Making Virtual Electric Guitars and Associated Effects Using Faust

Julius Smith

REALSIMPLE Project[*]

Center for Computer Research in Music and Acoustics (CCRMA)
Department of Music
Stanford University
Stanford, California 94305

Abstract

This series of laboratory exercises is concerned with building virtual stringed instruments and associated effects in the Faust programming language.

Contents

[*]Work supported in part by the Wallenberg Global Learning Network

1 Introduction

In this lab sequence, we will make "virtual stringed instruments," starting from a very simple case (the Karplus-Strong digitar algorithm) and working up to instruments of practical complexity. All of the examples can be classified as *digital waveguide* instruments, although the earliest algorithms were originally derived under different paradigms, as will be noted.

1.1 Prerequisites

We assume you have read the introductory tutorial entitled "Signal Processing in Faust and PD,"[1] or equivalent, and that you have installed the Planet CCRMA[2] distribution, or have separately installed `faust`, `pd`, and associated tools described in the tutorial. Familiarity with basic C++ programming in a Linux/GNU environment is assumed.

Finally, elementary signal processing proficiency on the level of [13] and [12] is assumed. Recommended background reading on digital-waveguide "theory of operation" is given by Chapters 1, 3, 4, and Appendix G of [14];[3] however, a detailed understanding of digital waveguide theory is not necessary for carrying out the laboratory exercises and/or building useful instruments.

1.2 Summary of the Labs

In the first lab exercise, we build a simple synth plugin based on the Karplus-Strong plucked-string "digitar" algorithm. This is followed by a number of extensions that add more features and improve sound quality and expressiveness. As the labs progress, a highly versatile "virtual electric guitar" is built, piece by piece.

Each lab assignment is preceded by a presentation of the basic theory of operation and a working Faust implementation for the virtual guitar component being considered. The exercises primarily consist of directed experiments using the provided Faust code, sometimes asking for simple modifications of the code.

1.3 Software Download

The software discussed in this module can be downloaded as a compressed tarball:

```
http://ccrma.stanford.edu/~jos/faust_strings/freeax.tar.gz
```

[1]http://ccrma.stanford.edu/realsimple/faust/
[2]`http://ccrma.stanford.edu/planetccrma/software/`
[3]`http://ccrma.stanford.edu/~jos/pasp/Digital_Waveguide_Theory.html`

2 Karplus-Strong Digitar Algorithm

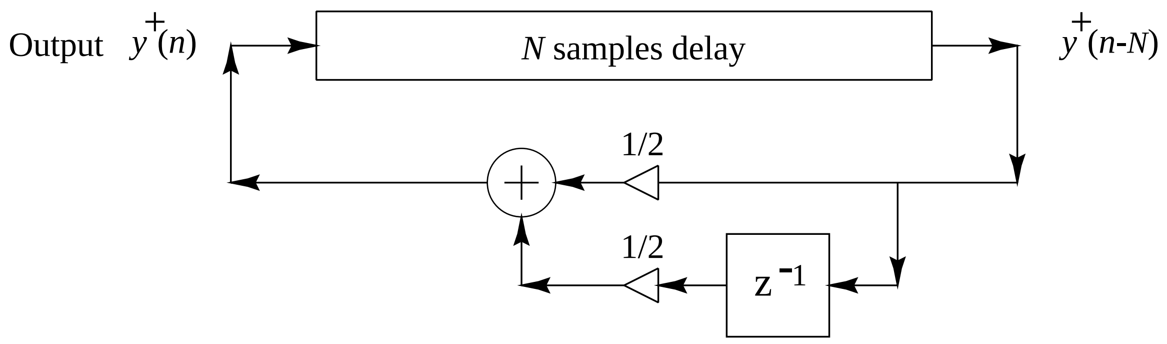

Figure 1: Signal flow graph of the Karplus Strong algorithm viewed as a digital filter excited from initial conditions: The delay line is initially filled with random numbers. The feedback filter $H(z) = [1 + z^{-1}]/2$ can be regarded as a *running two-point average*.

The KS digitar algorithm can be derived [14] as a simplified digital waveguide synthesis model for an "idealized" vibrating string (no stiffness, and very specific damping characteristics resulting in the two-point average). This physical interpretation is used to guide extensions to the basic algorithm.

The basic Karplus-Strong (KS) *digitar* algorithm [7][4] consists of a waveform memory that is read out and modified repeatedly each "period," where the modification is to replace each sample in the memory by the average of itself and the previous sample each time it is read. The algorithm is diagrammed as a digital filter in Fig. 1. There are other modes of operation of the KS algorithm described in [7], such as for percussive sounds and "bottle mode" that are not reviewed here. In other words, we consider only the *digitar* special case of the KS algorithm which simulates plucked string sounds. We begin our example series with the digitar because it is the simplest known string-synthesis algorithm that is both interesting to hear and derivable (in retrospect) from the physics of vibrating strings.

2.1 Exciting the Digitar String

A new "pluck" is obtained in the digitar algorithm by writing new random numbers into the waveform memory (the delay line in Fig. 1). The fundamental frequency F_0 is approximately given by the sampling rate f_s divided by the memory length N, or $F_0 \approx f_s/N$. This relation is not exact because the two-point average adds a half-sample *phase delay* [12].[5] A more accurate formula is therefore

$$F_0 = \frac{f_s}{N + \frac{1}{2}}. \tag{1}$$

[4]See also `http://en.wikipedia.org/wiki/Karplus-Strong_string_synthesis`

[5]`http://ccrma.stanford.edu/~jos/filters/Phase_Delay.html`

This formula can be used as exact for practical purposes, but it is not exact in theory due to the slight *decay* per period caused by the two-point average.[6]

Figure 2 lists a Faust program implementing the digitar algorithm (adapted from the programming example `karplus.dsp` distributed with Faust), and Fig. 3 shows the block diagram generated by `faust` for the `resonator` definition in Fig. 2 (using the `-svg` command-line option, as discussed in the Faust intro[7] [15]). If this Faust code is not self-explanatory, see [15] and/or [9].

```
import("music.lib"); // define noise, SR, delay

// MIDI-driven parameters:
freq = nentry("freq Hz", 440, 20, 20000, 1); // Hz
gain = nentry("gain", 1, 0, 10, 0.01);    // 0 to 1
gate = button("gate");                    // 0 or 1

// Excitation gate (convert gate to a one-period pulse):
diffgtz(x) = (x-x') > 0;
decay(n,x) = x - (x>0)/n;
release(n) = + ~ decay(n);
trigger(n) = diffgtz : release(n) : > (0.0);

// Resonator:
average(x) = (x+x')/2;
P = SR/freq;
resonator = (+ : delay(4096, P)) ~ (average);

process = noise : *(gain) : *(gate : trigger(P))
                : resonator;
```

Figure 2: Faust program `ks.dsp` specifying the Karplus-Strong (KS) digitar algorithm.

2.2 MIDI Control of the KS Digitar in PD

As discussed in the Faust intro [15], we can create an 8-voiced MIDI synthesizer for `pd` via the following shell commands on a Linux system with `faust` installed:

```
faust -xml -a puredata.cpp -o ks_pd.cpp ks.dsp
g++ -DPD -Wall -g -shared -Dmydsp=ks -o ks~.pd_linux ks_pd.cpp
faust2pd -s -n 8 ks.dsp.xml
```

The first line compiles `ks.dsp` to produce

- `ks_pd.cpp` — the `pd` external C++ source code, and
- `ks.dsp.xml` — an XML description of its user interface.

The second line compiles the C++ source, to produce

[6]Exact resonance tuning is found by determining the poles of the system and their angles in the z-plane [12]. More relevant perceptually are the frequencies of *local maxima* in the magnitude frequency response.

[7]http://ccrma.stanford.edu/realsimple/faust/

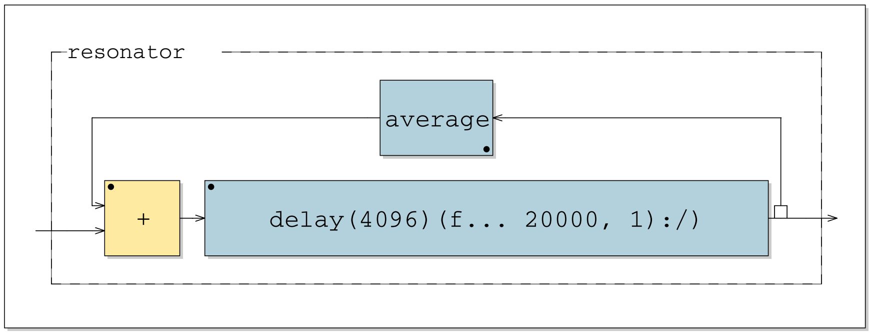

Figure 3: Faust-generated block diagram of the `resonator` definition in Fig. 2. For the digitar algorithm, a one-period burst of white noise may be injected as an input signal. For historical accuracy, the feedback should be eliminated during the excitation noise-burst (making the adder unnecessary), but this makes little difference in the sound under normal conditions.

- `ks~.pd_linux` — the loadable binary `pd` external module.

The last line generates

- `ks_pd.pd` — the `pd` plugin-wrapper with "graph-on-parent" controls.

As also discussed in the Faust intro, `faust2pd` comes with the `pd` abstraction `midi-in.pd` that maps MIDI key-number to Faust parameter `freq`, MIDI velocity to `gain`, and MIDI note-on/off to `gate`. Additional parameters are brought out as sliders et al. in the `pd` patch, but in this case, there are no additional parameters.

2.3 Karplus-Strong Laboratory Exercises

Use the above shell script to generate a `pd` synthesizer, and drive it either from the Virtual Keyboard or from an external MIDI keyboard, as described in the Faust intro. Answer the following questions:

1. *Exploring mistuning:* Since the delay line is not interpolated, the fundamental frequencies are quantized to frequencies given in Eq. (1) on page 4, where N is the (integer) delay-line length. This tuning error is very noticeable at low sampling rates (*e.g.*, 8 kHz or even 22 kHz). However, it is harder to hear at the standard 48 kHz sampling rate that is standard in AC-97 CODEC chips.

   (a) At what MIDI key-number do you start to hear the tuning error when the sampling rate is 44.1 kHz? (MIDI key-number 69 corresponds to A4 at 440 Hz, and key-number 60 is C4 ("middle C") at $440 \cdot 2^{60-69}/12 \approx 261.63$ Hz.) If possible, determine the first "out of tune" key-number when the sampling rate is 8 KHz.

   (b) Repeat the previous problem using `fdelay` in place of `delay`, and explain why your results make sense.

2. Measure the final signal value for three different notes after sound is no longer audible.

   (a) Explain why the signal usually does not decay to zero.

(b) What final values are possible?

(c) What property of the excitation noise burst can guarantee that the signal will decay to zero?

3 Extended Karplus-Strong Algorithm

The Extended Karplus-Strong (EKS) algorithm [4] extends the KS digitar in a number of ways that will be introduced one-by-one and then brought together in the complete program listing shown in Figures 9 and 10 starting on page 16. The EKS extensions were motivated by the demands of a musical composition[8] and the interpretation of the KS algorithm as a transfer-function model of a simplified physical string [11, pp. 158–198]. They illustrate how several small digital filters can achieve various desired musical effects. We will see that the EKS can be regarded as a *blend* of spectral and physical (transfer-function) modeling techniques.

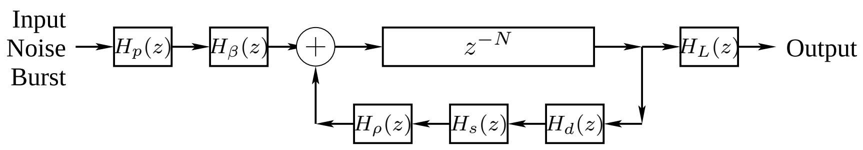

Figure 4: Extended Karplus-Strong (EKS) filters.

Figure 4 illustrates where the various filters may be located in the patch. The filters in series outside the feedback loop can of course be implemented in any order, and the filters within the feedback loop can be arbitrarily reordered. (The series order of linear, time-invariant filters may matter in fixed-point, but generally not in floating-point.)

3.1 Pick-Direction Lowpass Filter

The EKS pick-direction lowpass filter is simply a unity-dc-gain one-pole filter with a different coefficient for an "up-pick" than for a "down-pick". Thus, the filter transfer function is

$$H_p(z) \;=\; \frac{1-p}{1-p\,z^{-1}}$$

where p takes on two different (real) values (such as 0 and 0.9), depending on the picking direction. The idea is that real up-picks may be at different angles than down-picks, thus resulting in different plucking stiffness, among other possible effects.

In Faust, a unity-dc-gain one-pole filter can be defined by

```
pickdir =  *(1.0-(p)) : + ~ *(p);
```

where `p` is the pole location. This pole position can be modulated by a pick-direction "toggle" signal as follows:

```
p = 0.9 * checkbox("pick_direction"); // [0 or 0.9]
```

[8] "Silicon Valley Breakdown" by David A. Jaffe

3.2 Pick-Position Comb Filter

A natural model of string excitation is an input signal summed into a virtual physical location along the length of a digital waveguide string model, as described in [14]. This model can then be factored into a pick-position comb filter in series with a filtered delay loop, as used in the EKS [4, 11] and derived in [14].[9]

The EKS pick-position comb filter has the transfer function

$$H_\beta(z) \;=\; 1 - z^{-\lfloor \beta P \rfloor}$$

where P is the period (total loop delay) in samples, and $\beta \in (0,1)$ denotes normalized position along the string (0 being at the "bridge" and 1 being at the "nut"). The notation $\lfloor x \rfloor$ means the "greatest integer less than or equal to x," also called *truncation* to the next lower integer.[10] This transfer-function model of pick position is easily derived by simply factoring the transfer-function of the string from the picking point to any other point along the string, such as the bridge point [14, 11].

In Faust, a feedforward comb filter is readily implemented using the `delay` function defined in `music.lib`:

```
ppdel = beta*P; // pick position as fraction of period
pickpos = _ <: delay(Pmax,ppdel) : - ;
```

where `Pmax` is some power of 2 larger than `ppdel` (see the definition of `delay` in `music.lib` to understand why a power of 2). In Faust, we can bring out a "continuous" pick-position control spanning half the string as follows:

```
beta = hslider("pick_position", 0.13, 0, 0.5, 0.01); // 0-1/2
```

The block diagram generated by "`faust -svg --simple-names`" is shown in Fig. 5. Pick position accuracy is normally not critical, hence the 1% slider steps and lack of delay-line interpolation in the comb filter.

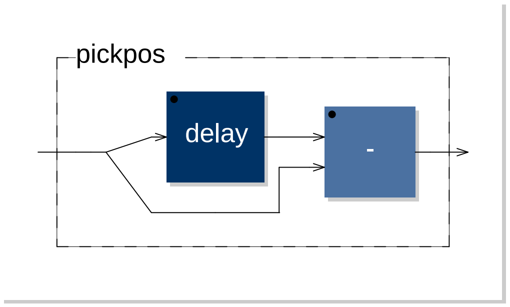

Figure 5: Faust-generated block diagram of the pick-position comb filter.

[9]http://ccrma.stanford.edu/ jos/pasp/Equivalent_Forms.html

[10]One may also use *rounding* to the nearest integer, which can be defined as $\lfloor x + 0.5 \rfloor$.

3.3 One-Zero String Damping Filter

The original EKS string-damping filter replaced the two-point average of the KS digitar algorithm by a *weighted* two-point average

$$H_d(z) = (1 - S) + Sz^{-1} \tag{2}$$

where $S \in [0, 1]$ is called the "stretching factor," and it adjusts the relative decay-rate for high versus low frequencies in the string. This filter goes in the string feedback loop, as shown in Fig. 4 above. At $S = 0$ or 1, the decay time is "stretched infinitely" (no decay), while fastest decay is obtained when $S = 1/2$, where it reduces to the KS digitar damping filter. The decay-time is always infinity for dc, and higher frequencies decay faster than lower frequencies when $S \in (0, 1)$.

To control the overall decay rate, another (frequency-independent) gain multiplier $\rho \in (0, 1)$ was introduced to give the loop filter

$$H_d(z) = \rho[(1 - S) + Sz^{-1}].$$

Since this filter is applied once per period P at the fundamental frequency, an attenuation by the factor $|H_d(e^{j2\pi/P})| \approx \rho$ occurs once each P samples. Setting ρ to achieve a decay of -60 dB in t_{60} seconds is obtained by solving

$$\rho^{\frac{t_{60}}{PT}} = 0.001 \;\Rightarrow\; \boxed{\rho = (0.001)^{PT/t_{60}}.}$$

In Faust, we can calculate ρ from the desired decay-time in seconds as follows:

```
t60 = hslider("decaytime_T60", 4, 0, 10, 0.01);  // seconds
rho = pow(0.001,1.0/(freq*t60));
```

where `freq` is the fundamental frequency (computed from the MIDI key number in the example of Fig. 9 on page 16).

The following Faust code implements the original EKS damping filter in terms of a "brightness" parameter B between 0 and 1:

```
B = hslider("brightness", 0.5, 0, 1, 0.01); // 0-1

b1 = 0.5*B; b0 = 1.0-b1; // S and 1-S
dampingfilter1(x) = rho * (b0 * x  +  b1 * x');
```

Relating to Eq. (2), we have the relation $S = B/2$.

3.4 Two-Zero String Damping Filter

A disadvantage of the decay-stretching parameter is that it affects tuning, except when $S = 0$. This can be alleviated by going to a second-order, symmetric, linear-phase FIR filter having a transfer function of the form [19]

$$H_d(z) = g_1 + g_0 z^{-1} + g_1 z^{-2} = z^{-1}\left[g_0 + g_1(z + z^{-1})\right].$$

Due to the symmetry of the impulse response $h_d = [g_1, g_0, g_1, 0, 0, \ldots]$ about time $n = 1$, only two multiplies and two additions are needed per sample. The previous one-zero loop-filter required one multiply and two additions per sample. Since the delay is equal to one sample at all frequencies (in the needed coefficient range), we obtain tuning invariance for the price of one additional multiply per sample. We also obtain a bit more lowpass filtering. Listening to both cases, one might agree that the one-zero loop filter has a "lighter, sweeter" tone than the two-zero case. In general, the tone is quite sensitive to the details of all filtering in the feedback path of Fig. 4.

See [14][11] for a derivation of the FIR filter coefficients `h0,h1` as a function of brightness `B`. A Faust implementation may then be written as follows:

```
t60 = hslider("decaytime_T60", 4, 0, 10, 0.01);  // sec
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;
dampingfilter2(x) = rho * (h0 * x' + h1*(x+x''));
```

3.4.1 Second-Order FIR Damping Filter Exercises

1. With brightness set to $B = 0$ and t_{60} set to 1 second, measure and report the decay time for all notes 'A' over the range of the piano keyboard. (A-440 is a sixth above middle-C.) Explain any systematic deviations measured. [The `faust2octave` utility can be useful for this exercise.]

2. Repeat the previous problem with $B = 1$.

3.5 Dynamic Level Lowpass Filter

In real strings, the spectral centroid typically rises as plucking/striking becomes more energetic. The EKS dynamic-level lowpass filter (diagrammed at the far right in Fig. 4) qualitatively models this phenomenon:[12]

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

This is another unity-dc-gain one-pole lowpass, with a pole at $z = R_L \in [0, 1)$ set such that the gain is the same for all fundamental frequencies [4]. Here we will derive simplified design formulas.

Assume that the ideal *continuous-time* filter has the transfer function

$$H_{L,\omega_1}(s) = \frac{\omega_1}{s+\omega_1} \tag{3}$$

where $\omega_1 = 2\pi f_1$ denotes the fundamental frequency in radians per second. This lowpass filter has unity dc gain, -3 dB gain at $s = j\omega_1$, and rolls off -6 dB/octave for $\omega \gg \omega_1$.[13] It also happens to be the 1st-order Butterworth lowpass with cut-off frequency set to ω_1 rad/sec. To achieve the

[11]http://ccrma.stanford.edu/ jos/pasp/Length_FIR_Loop_Filter.html

[12]A "spectral modeling filter" of this nature is only needed for spectrally monotonous excitations such as the KS digitar noise burst. A proper physical string-excitation model should have this behavior built in.

[13]The -3dB-gain frequency ω_1 is often called the *break frequency* in the context of classical control design. This is because, as frequency ω increases from 0, the pole at $s = -\omega_1$ has little effect on the frequency response until $\omega \approx \omega_1$, where the pole "breaks," resulting in a -6dB/octave roll-off in the amplitude response for higher frequencies ($\omega \gg \omega_1$).

dynamic level effect, the output of this filter is linearly panned with its input. If $x(n)$ denotes the lowpass input signal and $y(n)$ its output, then the formula is

$$L \cdot L_0(L) \cdot x(n) + (1 - L) \cdot y(n)$$

where the level variable $L \in [0, 1]$ may be set to achieve a desired dynamic level at the Nyquist limit, while L_0 controls the (lesser) attenuation at low frequencies as a function of level L (*e.g.*, $L_0(L) = L^{1/3}$). At maximum level $L = 1$, the lowpass filter is bypassed. Figure 7 on page 12 shows a family of filter responses at four different dynamic levels and six different fundamental frequencies.

An example GUI specification for the L calculation in Faust is as follows:

```
L  = hslider("dynamic_level", -10, -60, 0, 1) : db2linear;
```

where `db2linear(x)` is defined in `music.lib` as `pow(10, x/20.0)`.

In [14],[14] the *impulse-invariant* and *bilinear transform* methods are compared for digitizing the dynamic-level analog filter Eq. (4), and the bilinear transform method was deemed preferable because it gives more attenuation of high frequencies, which helps to reduce aliasing due to later nonlinear processing. A detailed derivation can be found there. The final digital filter so designed has the transfer function

$$H_{L,\omega_1}(z) = \frac{\tilde{\omega_1}}{1 + \tilde{\omega_1}} \frac{1 + z^{-1}}{1 - \left(\frac{1-\tilde{\omega_1}}{1+\tilde{\omega_1}}\right) z^{-1}} \tag{4}$$

with $\tilde{\omega_1} \overset{\Delta}{=} \omega_1 T/2$.

Figure 6 shows a family of magnitude responses for $H_{0,\omega_1}(e^{j\omega T})$ for 6 different fundamental frequencies ω_1.

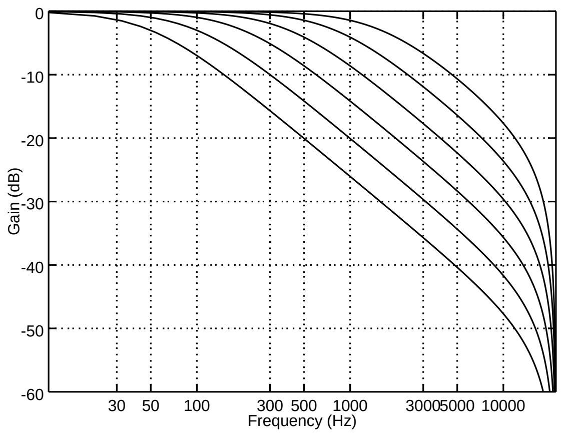

Figure 6: Dynamic level lowpass filter designed by the bilinear-transform method with $L = 0$. The filter amplitude response is plotted for 6 values of break frequency (50, 100, 200, 400, 800, and 1600 Hz). The sampling rate is $f_s = 44100$ Hz.

[14]`http://ccrma.stanford.edu/~jos/pasp/Making_Virtual_Electric_Guitars.html`

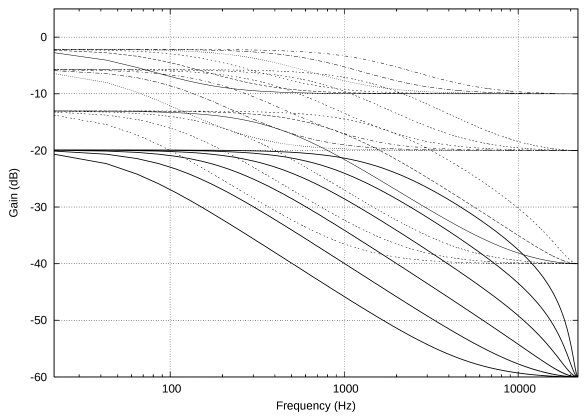

Figure 7: Dynamic level lowpass filter responses as in Fig. 6, but with $L = 0.001$, 0.01, 0.1, and 0.32, corresponding to desired Nyquist-limit levels of -60, -40, -20, and -10 dB, respectively. The dc level is defined to be one-third the Nyquist-limit level.

Faust Implementation (`effect.lib`)

```
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));
```

3.5.1 Dynamic Level Filter Laboratory Exercise

1. Implement the above dynamic level filter in a Faust EKS synth plugin, and find a good mapping for MIDI velocity to dynamic level L. Try to use a simple power law of the form (in Faust)

   ```
   L = scale * log(gain) + offset;
   ```

   where `gain` is between 0 and 1. Report your values for `scale` and `offset`.

2. Record a sequence of notes on a real guitar with progressively increasing amplitude. Plot the spectral centroid as a function of time for this recording.

3.6 Tuning the EKS String

At low sampling rates and/or high fundamental frequencies, the string simulation can sound "out of tune" because the main delay-line length N is an integer, which means that the fundamental frequency F_0 is quantized to values of the form $F_0 = f_s/(N + N_f)$ where f_s is the sampling rate and N_f is the delay (in samples) of any filters in the feedback loop. For example, in Fig. 4 on page 7, N_f equals the combined delay of filters $H_d(z)$, $H_s(z)$, and $H_\eta(z)$. In Eq. (1) on page 4, we had the digitar tuning formula $F_0 = f_s/(N+1/2)$ because $N_f = 1/2$ is the phase delay of the two-point average $y(n) = [x(n) + x(n-1)]/2$ used in the KS digitar algorithm.

In this section, we look at designing a *tuning filter* $H_\eta(z)$ so as to fine-tune the fundamental frequency as desired (even at low sampling rates). Keep in mind, however, that such a filter is not needed when the sampling rate is sufficiently high compared with the desired fundamental frequency.

For simplicity, here we will use the two-zero damping filter described in §3.4, so that its phase delay is always one sample. The tuning formula becomes

$$F_0 = \frac{f_s}{N + 1 + \Delta_\eta(\omega)}, \text{ or } \boxed{\Delta_\eta = \frac{f_s}{F_0} - N - 1,} \tag{5}$$

where

$$\Delta_\eta(\omega) \overset{\Delta}{=} -\frac{\angle H_\eta(e^{j\omega T})}{\omega T}$$

denotes the phase delay of the tuning filter H_η in samples.

3.6.1 Tuning by Linear Interpolation

An overview of linear interpolation, among others, is given in [14].[15] The transfer function of first-order linear-interpolation can be written as

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

where $\eta \in [0, 1]$ denotes the desired delay in samples at low frequencies compared with the sampling rate.

Faust Implementation.

Faust includes a function `fdelay(n,d,x)` defined in `music.lib` which provides fractional (non-integer) delay by means of linear interpolation:

```
fdelay(n,d,x) = delay(n,int(d),x)*(1 - frac(d))
              + delay(n,int(d)+1,x)*frac(d);
```

Note that it also specifies a second delay line. However, a second delay-line is not implemented in the generated C++ code because Faust has an optimization rule that consolidates all delay-lines having the same input signal to one shared delay line.

A single-delay-line version can be defined as follows:

```
linterp(d,x) = (1-d) * x + d * x';
fdelay1(n,d,x) = delay(n,int(d),x) : linterp(frac(d));
```

[15]http://ccrma.stanford.edu/ jos/pasp/Delay_Line_Interpolation.html.html

Note that these definitions are *not equivalent.* While they are equivalent when the delay `d` is fixed, they diverge when `int(d)` changes from one sample to the next. In the dual-delay-line specification, the desired result is obtained, while in the single-delay-line case, `x'` becomes a one-sample delay of the *old* delay-line output instead of the *new* delay-line output. This inconsistency can potentially cause audible clicks when the tuning is rapidly varied.

A proper implementation of the single-delay-line case involves resetting the linear-interpolator state variable when the delay-length changes. Conceptually, the linear interpolator should be implemented as two delay-line taps with gains $(1-\eta)$ and η. This structure is produced by the Faust optimization rules from the dual-delay-line specification as above.

Linear delay-line interpolation works well in a digital waveguide string model as long as the modeled string is sufficiently *damped.* Specifically, the string damping must be sufficient to mask the changing roll-off in the amplitude response of the linear interpolator. In the case of very light damping (such as when simulating steel strings at normal audio sampling rates), certain notes (such as B-flat at a sample rate of 44.1 kHz) will jump out as "buzzy" when they correspond to a nearly integer delay-line length (fundamental frequencies close to the sampling rate divided by an integer). This artifact diminishes with oversampling factor, of course.

3.6.2 Tuning by Lagrange Interpolation

Researchers at the Helsinki University of Technology have historically used Lagrange interpolation for digital-waveguide fine-tuning [5, 18, 6]. This has the advantage of being robust under rapidly time-varying conditions, as opposed to allpass interpolation which can exhibit artifacts when the delay changes too rapidly. However, Lagrange interpolation, like all FIR filters except pure delays, has the disadvantage of introducing some gain error in the string feedback loop, unlike allpass interpolation.

As discussed further in [14],[16] the closed-form expression for Lagrange-interpolation coefficients is

$$h_\Delta(n) = \prod_{\substack{k=0\\k\neq n}}^{N} \frac{\Delta - k}{n-k}, \quad n = 0,1,2,\ldots,N$$

where N is the length of the interpolator and Δ is the desired delay, centered about $(N-1)/2$. Note that $N=1$ corresponds to linear interpolation. For audio delay-line interpolation, the fourth-order case $N=4$ is often sufficient (and sometimes overkill).

For the same reasons discussed in §3.6.1, fourth-order Lagrange interpolation of a delay line is best viewed as a five-tap *interpolating read* of the delay line at the desired fractional delay. In normal object-oriented languages, an interpolating read is naturally implemented internal to the interpolating delay-line object. In Faust, the same effect is obtained by specifying *five* delay lines, as shown in Fig. 8. While it appears that five delay lines are needed in the Faust implementation, only one is actually used in the generated C++ code, thanks to compiler optimizations. Faust implementations of Lagrange interpolation of orders 1 through 4 may be found in the file `filter.lib` distributed with Faust (starting with version 0.9.9.3).

[16]http://ccrma.stanford.edu/ jos/pasp/Lagrange_Interpolation.html.html

```
fdelay4(n,d,x) = delay(n,id,x) * fdm1*fdm2*fdm3*fdm4/24
               + delay(n,id+1,x) * (0-fd*fdm2*fdm3*fdm4)/6
               + delay(n,id+2,x) * fd*fdm1*fdm3*fdm4/4
               + delay(n,id+3,x) * (0-fd*fdm1*fdm2*fdm4)/6
               + delay(n,id+4,x) * fd*fdm1*fdm2*fdm3/24
with {
  id = int(d-1);
  fd = 1+frac(d);
  fdm1 = fd-1;
  fdm2 = fd-2;
  fdm3 = fd-3;
  fdm4 = fd-4;
};
```

Figure 8: Faust implementation of fourth-order Lagrange interpolation.

3.7 EKS Tuning and Decay Exercises

Use the shell script in §3.9 below to generate a `pd` synthesizer based on the EKS, and drive it either from the Virtual Keyboard or from an external MIDI keyboard, as described in the Faust intro.

1. Checking Tuning
   (a) Using `delay` (no interpolation) for the main delay line, determine if you can hear the tuning error when the sampling rate is 44.1 kHz, and if so, report the lowest MIDI key-number and other settings used.
   (b) Repeat using `fdelay` (linear interpolation).
   (c) Repeat using `fdelay4` (4th-order Lagrange interpolation).

2. Uniformity of Decay
   (a) Using `fdelay` for tuning, set the brightness set to 1 and the decaytime to maximum. Determine the set of MIDI key numbers, if any, at which there is noticeably reduced decay. These are notes that sound markedly brighter than their neighbors and tend to "jump out" when playing a chromatic scale.
   (b) Repeat with brightness set to 0.
   (c) Repeat with brightness set to 1 and using `fdelay4` in place of `fdelay`.

3. Measure and report the decay time (either by ear, or by measuring t_{60} on a dB-magnitude waveform display) at notes C1, C2, C3, and C4 when the decay-time parameter is set to its minimum and maximum values. How good is t_{60} is a definition of decay time?

3.8 EKS Faust Listing

Figures 9 and 10 give a Faust implementation of the Extended-Karplus-Strong algorithm.

```
declare name "EKS Electric Guitar Synth";
declare author "Julius Smith";
declare version "1.0";
declare license "STK-4.3";
declare copyright "Julius Smith";
declare reference "http://ccrma.stanford.edu/~jos/pasp/vegf.html";
// -> Virtual\_Electric\_Guitars\_Faust.html";

import("music.lib");    // Define SR, delay
import("filter.lib");   // smooth, ffcombfilter,fdelay4
import("effect.lib");   // stereopanner

//==================== GUI SPECIFICATION ================

// standard MIDI voice parameters:
// NOTE: The labels MUST be "freq", "gain", and "gate" for faust2pd
freq = nentry("freq", 440, 20, 7040, 1);  // Hz
gain = nentry("gain", 1, 0, 10, 0.01);    // 0 to 1
gate = button("gate");                    // 0 or 1

// Additional parameters (MIDI "controllers"):

// Pick angle in [0,0.9]:
pickangle = 0.9 * hslider("pick_angle",0,0,0.9,0.1);

// Normalized pick-position in [0,0.5]:
beta = hslider("pick_position [midi: ctrl 0x81]", 0.13, 0.02, 0.5, 0.01);
       // MIDI Control 0x81 often "highpass filter frequency"

// String decay time in seconds:
t60 = hslider("decaytime_T60", 4, 0, 10, 0.01);  // -60db decay time (sec)

// Normalized brightness in [0,1]:
B = hslider("brightness [midi:ctrl 0x74]", 0.5, 0, 1, 0.01);// 0-1
    // MIDI Controller 0x74 is often "brightness"
    // (or VCF lowpass cutoff freq)

// Dynamic level specified as dB level desired at Nyquist limit:
L = hslider("dynamic_level", -10, -60, 0, 1) : db2linear;
// Note: A lively clavier is obtained by tying L to gain (MIDI velocity).

// Spatial "width" (not in original EKS, but only costs "one tap"):
W = hslider("center-panned spatial width", 0.5, 0, 1, 0.01);
A = hslider("pan angle", 0.5, 0, 1, 0.01);

// Append Part 2 here
```

Figure 9: Part 1 of Faust program `eks.dsp` specifying an implementation of the Extended Karplus Strong (EKS) algorithm.

```
//=================== SIGNAL PROCESSING ================

//----------------------- noiseburst -------------------------
// White noise burst (adapted from Faust's karplus.dsp example)
// Requires music.lib (for noise)
noiseburst(gate,P) = noise : *(gate : trigger(P))
with {
  diffgtz(x) = (x-x') > 0;
  decay(n,x) = x - (x>0)/n;
  release(n) = + ~ decay(n);
  trigger(n) = diffgtz : release(n) : > (0.0);
};

P = SR/freq; // fundamental period in samples
Pmax = 4096; // maximum P (for delay-line allocation)

ppdel = beta*P; // pick position delay
pickposfilter = ffcombfilter(Pmax,ppdel,-1); // defined in filter.lib

excitation = noiseburst(gate,P) : *(gain); // defined in signal.lib

rho = pow(0.001,1.0/(freq*t60)); // multiplies loop-gain

// Original EKS damping filter:
b1 = 0.5*B; b0 = 1.0-b1; // S and 1-S
dampingfilter1(x) = rho * ((b0 * x) + (b1 * x'));

// Linear phase FIR3 damping filter:
h0 = (1.0 + B)/2; h1 = (1.0 - B)/4;
dampingfilter2(x) = rho * (h0 * x' + h1*(x+x''));

loopfilter = dampingfilter2; // or dampingfilter1

filtered_excitation = excitation : smooth(pickangle)
    : pickposfilter : levelfilter(L,freq); // see filter.lib

stringloop = (+ : fdelay4(Pmax, P-2)) ~ (loopfilter);
//Adequate when when brightness or dynamic level are sufficiently low:
//stringloop = (+ : fdelay1(Pmax, P-2)) ~ (loopfilter);

// Second output decorrelated somewhat for spatial diversity over imaging:
widthdelay = delay(Pmax,W*P/2);

// Assumes an optionally spatialized mono signal, centrally panned:
stereopanner(A) = _,_ : *(1.0-A), *(A);

process = filtered_excitation : stringloop
          <: _,_ : widthdelay : stereopanner(A);
```

Figure 10: Part 2 of Faust program `eks.dsp` specifying an implementation of the Extended Karplus Strong (EKS) algorithm.

3.9 MIDI Control of an EKS Patch in PD

As detailed in the Faust intro[17] [15], we create an 8-voiced MIDI synthesizer for `pd` as follows:

```
faust -xml -a puredata.cpp -o eks_pd.cpp eks.dsp
g++ -DPD -Wall -g -shared -Dmydsp=eks -o eks~.pd_linux eks_pd.cpp
faust2pd -s eks.dsp.xml
```

The last line produces `eks_pd.pd` which can be loaded into `pd`. In addition to the basic three MIDI-control parameters (`gate`, `freq`, and `gain`), we have the following additional parameters that are brought out as sliders in `pd`:

- Brightness
- Decay time (t_{60})
- Dynamic level
- Pick position
- Pick angle (or direction)

Finally, there are some simply computed parameters associated purely with panning the output sound into the stereo field:

- Panning angle
- Spatial width

See Fig. 9 and Fig. 10 for details.

3.10 Generality of the EKS Algorithm

As discussed in [14],[18] the white-noise excitation used in the KS digitar and EKS algorithms can be interpreted physically as a *random initial displacement and velocity* for each string element (spatial sample), and the resulting string vibration decays exponentially, corresponding to a linear string model. This type of model works well for *plucked and struck strings*, particularly a variety of lively *clavier*-type instruments. However, to model a real acoustic stringed instrument, such as a classical guitar or piano, we need to model the *body resonator* (guitar body, soundboard, etc.).

Note that solid-body *electric guitars*, such as the Les Paul or Stratocaster, do not need a resonator model, because their string vibrations are measured directly by magnetic pickups, and body resonances (which are minimized in solid-body design and construction) have only a small effect on the pickup signal. Therefore, all we really need for these instruments, beyond our string model, is guitar *effects*, such as distortion, and a good *amp model*, because electric guitar players classically achieve their ultimate sound in conjunction with the nonlinear operation of a good guitar amp (preferably a *tube* amp, as of this writing).

[17]http://ccrma.stanford.edu/realsimple/faust/
[18]http://ccrma.stanford.edu/ jos/pasp/Karplus_Strong_Algorithm.html

4 Distortion and Amplifier Feedback

This section discusses adding *distortion* and *amplifier feedback* to the basic EKS algorithm, as introduced in [17]. The resulting synthesis model is capable of convincing synthesis of "howling" overdriven electric guitars.[19]

4.1 Cubic Nonlinear Distortion

To minimize aliasing, it is helpful to use nonlinearities that are approximated by polynomials of low order. An often-used *cubic nonlinearity* is given by [17]

$$f(x) = \begin{cases} -\frac{2}{3}, & x \le -1 \\ x - \frac{x^3}{3}, & -1 < x < 1 \\ \frac{2}{3}, & x \ge 1. \end{cases} \tag{6}$$

and diagrammed in Fig. 11.[20] An input gain may be used to set the desired degree of distortion. Analysis of spectral characteristics and associated aliasing due to nonlinearities appears in [14].[21] As discussed there, a non-saturating cubic nonlinearity does not alias at all when the input signal is oversampled by 2 or more and the nonlinearity is followed by a half-band lowpass filter, which eliminates aliasing since it is confined to the upper half-spectrum between $\pi/2$ and π rad/sample. High quality commercial guitar distortion simulators are said to use oversampling factors of 4 to 8.

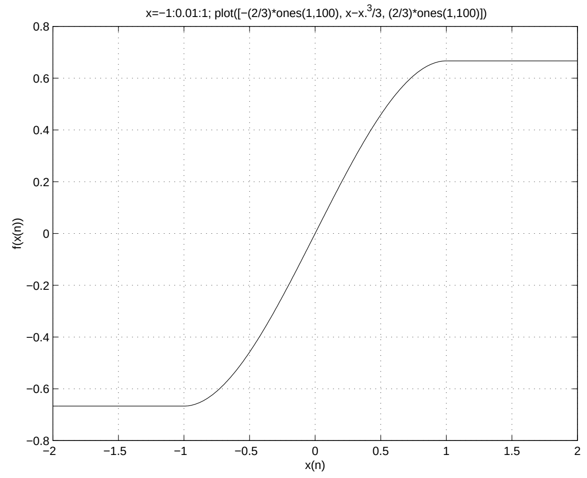

Figure 11: Soft-clipper defined by Eq. (6).

The cubic nonlinearity, being an *odd* function, produces only odd harmonics. To break the odd symmetry and bring in some even harmonics, a simple input offset can be used [10]. It was found empirically that a dc blocker [12][22] was needed to keep the signal properly centered in the output

[19]`http://ccrma.stanford.edu/~jos/pasp/Sound_Examples.html`

[20]The `faust2pd` distribution includes a "Fuzz effect," based on taking an absolute value, in the file `karplusplus.dsp`.

[21]http://ccrma.stanford.edu/~jos/pasp/Spectrum_Memoryless_Nonlinearities.html

[22]http://ccrma.stanford.edu/~jos/filters/DC_Blocker.html

dynamic range. Since amplifier loudspeakers have a +12 dB/octave low-frequency response, at least two dc blockers are appropriate anyway.

While the cubic nonlinearity is the odd nonlinearity with the least aliasing (thereby minimizing oversampling and guard-filter requirements), it is sometimes criticized as overly *weak* as a nonlinearity, unless driven into the hard-clipping range where it is no longer bandlimited to three times the input signal bandwidth.

Faust Implementation

In Faust, we can describe the cubic nonlinearity as follows (contained in `effect.lib` distributed with Faust):

```
//-------------------- cubicnl(drive,offset) ----------------------
// Cubic nonlinearity distortion
// USAGE: cubicnl(drive,offset), where
//   drive  = distortion amount, between 0 and 1
//   offset = constant added before nonlinearity to give even harmonics
// Reference:
// http://ccrma.stanford.edu/~jos/pasp/Nonlinear_Distortion.html#18254
//
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;
};
```

A simple test program is as follows:

```
 // tcubicnl.dsp

 import("effect.lib");

 // GUI Controls:
 O  = hslider("even_harmonics",0,0,0.5,0.01);
 D  = hslider("distortion [midi: ctrl 0x70]",0.1,0.01,1,0.01);
 g   = hslider("level [midi: ctrl 0x7]",0.1,0,1,0.01);
 process = ramp(0.01) : cubicnl
 with {
   integrator = + ~ _ ;
   ramp(slope) = slope : integrator - 2.0;
 };
distortion = cubicnl(O,D); // effect.lib

process = ramp(0.01) : -(1.5) : distortion;
```

To plot the output signal, say, in a shell, for example,

```
faust2octave tcubicnl.dsp
```

4.2 Nonlinearity Exercises

1. Excite the cubic nonlinearity with a sinusoid of amplitude 0.5 and frequency $f = 0.4 f_s$. Find the amplitude of all spectral components, including the fundamental frequency (normalized to 0 dB), and any aliased components.

2. Repeat the previous problem with a doubled sampling rate.

4.3 Amplifier Feedback

In [17], amplifier feedback to the strings was simulated as follows: The sum of all vibrating strings was passed through the cubic nonlinearity, multiplied by a feedback gain, delayed, and summed into the strings. This is easily implemented in Faust. (See `freeax.dsp` distributed with this module.)

5 Coupled Strings in Faust

A diagram for two coupled strings is given in [14].[23] A Faust template for this block diagram is displayed in Fig. 12 and specified as follows:

```
import("music.lib");
g1 = 0.999; g2 = 0.995; P = 200; Pmax = 256; detune = 1;
d1 = fdelay(Pmax,P-2);
d2 = delay(Pmax,P*(1.0 - 0.01*detune)-2);
bridgefilter = + : *(-0.5);
g = g1;
stringloop = ( _,+ : ((d2 <: _,_),(d1 <: _,_))
    : (_, (bridgefilter <: _,_) ,_)
    : +,+) ~ (*(g2),*(g1)) ;

process = stringloop;
```

where `P` is the fundamental period, in samples, and g denotes the round-trip filtering on the string during one period. (Placeholder values are given in the Faust listing so it will compile and generate Fig. 12.)

Note that the excitation only enters one of the string loops in Fig. 12. This corresponds, for example, to plucking the string in the horizontal plane, say (the `d1` loop), with the vertical plane (`d2` loop) vibrating "sympathetically". More generally, the two loops may be excited by varying amounts of the excitation signal, corresponding to a physically inexact excitation plane.

As discussed in [14],[24] the `bridgefilter` $H_b(z)$ is of the form

$$H_b(z) \triangleq \frac{2}{2 + R_b(s)/R}$$

[23]`http://ccrma.stanford.edu/\char'~jos/pasp/Two_Ideal_Strings_Coupled.html`

[24]http://ccrma.stanford.edu/ jos/pasp/Two_Ideal_Strings_Coupled.html

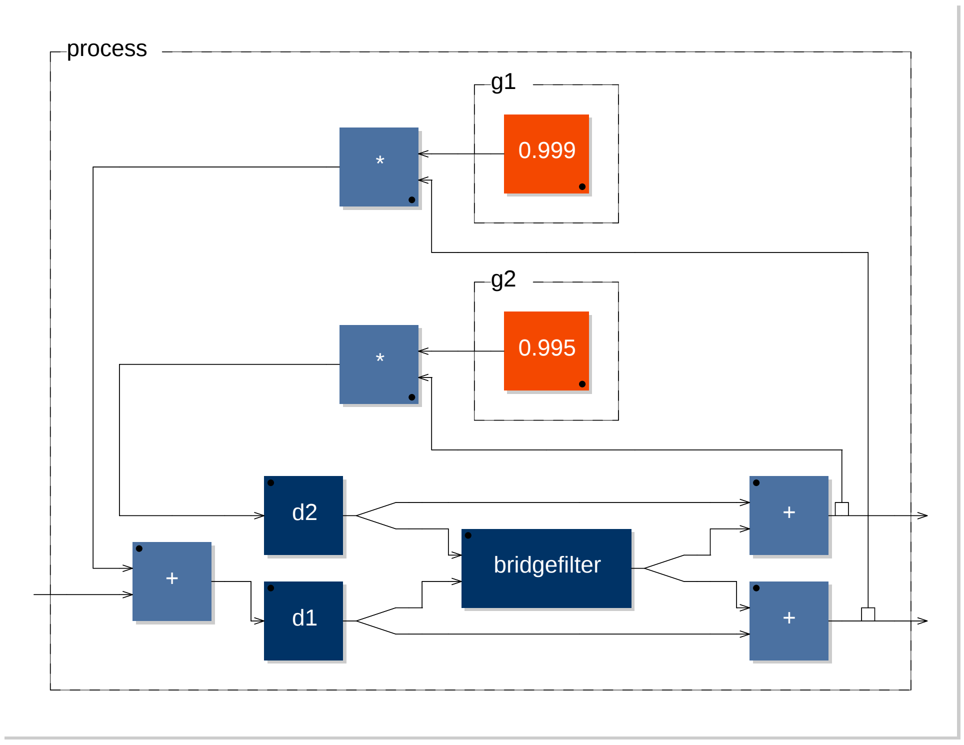

Figure 12: Two strings coupled by a general bridge impedance.

where R is the (real, positive) wave impedance of the string, and $R_b(s)$ denotes the bridge driving-point impedance (a positive-real function of the Laplace variable s). The special case indicated in the Faust listing above, $H_b(z) = 0.5$, corresponds to $R_b = 2R$, which is similar to the following simplified diagram (shown in Fig. 13) when `g1` = `g2` = `g`:

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

This simplified coupling algorithm runs about twice as fast as the full algorithm (based on Faust benchmarks using the `bench.cpp` architecture file).

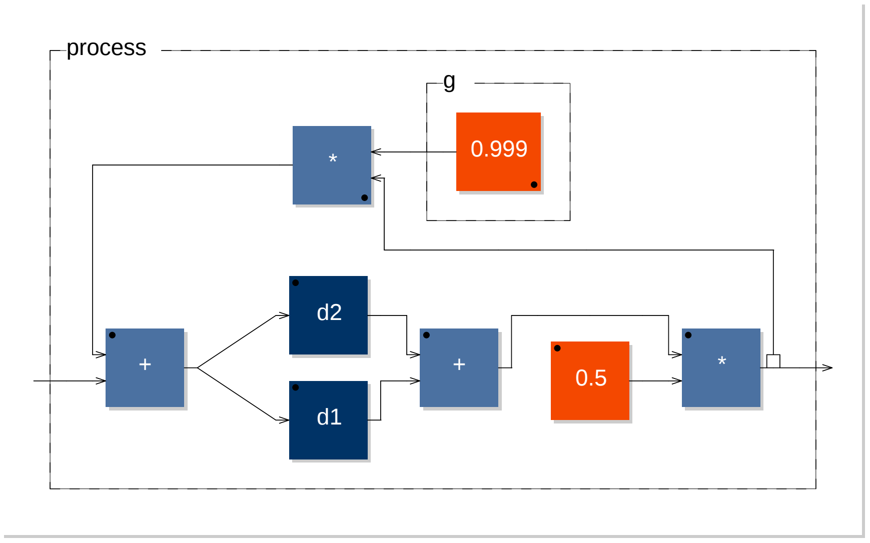

Figure 13: Two strings coupled at a real bridge impedance equal to twice the string impedance, *i.e.*, $R_b = 2R$.

6 Adding a Wah Pedal

A *wah pedal* (or *wah-wah*, or *CryBaby* pedal) operates by sweeping a single resonance through the spectrum. The resonance is conventionally second-order.

6.1 Digitizing the CryBaby

Figures 14, 15, and 16 (solid lines) show the amplitude responses of the author's CryBaby wah pedal measured (as described in §6.1.2 below) at three representative pedal settings (rocked fully backward, middle, and forward). Our goal is to "digitize" the CryBaby by devising a second-order sweeping resonator that audibly matches these three when the "wah" variable is 0, 1/2, and 1, respectively.

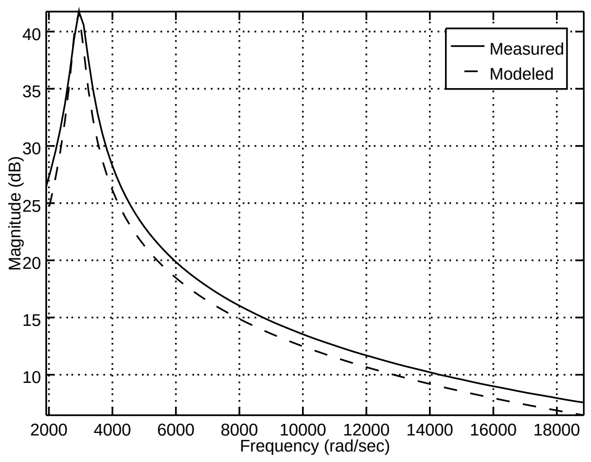

Figure 14: Measured (solid line) and modeled (dashed line) amplitude responses of the CryBaby pedal rocked back full.

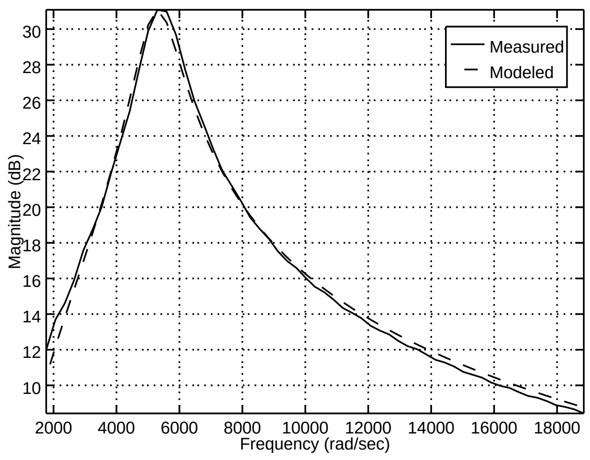

Figure 15: Measured (solid line) and modeled (dashed line) amplitude responses of the CryBaby pedal set to the middle of its excursion.

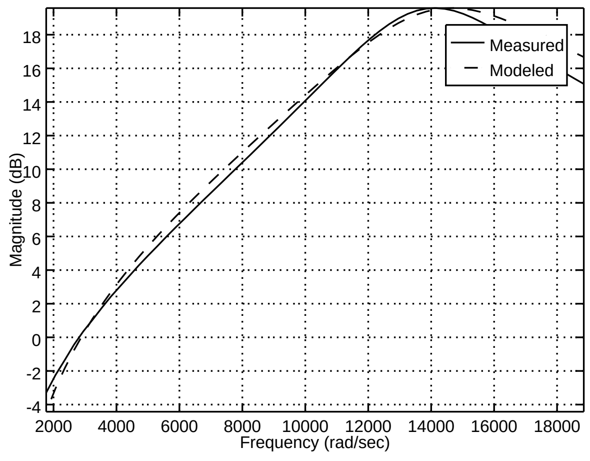

Figure 16: Measured (solid line) and modeled (dashed line) amplitude responses of the CryBaby pedal rocked fully forward.

6.1.1 Choice of Wah Filter Structure

A classic second-order resonator with *separate controls* for resonance frequency and resonance Q (quality factor) is the *state variable filter* [16, 2, 3]. However, the measurements described below reveal that resonance-frequency, Q, and gain all vary significantly with pedal angle. For that reason, and because our Faust implementation uses floating point (thus eliminating the need to consider special filter structures for improved fixed-point behavior), we choose the simple biquad section [12][25] to implement the wah resonator.

In Faust, the function `TF2(b0,b1,b2,a1,a2)` (defined in `music.lib`) implements a biquad filter section:

```
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;
};
```

It remains to express the five biquad coefficients as a function of a single `wah` variable. This will be done by fitting a biquad to three measured frequency responses and coming up with an interpolation formula for the varying coefficients.

6.1.2 Measuring the CryBaby Frequency Response

Measuring the frequency response of a wah pedal is relatively easy because it is a single-input, single-output, analog audio filter, with quarter-inch input/output jacks. A CryBaby pedal[26] was hooked up to an input and output of a Gina3G audio interface connected to a Linux PC (Red Hat Fedora 7 distribution) with Planet CCRMA installed. The response measurements shown in Figures 14 through 16 were carried out in `pd` and Octave[27] using software from the RealSimple Transfer Function Measurement Toolbox (RTFMT) [1].[28] The Octave command-line for generating the test input data (a sine sweep whose frequency increases exponentially with time) was as follows:

```
generate_sinesweeps(40,10000,48000,2);
```

This specifies a sine sweep from 40 Hz to 10 kHz lasting 2 seconds, with the sampling rate set to 48 kHz. Next, the shell command-line

```
pd sinesweeps.pd
```

opens the `pd` patch shown in Fig. 17. This `pd` patch (also distributed with the RTFMT) plays the sinesweep and records the response when the button labeled "Record Response To The Sine Sweeps" is clicked. The captured sweep-response is displayed so that the "Output Volume"can be adjusted to achieve a good level. When the level looks good, the captured response is written to `Resp.wav` by clicking the button labeled "Write Response To Disk." This was repeated for three settings of the wah pedal as described above (min, middle, and max pedal angles).

[25]`http://ccrma.stanford.edu/~jos/filters/BiQuad_Section.html`
[26]"Original CryBaby," Model GCB-95
[27]`http://www.octave.org`
[28]`http://ccrma.stanford.edu/realsimple/imp_meas/`

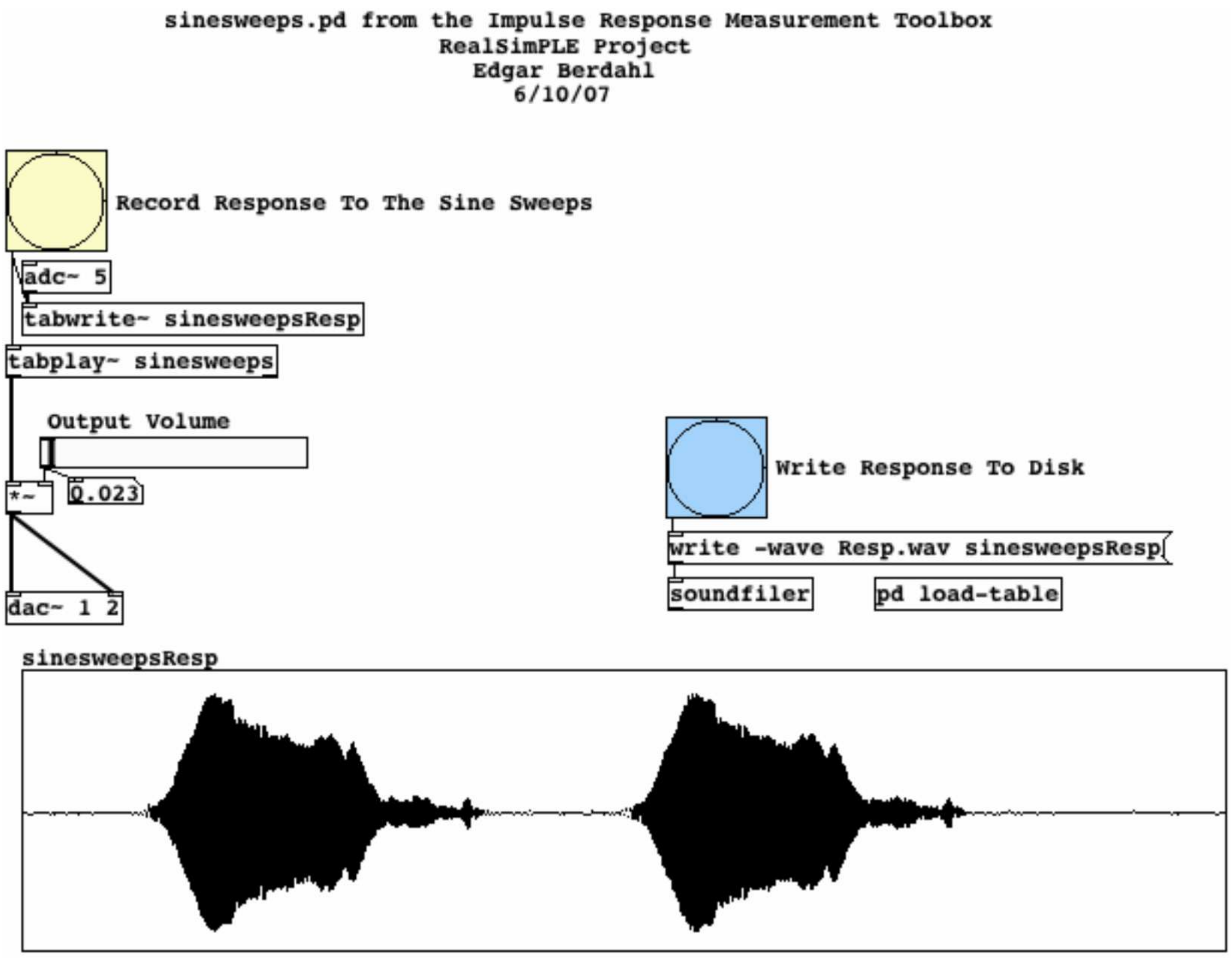

Figure 17: `sinesweeps.pd` after making a measurement with an appropriate input level (from [1]).

The captured response is in the form of a measured impulse response. The next step is to convert each of the three measured impulse responses to resonator filter coefficients. There are many ways of doing this [11]. For this exercise, the matlab[29] scripts shown in Figures 18 and 19 were used.

```
f1 = 40; f2 = 10000;   % used in filename (can't change)
f1z = 300; f2z = 3000; % zoom-in range (improves estimates)
del = [3000 2000 2000]; % measurement system delay (samples)
dur = [2048 1024 1024]; % impulse-response duration to take
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);
  if (del(i)+dur(i))>length(wahir)
    error('Signal is too short (less than system delay)');
  end
  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)  % print out estimated pole frequencies
```

Figure 18: Listing of a matlab script for estimating the Q and pole-frequency in three measured frequency responses for the CryBaby wah pedal at three different pedal settings. The function `invfreqsmethod` is defined in Fig. 19.

[29] Here, "matlab" refers to the matlab *language*, as opposed to the Matlab *product* by The Mathworks, Inc. The program `octave` (`http://www.octave.org`), including the `octave-forge` collection, was used to execute all matlab scripts in this module. In a few cases, `octplot` was used for figures in place of the standard `gnuplot` used by `octave`. All software used for this project is free and open-source, to the author's knowledge.

6.1.3 Notes regarding the matlab script in Fig. 18:

- The `del` parameter specifies the number of initial samples to skip over in the input sound (the round-trip delay of the measurement system). It can be a very critical parameter if the filter-design method is sensitive to frequency-response phase. Since `invfreqsmethod` converts the measured frequency response to minimum phase, it is not sensitive to `del` over a wide range.

- The frequency-zoom interval `f1z` to `f2z` should include the resonance peak. Since the measured response is noisier at very low and very high frequencies, frequency-zooming improves the resulting match.

- The estimated Q values printed at the end are

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

  and the estimated pole frequencies are

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

  While these estimates could be improved by various means [8], they appear to be more than sufficiently accurate for the application at hand.

- The plot overlays in Figures 14 through 16 are plots of the returned frequency responses `Hp` (measured) and `Hd` (the model) versus sampled radian frequency `w`.

6.1.4 Notes regarding `invfreqsmethod` listing in Fig. 19:

- The basic method is to use `invfreqs` to find the coefficients of a second-order *analog* filter having a frequency response close to what we measured. Then, we calculate the Q and resonance frequency ω_r from the simple formulas relating these quantities to the transfer-function coefficients:

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

  where ξ is an arbitrary finite zero location near dc. Finally, using the impulse invariant method,[30] we map the estimated Q and ω_r to digital biquad coefficients, as shown in the code.

- A useful quantitative measure of filter approximation error can be defined as follows (insert after the call to `freqs`):

```
err = norm(wt(:) .* (db(Hp(:))-db(Hph(:))))/norm(wt(:) .* Hp(:));
disp(sprintf(['Relative weighted L2 norm of frequency ',
              'db-magnitude response error = %f'],err));
```

[30]`http://ccrma.stanford.edu/~jos/pasp/Impulse_Invariant_Method.html`

```
function [Q,wp,Hp,Hd,w] = invfreqsmethod(h,f1,f2,fs);
%
% INVFREQSMETHOD - use invfreqs to estimate Q and resonance
%              frequency from a resonator impulse response.
% USAGE:
%       [Q,wp,w,Hp,Hd] = 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)
%
% invfreqs and invfreqz are VERY sensitive to where time 0
% is defined.  Normalize this by converting the impulse
% response to its minimum-phase counterpart:
h = minphaseir(h);      H = fft(h);

L0 = length(h);
k1  = round(L0*f1/fs);  k2  = round(L0*f2/fs);
Hp = H(k1:k2);          L0p = length(Hp)

w = 2*pi*[k1:k2]*fs/L0;
wt = 1 ./ w.^2; % Nominal weighting proportional to 1/freq
[Bh,Ah] = invfreqs(Hp,w,2,2,wt);
Hph = freqs(Bh,Ah,w);

% 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);
fp = wp/(2*pi);
disp(sprintf('Pole frequency = %f Hz',fp));
disp(sprintf('Q = %f',Q));

kp = L0*fp/fs - k1 + 1; % bin number of pole (+1 for matlab)
k1z = max(1,floor(kp*0.5));
k2z = min(length(w),ceil(kp*1.5));
ndx = [k1z:k2z];

% BiQuad fit using z = exp(s T) ~ 1 + sT approximation:
frn = fp/fs; % Normalized pole frequency (cycles per sample)
R = 1 - pi*frn/Q; % pole radius
theta = 2*pi*frn; % pole angle
A = [1, -2*R*cos(theta), R*R];
B = [1 -1]; % zeros guessed by inspection of amp response
Hd = freqz(B,A,w/fs);
gain = max(abs(Hp))/max(abs(Hd))
B = B*gain;  Hd = Hd*gain;
```

Figure 19: Listing of matlab code for estimating the Q and pole-frequency in the measured frequency response of a second-order resonator (wah pedal).

- The `minphaseir` function for converting a spectrum to its minimum-phase counterpart is listed in Fig. 20 and discussed in [12].[31] This is a time-domain version of `mps.m` from the RTFMT matlab code.[32] The support routines `clipdb` and `fold` are similarly listed and discussed in [12] (and included with the RTFMT matlab code).

  Conversion to minimum phase is an important preprocessing step for phase-sensitive filter-design methods when the desired frequency response is given as the FFT of a measured impulse response with an unknown excess delay. Otherwise, the leading zeros can be trimmed away manually. Further discussion on this point appears in [11].

  Note that `minphaseir` will give poor results (in the form of time-aliasing) if there are poles or zeros too close to the unit circle in the z plane. To address this, the spectrum of `h` can be *smoothed* to eliminate any excessively sharp peaks or nulls. The requirement is that the inverse-FFT of the *log* magnitude spectrum `H = fft(h)` must not time-alias appreciably at the FFT size used (which is determined by the smoothness and the amount of zero padding used in the FFT).

- The approximation $z = e^{sT} \approx 1 + sT$ assumes the highest resonance frequency (measured to be 2.2 kHz here—see Fig. 16) is much less than the sampling rate f_s. Since we will use sampling rates no lower than $f_s = 44.1$ kHz, and since the resonance frequency does not need to be exact, this approximation is adequate for wah-pedal simulation.

```
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)

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));
```

Figure 20: Listing of matlab function `minphaseir` for converting an impulse response to its minimum-phase counterpart.

In summary, a second-order analog transfer-function was fit to each RTFMT-measured frequency response using `invfreqs` in Octave. Closed-form expressions relating the returned coefficients to Q, peak-frequency, and peak-gain were used to obtain these parameters.

[31]`http://ccrma.stanford.edu/\char‘~jos/filters/Minimum_Phase_Polynomials.html`
[32]`http://ccrma.stanford.edu/realsimple/imp_meas/tf_meas.zip`

6.1.5 The Digital CryBaby in Faust

The following Faust code approximately interpolates among the three measured pedal settings as a function of a single "wah" variable normalized to lie between 0 and 1:

```
Q  = pow(2.0,(2.0*(1.0-wah)+1.0));
fr = 450.0*pow(2.0,2.3*wah);
g  = 0.1*pow(4.0,wah);
```

Closed-form expressions for biquad coefficients in terms of `(Q,fr,g)` based on $z = \exp(sT) \approx 1+sT$ (low-frequency resonance assumed) yield the following Faust code:

```
// BiQuad fit using z = exp(s T) ~ 1 + sT for low frequencies:
frn = fr/SR; // Normalized pole frequency
R = 1 - PI*frn/Q; // pole radius
theta = 2*PI*frn; // pole angle
a1 = -2.0*R*cos(theta); // biquad coeff
a2 = R*R;               // biquad coeff
// biquad denominator A = [1 a1 a2];
```

Finally, each time-varying biquad coefficient was smoothed by a unity-gain one-pole smoother with pole at $z = 0.999$.

A Faust program implementing the digital approximation to the CryBaby wah pedal is shown Fig. 21, and a test program is listed in Fig. 22. The function `crybaby(wah)` is included in `effect.lib` starting with Faust version 0.9.9.3.

To test that the Faust version is producing the correct response, replace the last line in Fig. 22 with the following:

```
process = 1-1' : *(gs) : wahres;
```

The signal `1-1'` is an impulse, so this process statement produces the *impulse response* of the wah model at its default setting (`wah = 0.1`) which can be edited to check different values. The amplitude response can then be seen by running

```
faust2octave tcrybaby.dsp
```

followed by (in Octave)

```
freqz(faustout);
```

```
// Excerpt from effect.lib (Faust 0.9.9.3)
//
//------------------------ crybaby(wah) ----------------------------
// Digitized CryBaby wah pedal
// USAGE: crybaby(wah), where wah = "pedal angle" from 0 to 1.
// Requires filter.lib.
// Reference "http://ccrma.stanford.edu/~jos/pasp/vegf.html";
//
crybaby(wah) = *(gs(s)) : tf2(1,-1,0,a1s(s),a2s(s))
with {
  s = 0.999; // smoothing parameter (one-pole pole location)
  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:
  a1s(s) = a1 : smooth(s);
  a2s(s) = a2 : smooth(s);
  gs(s) =  g  : smooth(s);
};
```

Figure 21: Listing of Faust program `tcrybaby.dsp` for testing the Digital CryBaby wah pedal.

```
// tcrybaby.dsp = test patch for digitized CryBaby pedal.

import("effect.lib"); // Faust 0.9.9.3 and higher

g    = hslider("level [midi: ctrl 0x7]",0.1,0,1,0.01);
wah  = hslider("wah [midi: ctrl 0x71]",0.4,0,1,0.01);
// MIDI Controller 0x71 is often "resonance" or "timbre"

// Some VST plugin hosts require stereo in and out:
process = + : *(g) : crybaby(wah) <: _,_;

// faust2octave impulse-response test using default wah-slider value:
// process = 1-1' : crybaby(wah);
```

Figure 22: Listing of Faust program `tcrybaby.dsp` for testing the Digital CryBaby wah pedal.

6.1.6 Conclusions Regarding the Digital CryBaby

The Faust-generated wah pedal sounds very accurate to the author—in fact too accurate. At low resonance frequencies, the loudness is significantly greater than at high resonance frequencies. Therefore, it is planned to determine a new scaling function `g(wah)` that preserves *constant loudness* as much as possible as the pedal varies.

6.1.7 Laboratory Exercises Using the Digital CryBaby

1. Print a plot of the amplitude response for the `wah` parameter set to 0, 0.5, and 1.
2. Using the Faust architecture file `bench.cpp`, determine the number of cycles required by the Digital Crybaby.
3. (Optional) Find a new scaling function `g(wah)` that preserves *constant loudness* as much as possible as the pedal varies.

7 Conclusions

In this series of laboratory exercises, we built a variety of virtual stringed instruments and associated effects in the Faust programming language which compiled to produce C++ code for a variety of hosting environments. We hope you have built some plugins that you will enjoy using in your personal music studio.

References

[1] E. J. Berdahl and J. O. Smith, "Transfer function measurement toolbox," June 2007, `http://ccrma.stanford.edu/realsimple/imp_meas/`.

[2] H. Chamberlin, *Musical Applications of Microprocessors*, New Jersey: Hayden Book Co., Inc., 1980.

[3] J. Dattorro, "The implementation of recursive digital filters for high-fidelity audio," *Journal of the Audio Engineering Society*, vol. 36, pp. 851–878, Nov. 1988, Comments, ibid. (Letters to the Editor), vol. 37, p. 486 (1989 June); Comments, ibid. (Letters to the Editor), vol. 38, pp. 149-151 (1990 Mar.).

[4] D. A. Jaffe and J. O. Smith, "Extensions of the Karplus-Strong plucked string algorithm," *Computer Music Journal*, vol. 7, no. 2, pp. 56–69, 1983.

[5] M. Karjalainen and U. K. Laine, "A model for real-time sound synthesis of guitar on a floating-point signal processor," in *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing, Toronto*, vol. 5, (New York), pp. 3653–3656, IEEE Press, May 1991.

[6] M. Karjalainen, V. Välimäki, and T. Tolonen, "Plucked string models: From the Karplus-Strong algorithm to digital waveguides and beyond," *Computer Music Journal*, vol. 22, pp. 17–32, Fall 1998, available online at `http://www.acoustics.hut.fi/~vpv/publications/cmj98.htm`.

[7] K. Karplus and A. Strong, "Digital synthesis of plucked string and drum timbres," *Computer Music Journal*, vol. 7, no. 2, pp. 43–55, 1983.

[8] N. Lee and J. O. Smith, "Virtual stringed instruments," Dec. 2007, `http://ccrma.stanford.edu/realsimple/phys_mod_overview/`.

[9] Y. Orlarley, A. Gräf, and S. Kersten, "DSP programming with Faust, Q and SuperCollider," in *Proceedings of the 4th International Linux Audio Conference (LAC2006), http://lac.zkm.de/2006/proceedings.shtml*, 2006, `http://www.grame.fr/pub/lac06.pdf`.

[10] B. Santo, "Volume cranked up in amp debate," *Electronic Engineering Times*, pp. 24–35, October 3, 1994, `http://www.trueaudio.com/at_eetjlm.htm`.

[11] J. O. Smith, *Techniques for Digital Filter Design and System Identification with Application to the Violin*, PhD thesis, Elec. Engineering Dept., Stanford University (CCRMA), June 1983, Available as CCRMA Technical Report STAN-M-14.

[12] J. O. Smith, *Introduction to Digital Filters with Audio Applications*, `http://w3k.org/books/`: W3K Publishing, 2007, `http://ccrma.stanford.edu/~jos/filters/`.

[13] J. O. Smith, *Mathematics of the Discrete Fourier Transform (DFT), with Audio Applications, Second Edition*, `http://w3k.org/books/`: W3K Publishing, 2007, `http://ccrma.stanford.edu/~jos/mdft/`.

[14] J. O. Smith, *Physical Audio Signal Processing*, `http://ccrma.stanford.edu/~jos/pasp/`, Aug. 2007, online book.

[15] J. O. Smith, "Signal processing in Faust and Pd," Nov. 2007, `http://ccrma.stanford.edu/realsimple/faust/`.

[16] T. Stilson, *Efficiently Variable Algorithms in Virtual-Analog Music Synthesis—a Root-Locus Perspective*, PhD thesis, Elec. Engineering Dept., Stanford University (CCRMA), June 2006, `http://ccrma.stanford.edu/~stilti/`.

[17] C. R. Sullivan, "Extending the Karplus-Strong algorithm to synthesize electric guitar timbres with distortion and feedback," *Computer Music Journal*, vol. 14, pp. 26–37, Fall 1990.

[18] V. Välimäki, *Discrete-Time Modeling of Acoustic Tubes Using Fractional Delay Filters*, PhD thesis, Report no. 37, Helsinki University of Technology, Faculty of Electrical Engineering, Laboratory of Acoustic and Audio Signal Processing, Espoo, Finland, Dec. 1995, `http://www.acoustics.hut.fi/ vpv/publications/vesa_phd.html`.

[19] R. VanderKam, "class project," spring 1991.