaseir(h)
% where
%
% h   = impulse response (any length - will be zero-padded)
% hmp = min-phase impulse response (at zero-padded length)
```

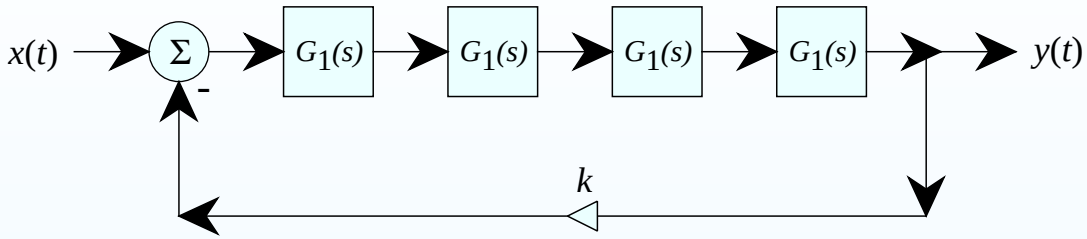
Octave function `minphaseir`, continued

```
nh = length(h);
nfft = 2^nextpow2(5*nh);
Hzp = fft(h,nfft);
Hmpzp = exp(fft(fold(ifft(log(clipdb(Hzp,-100))))));
hmpzp = ifft(Hmpzp);
hmp = real(hmpzp(1:nh));
```

Faust Digital CryBaby (effect.lib)

```
crybaby(wah) = *(gs(s)) : tf2(1,-1,0,a1s(s),a2s(s))
with { // wah = pedal angle in [0,1]
  s = 0.999; // smoothing parameter (pole location)
  Q  = pow(2.0,(2.0*(1.0-wah)+1.0)); // Resonance Q,
  fr = 450.0*pow(2.0,2.3*wah);    // tuning, and
  g  = 0.1*pow(4.0,wah);          // gain (optional)

  // Biquad fit using z = exp(s T) -> 1 + sT
  frn = fr/SR; // 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

  a1s(s) = a1 : smooth(s); // "dezippering"
  a2s(s) = a2 : smooth(s);
  gs(s) =  g  : smooth(s);
};
```

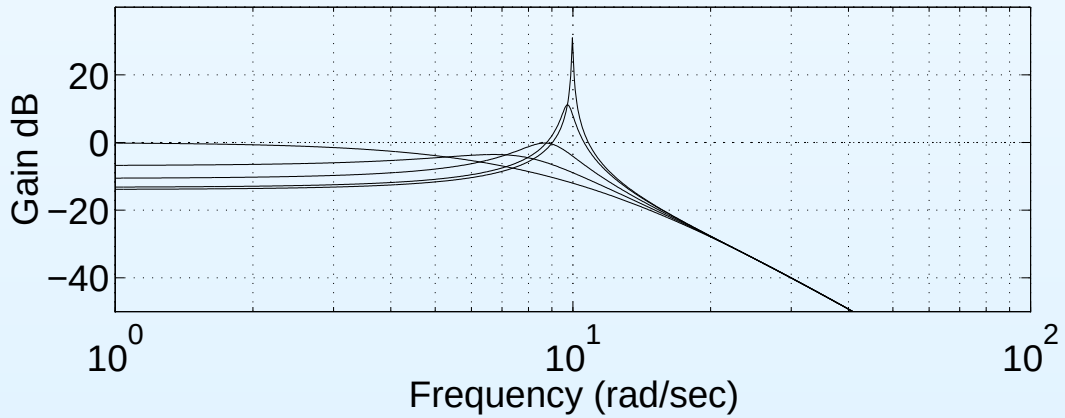
Moog VCF



$$G_1(s) = \frac{1}{1 + s/\omega_c}$$

Controls

- Pole location $s = -\omega_c$: controls *cut-off frequency*
- Feedback gain $k \leq 4$: controls *resonance*



Moog VCF in Faust (`effect.lib`)

```
onepole(p) =  *(1.0-p) : + ~ *(p);
moogvcf(mk,p) =
        (+ : onepole(p) : onepole(p)
           : onepole(p) : onepole(p)) ~ *(mk);
```

4th Order Wah Effect:

```
wah4(fr) = moogvcf(-3.8,pole(fr))
with { pole(fr) = 1.0-fr*2.0*PI/SR; };
```

$Q = 3.8$ can be changed as preferred

Faust Signal Processing Libraries

Faust Signal Processing Libraries

The following Faust libraries were developed in the course of this work:

- `filter.lib` - digital filters of various types
- `effect.lib` - digital audio effects
- `osc.lib` - filter-based oscillators (sine and sawtooth)

filter.lib

Simple first-order filters (mainly for readability):

```
zero(z) =  _ <: _,mem : _,*(z) : -;
pole(p) = + ~ *(p);
integrator = + ~ _ ;
tau2pole(tau) = exp(-1.0/(tau*SR));
smooth(s) = *(1.0 - s) : + ~ *(s);
dcblocker = zero(1) : pole(0.995);
dcblockerat(fb); // pole set for desired "break frequency"
```

Comb Filters:

```
ffcombfilter(maxdel,del,g) = _ <: delay(maxdel,del) : *(g): +;
fbcombfilter(maxdel,idel,g) = (+ : delay(maxdel,idel)) ~ *(g);
```

`filter.lib`, **continued**

Elementary Transfer Functions in Direct Form:

```
tf1(b0,b1,a1) = _ <: *(b0), (mem : *(b1)) :> + ~ *(0-a1);
tf2(b0,b1,b2,a1,a2) = sub ~ conv2(a1,a2) : conv3(b0,b1,b2) ...
tf3(b0,b1,b2,b3,a1,a2,a3) = sub ~ conv3(a1,a2,a3) : conv4( ...
tf4(b0,b1,b2,b3,b4,a1,a2,a3,a4) = sub ~ conv4(a1,a2,a3,a4) ...
```

Numerically Robust Quadrature Resonators (for Oscillators):

```
nlf2(f,r,x);// 2nd-order normalized ladder resonator
wgr(f,r,x); // Transformer-normalized waveguide resonator
```

filter.lib, **continued**

Digital filters specified by their ANALOG transfer functions:

```
tf1s(b0,a0,w1); // 1st-order section H(s) = b0 / (s + a0)
tf2s(b1,b0,a1,a0,w1); // 2nd-order section (biquad)
```

Variable cut-off Butterworth lowpass filters:

```
lowpass1(fc) = tf1s(1,1,2*PI*fc);                    // 1st-order
lowpass2(fc) = tf2s(0,1,sqrt(2),1,2*PI*fc);          // 2nd-order
lowpass3(fc) = tf2s(0,1,1,1,w1) : tf1s(1,1,w1)...    // 3rd order
lowpass4(fc) = tf2s(0,1,a11s,1,w1) : tf2s( ...       // 4th order
```

Parametric equalizer section:

```
pareq(Q,F,G) = tf2(...); // (adapted from bandfilter.dsp)
```

`filter.lib`, **continued**

Interpolating delay lines generalizing `fdelay(n,d,x)`

Lagrange polynomial interpolation:

```
fdelay1(n,d,x);  // 1st-order Lagrange interpolation
fdelay2(n,d,x);  // 2nd-order
fdelay3(n,d,x);  // 3rd-order
fdelay4(n,d,x);  // 4th-order
```

Thiran allpass interpolation:

```
fdelay1a(n,d,x); // 1st-order allpass interpolation
fdelay2a(n,d,x); // 2nd-order
fdelay3a(n,d,x); // 3rd-order
fdelay4a(n,d,x); // 4th-order
```

effect.lib — Current Contents

Dynamic Level Lowpass Filter

```
levelfilter(level,freq); // from EKS
levelfilterN(N,freq,L) = seq(i,N,levelfilter((L/N),freq));
```

Overdrive

```
cubicnl(drive,offset);
```

Wah Pedals and VCFs

```
crybaby(wah);


moogvcf(Q,fr);
wah4(fr) = moogvcf(3.8,fr);
```

Audio Speaker Bandpass

```
speakerbp(f1,f2)    // "Loudspeaker Bandpass"
  = dcblockerat(f1) : dcblockerat(f1) : lowpass4(f2);
```

`osc.lib` — Current Contents

Misc.

```
impulse = 1-1';
sawtooth(freq);
```

Filter-Based oscillators `osc*(f)`

```
oscb(f) = impulse : tf2(1,0,0,a1,1) ... ; // ringing biquad

oscs(f); // ringing state-variable filter
         // = "magic circle", or "modified coupled form"
```

osc.lib — continued

Undamped Second-Order Normalized Ladder Resonator:

```
oscrs(f) =  impulse : nlf2(f,1) : _,!; // sine
oscrc(f) =  impulse : nlf2(f,1) : !,_; // cosine
oscrq(f) =  impulse : nlf2(f,1);       // quadrature
oscr = oscrs;                          // default = sine
```

Undamped Second-order Transformer-Normalized Waveguide Resonator:

```
oscwc(fr) = 1-1' : wgr(fr,1) : _,!; // cosine (1 mpy/sample)
oscws(fr) = 1-1' : wgr(fr,1) : !,_; // sine (2nd scaling mpy)
oscq(fr)  = 1-1' : wgr(fr,1);       // phase quadrature outs
oscw = oscwc;
```

Online Resources

- RealSimple module (including software download):
  `http://ccrma.stanford.edu/realsimple/faust_strings/`
- Online Book (physics-based synthesis of musical instruments):
  `http://ccrma.stanford.edu/~jos/pasp/`