Virtual Electric Guitars and Effects Using Faust and Octave

Julius SMITH*
Center for Computer Research in Music and Acoustics (CCRMA)
Music Dept., Stanford University, Stanford, CA 94306, USA
jos@ccrma.stanford.edu

Abstract

Implementations of virtual electric guitar components and associated effects in a Linux environment are described. The Faust language is used for component specification and Octave is used for filter design and testing. To maximize compatibility with the free-software world, only methods free of known patent restrictions are used. While most of this paper classifies as tutorial review with emphasis on implementation, the dynamic level filter and wah pedal effects are believed to be new.

Keywords

Faust, Octave, virtual electric guitar, Karplus-Strong, digital waveguide, guitar effects, plugins, wah pedal, overdrive, Moog VCF

1 Introduction

This paper describes use of the Faust programming language[1] by Yann Orlarey [8; 3] to generate real-time DSP plugins from a high-level specification. An additional introductory tutorial appears in [11].[2] More specifically, this paper discusses development (in Octave) and implementations (in Faust) of virtual electric guitar components and associated effects.

In [8], a Faust implementation of a derivative of the Karplus-Strong algorithm was presented, and further extensions appeared in the `faust2pd` distribution [3]. This paper describes (with minimized overlap) additional components for virtual electric guitars and associated effects, while avoiding any methods that are known to be patented.

* Work supported in part by the Wallenberg Global Learning Network (`http://www.wgln.org/`)

[1] The Faust home page is `http://faust.grame.fr/`. Faust is included in the Planet CCRMA distribution (`http://ccrma.stanford.edu/planetccrma/software/`). The examples in this paper have been tested with Faust version 0.9.9.3.

[2] The same tutorial is available online at `http://ccrma.stanford.edu/realsimple/faust/`

2 String Simulation

As detailed in [13],[3] the Karplus-Strong (KS) *digitar* algorithm [6] can be interpreted as a simplified digital waveguide synthesis model for an "idealized" vibrating string (no stiffness, and very specific damping characteristics), thereby giving it a physical interpretation that can guide extensions to the basic algorithm. For further information regarding KS algorithms, see the references and Faust software distribitions.

The Extended Karplus-Strong (EKS) algorithm [4] extends the KS digitar in a number of ways that were motivated by the demands of a musical composition,[4] and by the interpretation of the KS algorithm as a transfer-function model of a simplified physical string [10, pp. 158–198]. They illustrate how several small digital filters can achieve various desired musical effects.

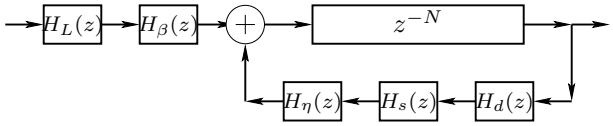

Figure 1: EKS high-level block diagram.

Figure 1 illustrates where the various filters may be located in the EKS patch.

2.1 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 [13]. 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; 10].

The EKS pick-position comb filter has transfer function

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

where P is the period (total loop delay) in samples, $\beta \in (0,1)$ denotes normalized position

[3] `http://ccrma.stanford.edu/~jos/pasp/Karplus_Strong_Algorithm.html`

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

along the string (0 at the "bridge" and 1 at the "nut"), and $\lfloor x \rfloor$ means the "greatest integer less than or equal to x." This transfer-function model of pick position is easily derived by factoring the transfer-function of the string from the picking point to any other point along the string, such as the bridge point [13; 10].

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

```
pickpos = _ <: delay(Pmax,beta*P) : - ;
```

where `Pmax` is some power of 2 larger than `beta*P` (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);
```

2.2 EKS String Damping Filter

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

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

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. 1 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.

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

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

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

```
t60 = hslider("decaytime_T60_sec",
              4, 0, 10, 0.01);
rho = pow(0.001,1.0/(freq*t60));
```

where `freq` is the fundamental frequency.

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

2.3 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 [17]

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

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. 1.

Faust Implementation

```
t60 = hslider("decaytime_T60_sec",
              4, 0, 10, 0.01);
B = hslider("brightness",
              0.5, 0, 1, 0.01);

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''));
```

For a derivation, see [13]:[5]

2.4 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 left in Fig. 1) qualitatively models this phenomenon:[6]

$$H_L(z) \;=\; \frac{1 - R_L}{1 - R_L z^{-1}}$$

This is a 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 using methods that are applicable in other situations as well.

[5] `http://ccrma.stanford.edu/~jos/pasp/-Length_FIR_Loop_Filter.html`

[6] 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.

We assume that the ideal *continuous-time* filter has the transfer function

$$H_L(s) = \frac{\omega_1}{s + \omega_1} \tag{2}$$

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 at -6 dB/octave for $\omega \gg \omega_1$. It also happens to be the 1st-order Butterworth lowpass with its cut-off frequency set to ω_1 rad/sec. To achieve the dynamic level effect, the output of this filter is linearly panned with its input. That is, at maximum level L, the lowpass filter is bypassed. 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.*, $L_0(L) = L^{1/3}$). Figure 3 shows a family of filter responses at four different dynamic levels and six different fundamental frequencies.

A Faust implementation of the L calculation is as follows:

```
(L,L0) = hslider("dynamic_level",
         -10, -60, 0, 1) : db2linear;
L0 = pow(L,1/3);
```

In [12], the *impulse-invariant* and *bilinear transform* methods are compared for digitizing the dynamic-level analog filter Eq. (3), 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

$$H_L(z) = \frac{\tilde{\omega}}{1 + \tilde{\omega}} \frac{1 + z^{-1}}{1 - \left(\frac{1-\tilde{\omega}}{1+\tilde{\omega}}\right) z^{-1}} \tag{3}$$

with $\tilde{\omega} \triangleq \omega_1 T/2$.

Figure 2 shows a family of magnitude responses. 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.

Faust Implementation

```
levelfilter(L,x) = L * L0 * x
          + (1.0-L) * lp2out(x)
```

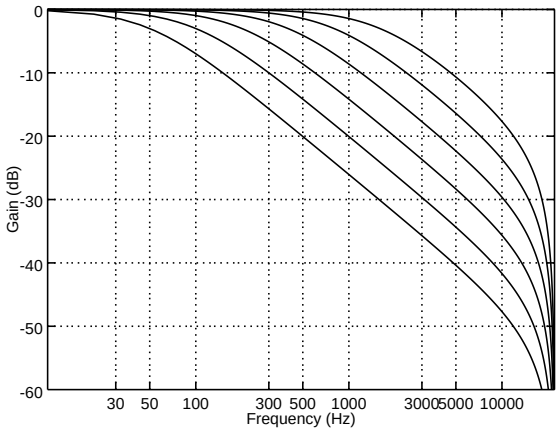

Figure 2: Dynamic level lowpass filter designed by the bilinear-transform method with $L = 0$. The filter amplitude response is plotted for 6 values of break frequency (50, 100, 200, 400, 800, and 1600 Hz). The sampling rate is $f_s = 44100$ Hz.

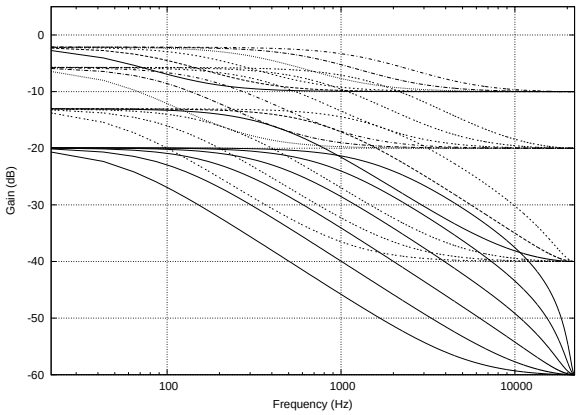

Figure 3: Dynamic level lowpass filter responses as in Fig. 2, but with $L = 0.001$, 0.01, 0.1, and 0.32, corresponding to desired Nyquist-limit levels of -60, -40, -20, and -10 dB, respectively. The dc level is defined to be one-third the Nyquist-limit level.

```
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);
};
```

2.5 Tuning the EKS String

At low sampling rates and/or high fundamental frequencies, the string simulation can sound "out of tune" if the main delay-line length N is restricted to integer values. This means an *interpolating delay line* is needed. An overview of linear, allpass, and Lagrange interpolation, among

others, is included in [13].[7] The transfer function of first-order linear-interpolation can be written

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

where $\eta \in [0,1]$ denotes the desired delay (in samples) at the low-frequency limit.

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.

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 sampling rate of 44.1 kHz) will jump out as "buzzy" when they correspond to a nearly integer delay-line length. This artifact diminishes with oversampling factor, of course.

In the original EKS algorithm [4], first-order *allpass interpolation* was chosen in order to avoid affecting the damping of the string loop. Researchers at the Helsinki University of Technology have historically used Lagrange interpolation for digital-waveguide fine-tuning [16; 5]. This has the advantage of being more robust under rapidly time-varying conditions, but introducing some gain error in the string feedback loop.

For Faust implementation, Lagrange interpolation is very cleanly implemented analogous to `fdelay(n,d,x)` in `music.lib` (linear interpolation is in fact first-order Lagrange interpolation). For example, the fourth-order case may be written as follows:

[7] `http://ccrma.stanford.edu/~jos/pasp/Delay_Line_Interpolation.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;
};
```

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, orders 1 through 4, are included in the Faust library file `filter.lib` (as of version 0.9.9.3). Further discussion appears in [12].

3 Distortion and Amplifier Feedback

The addition of *distortion* and *amplifier feedback* to the basic EKS algorithm was introduced in [15]. The resulting synthesis model is capable of convincing synthesis of "howling" overdriven electric guitars.[8]

3.1 Cubic Nonlinear Distortion

To minimize aliasing, it is helpful to use nonlinearities that are approximated by polynomials of low order.[9] An often-used *cubic nonlinearity* is given by [15]

$$f(x) = \begin{cases} -\frac{2}{3}, & x \le -1 \\ x - \frac{x^3}{3}, & -1 < x < 1 \\ \frac{2}{3}, & x \ge 1. \end{cases} \qquad (4)$$

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 [13].[10] 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

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

[9] The `faust-pd` distribution includes a "Fuzz effect" based on taking an absolute value in the file `karplusplus.dsp`.

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

commercial guitar distortion simulators are said to use oversampling factors of 4 to 8.

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 [9]. It was found empirically that a dc blocker [11] was needed to keep the signal properly centered in the output dynamic range. Since amplifier loudspeakers have a +12 dB/octave low-frequency response, at least two dc blockers may be used in a typical modeling framework anyway.

While the cubic nonlinearity is the odd nonlinearity with the least aliasing (thereby minimizing oversampling and guard-filter requirements), it can be 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. A diode clipping curve is typically modeled as a hyberbolic tangent function.

Faust Implementation

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

```
cubicnl(drive,offset) = +(offset)
  : *(pregain) : cubicnl : *(postgain)
  : dcblocker
with {
  tt = 2.0/3.0;
  stage1(x) = select2(x>(-1.0), 0-tt,
              x - pow(x,3)/3.0);
  cubicnl(x) = select2(x>1.0,
               stage1(x), tt);
  pregain = pow(10.0,2*drive);
  postgain = max(1.0,1.0/pregain);
};
```

3.2 Amplifier Feedback

In [15], 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 the `process` statement of `freeax.dsp` [12].)

Segue to the Present

We now leave our more or less historically rooted review of virtual electric guitar developments, and continue with selected topics.

4 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.

4.1 Digitizing the CryBaby

Figure 4 (solid line) shows the amplitude response of the author's CryBaby wah pedal measured (as described in §4.1.2 below) at the middle pedal setting. Our goal is to "digitize" the CryBaby by devising a second-order sweeping resonator that audibly matches measurements such as this for various pedal angles.

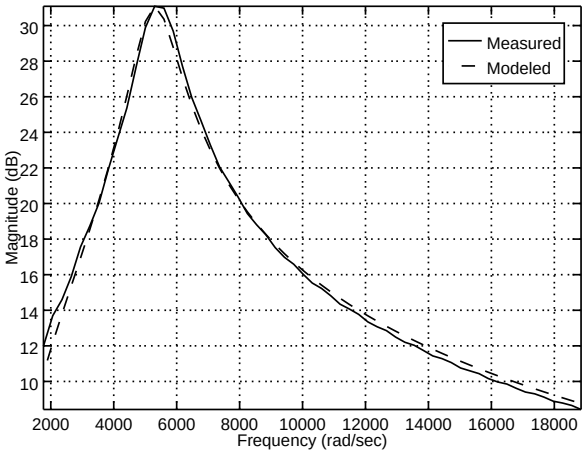

Figure 4: Measured (solid line) and modeled (dashed line) amplitude responses of the CryBaby pedal set to the middle of its excursion.

4.1.1 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* [14]. However, the measurements described below revealed that resonance frequency, Q, and gain all varied 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 [11][11] to implement the wah resonator.

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

[11] `http://ccrma.stanford.edu/~jos/filters/-BiQuad_Section.html`

three measured frequency responses and coming up with an interpolation formula for the varying coefficients.

4.1.2 Measuring the CryBaby

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[12] 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 like the one shown in Fig. 4 were carried out in `pd` and Octave[13] using software from the RealSimple Transfer Function Measurement Toolbox (RTFMT) [2].[14] 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 (also distributed with the RTFMT), which plays the sinesweep and records the response. This was repeated for three settings of the wah pedal as described above (min, middle, and max pedal angles).

The next step is to convert each of the three measured impulse responses to resonator filter coefficients. There are many ways of doing this [10]. For this exercise, matlab scripts running in Octave were used [12].[15] Briefly, 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. Finally, the following Faust code was used to approximately interpolate among the three measured 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)` were implemented based on $z = \exp(sT) \approx 1 + sT$ (low-frequency resonance assumed) to obtain

```
frn = fr/SR; // fr normalized
R = 1 - PI*frn/Q; // pole radius
theta = 2*PI*frn; // pole angle
a1 = -2.0*R*cos(theta); // biquad a1
a2 = R*R;               // biquad a2
// biquad denominator A = [1 a1 a2];
```

Finally, each time-varying BiQuad coefficient was smoothed by a unity-gain one-pole smoother with pole at $z = 0.999$. For remaining details, see `crybaby(wah)` in `effect.lib` distributed with `faust-0.9.9.3`. For further discussion, see [12].

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.

4.2 Wah Pedal Based on the Moog VCF

Voltage Controlled Filters (VCFs) used by analog *synthesizers* (as opposed to wah pedals) are typically based on the Moog VCF [14]. The Moog VCF can also be described as a variable lowpass with corner resonance, and therefore it can be used to implement a wah effect. However, because the Moog VCF employs a *fourth-order* feedback loop, the effect is stronger than the second-order case of the previous section. Specifically, while the gain from dc to resonance can be made quite similar, the fourth-order case exhibits a -24 dB per octave rolloff above resonance ("cutoff"), while the classic wah (being second-order) has only at most a -12 dB/octave rolloff.

As derived in [14], the Moog VCF cutoff (resonance) frequency f_r is controlled by varying a single real pole p_a in the s plane (the subscript 'a' here stands for 'analog'). At high Q (quality

[12] "Original CryBaby," Model GCB-95

[13] The program `octave` (`http://www.octave.org`), including with the `octave-forge` collection, was used to execute all matlab scripts in this paper. (Here, "matlab" (uncapitalized) refers to the matlab *language*, as opposed to the Matlab *product* by The Mathworks, Inc.) 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.

[14] `http://ccrma.stanford.edu/realsimple/imp_meas/`

[15] Space limitations do not permit listing the matlab scripts here, but they are downloadable from the URL given for [12], and further discussion appears there and in [13].

factor) [11], the resonance frequency ω_r is given approximately in radians per second by

$$\omega_r = 2\pi f_r = |p_a|.$$

That is, the pole magnitude is approximately equal to the resonance frequency.

There are various ways to digitize an analog system such as the Moog VCF [14]. Perhaps the simplest method is the so-called *impulse-invariant method*, in which the analog pole at $s = p_a$ in the Laplace domain (s-plane) maps to $p_d = e^{-p_a T}$ in the z-plane, where T denotes the sampling interval in seconds. When the sampling rate $f_s = 1/T$ is much higher than the resonance frequency f_r (as is typical), we may approximate the digital pole location as

$$p_d \;\triangleq\; e^{-p_a T} \;\approx\; 1 - p_a T \;=\; 1 - 2\pi\frac{f_r}{f_s}.$$

A Faust implementation is quite simple:

```
onepole(p) =  *(1.0-(p)) : + ~ *(p);
moogvcf(mk,p) = (+ : onepole(p)
      : onepole(p) : onepole(p)
      : onepole(p)) ~ *(mk);
wah4(fr) = moogvcf(-3.8,pole(fr))
with
{ pole(fr) = 1.0-fr*2.0*PI/SR; };
```

This code comprises `wah4` in `effect.lib` distributed with `faust-0.9.9.3`. The choice $Q = 3.8$ was an arbitrary preference.

5 Generalized Pickup/Pick-Position

Given a digital-waveguide model of a vibrating string, using velocity waves, we have that a virtual pickup is given by the sum of two delay-line taps, while a virtual excitation is given by summing into two delay-line cells. Thus, an excitation point is formally the *transpose* [11][16] of a pickup point.

Since the pick-position comb-filter may be implemented as a pickup comb-filter (*i.e.*, moved from the input side of the filtered delay loop to the output side), it follows that both excitation points and pickup points correspond to delay-line taps that are summed. If $\underline{x}(n) = [x_0(n), x_1(n), \ldots, x_N(n)]$ denotes the state of the delay line in the filtered delay loop at time n, then the output signal $y(n)$ is given as a $1 \times N$ *mixing matrix*[17] $\mathbf{C}$ times the state vector:

$$y(n) = \mathbf{C}\underline{x}(n)$$

The mixing matrix $\mathbf{C}$ typically contains a 1 and a -1 for each input or output point in a velocity-wave simulation. (The -1 implements the effect of the inverting reflections at the string terminations for velocity waves.)

This suggests a generalized string simulation in which the string is driven and observed at an arbitrary set of points along its length, *i.e.*, $\underline{y}(n) = \mathbf{C}\underline{x}(n)$ where $\mathbf{C}$ is any mixing matrix whatsoever, and $\underline{y}(n)$ is a *vector* of outputs (*e.g.*, dimension 2 for a stereo output, or dimension 8 for an 8-channel output).

In other words, up to N linearly independent output signals may be formed by means of different linear combinations of the delay-line contents. For example, one can assign each "virtual pickup" created in this way to its own loudspeaker.

A very simple special case consists of using two taps with a variable separation in order to provide a stereo "width" parameter analogous to that used in `freeverb`:[18]

```
W = hslider("center-panned spatial width",
            0.5, 0, 1, 0.01);
A = hslider("pan angle", 0.5, 0, 1, 0.01);
widthdelay = delay(Pmax,W*P/2);
stereopanner(A) = _,_ : *(1.0-A), *(A);
process = stringsynth <: _,_
        : widthdelay : stereopanner(A);
```

For this Faust specification, the mono `stringsynth` output is branched so that one of the branches can be delayed to provide the "width" parameter. This sounds equivalent to a second read-pointer in the main string delay line.

6 Conclusion

The Faust implementations discussed above (and more) may be found in the files `freeax.dsp` [12], and in the libraries `effect.lib` and `filter.lib` distributed with Faust release 0.9.9.3 or later, and described more fully in [12]. An expanded (and still growing) version of this paper is included in [13].[19]

[16] `http://ccrma.stanford.edu/~jos/filters/-Transposed_Direct_Forms.html`

[17] The term "mixing matrix" comes from the subject of *artificial reverberation* where it refers to the output matrix used to form multichannel audio outputs from the internal states of a reverberator.

[18] `http://ccrma.stanford.edu/~jos/pasp/Freeverb.html`

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

7 Future Work

Additional features that time and space do not permit in this short paper include generalization of the wah pedal to a *vowel pedal*,[20] design of a *string-stiffness allpass* using the method of [1], measurement-based damping filters [5; 7], a *tone stack* based on [18], and a speaker cabinet model again using software from [2]. Additionally useful are an initial frequency skew to give the "twang" effect associated with low-tension strings, dual delay lines for string-coupling/beating/two-stage-decay effects, explicit plucking/striking/bowing models (or at least an initial string state estimated from real recordings to replace the noise burst). Details (and Faust implementations) are planned for a future appendix of [13].

8 Acknowledgements

Thanks to Yann Orlarey (Faust creator), Albert Gräf (Q and `faust2pd`), David Yeh (CCRMA EE graduate student), and Jonathan Abel (CCRMA Consulting Prof.) for their remote assistance while this work was being carried during a one-quarter sabbatical leave, fall 2007. Thanks also to CCRMA graduate students Ed Berdahl (EE) and Nelson Lee (CS) for editorial input.

References

[1] J. Abel and J. O. Smith, "Robust design of very high-order allpass dispersion filters," *Proc. Int. Conf. Digital Audio Effects (DAFx-06), Montreal, Canada*, Sept. 2006, `http://www.dafx.de/`.

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

[3] A. Gräf, "Interfacing Pure Data with Faust," in *Proc. 5th Int. Linux Audio Conf. (LAC2007), http://www.kgw.tu-berlin.de/~lac2007/-proceedings.shtml*.

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

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

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

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

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

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

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

[11] J. O. Smith, *Introduction to Digital Filters with Audio Applications*, `http://ccrma.stanford.edu/~jos/filters/`, Sept. 2007, online book.

[12] J. O. Smith, "Making virtual electric guitars and associated effects using Faust," Dec. 2007, `http://ccrma.stanford.edu/realsimple/-faust_strings/`.

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

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

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

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

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

[18] D. Yeh and J. O. Smith, "Discretization of the '59 Fender Bassman tone stack," *Proc. Int. Conf. Digital Audio Effects (DAFx-06), Montreal, Canada*, Sept. 2006, `http://www.dafx.de/`.

[20] `http://www.geofex.com/Article_Folders/-wahpedl/voicewah.htm`