EIN: A Signal Processing Scratchpad

Paul Lansky
Dept. of Music
Princeton University
Princeton, NJ 08544
paul@silvertone.princeton.edu

Kenneth Steiglitz
Dept. of Computer Science
Princeton University
Princeton, NJ 08544
ken@cs.princeton.edu

Computer systems designed for music synthesis usually encapsulate signal-processing algorithms as macros or functions, and thus provide a modular interface which facilitates the development of complex structures. This is the thinking behind Mathews' original concept of the ''unit generator'' (Mathews 1969) and most subsequent software synthesis languages use this approach. While it has proven to be an effective method it has not generally provided a means by which the users of these systems, often more musically prepared than wise in the ways of digital signal processing, can gain an intuitive understanding of the mechanisms used to modify and create digital signals. Indeed, there is little conceptual difference between patching a signal through a bank of two-pole resonating filters/unit generators and tweaking the sliders on a graphic equalizer. EIN is an attempt to provide an interface in which the user has direct control over every add, multiply and store applied to each sample, and can gain a more direct understanding of the machinery of digital signal processing. While its main use has been instructional, it also provides a way to experiment with digital filters and design more complex instruments and algorithms. It is, in effect, a kind of low-level circuit design kit for signal processing.

EIN Syntax

EIN provides the routine machinery for calling the user's code, executing it, writing the resulting output sound samples to a file, playing the file, and displaying and analyzing it in the time and frequency domains. The user provides a script in a language that is a superset of C. The EIN system then compiles it and provides a wrapper that calls the script for each time sample from `t = 0` up to the specified number of samples, `nsamps`; computes the next output sample `y` (for the mono case), or the variables `left` and `right` (for the stereo case); and writes the output samples to an output sound file, formated for the sampling rate of `sr` samples/sec.

The `int` variables `t` and `nsamps`, and the `float` variables `y`, `left`, `right`, and `sr`, are reserved by EIN, and should not be used for other purposes. The mono/stereo option, and the values of `sr` and `nsamps` are selected with radio buttons on the interface. The remaining reserved names are described in Appendix 3.

As an example, here is a one-line EIN script that produces a sine wave at 440 Hz:

```
y = sin(t*two_PI*440./sr);
```

The constant `two_PI` (2π) is provided as a convenience because it is used so often in the arguments of trigonometric functions. Of course, since this code is in a loop and called once per

Version September 26, 1994.

sample, it is more efficient and clearer to predefine the radian frequency in the argument of the sine as follows:

```
#define F 440.                  // frequency in Hz.
float omega;                    // freq. in radians per sample
if(!t) omega = two_PI*F/sr;     // one-time only initialization
y = sin(omega*t);
```

Notice that the initialization of `omega` takes place only when `t` is zero; that is, only at the first time sample.

EIN also provides one new command, called `tap`, which allows the user to store signals in delay lines for later use. This feature makes it easy to implement filters without having to fuss with the details of managing buffers. The signal value that is stored is always the value of `y` at the point that `tap` is employed. The `tap` command is invoked by a line of the form

```
tap i k
```

where `i` and `k` are two constants of type `int`. (Notice that there is no semicolon terminating the line.) This line has two effects. First, it causes the buffer `Bi` to be created, which stores the value of the signal at that point. Second, the variable `Si` is recognized anywhere else in the code as the current contents of that buffer, which is the signal at that point delayed by `k` sampling intervals. Except for the translation of `Si` variables, lines not beginning with the keyword `tap` are transparent to the EIN pre-processor, and are passed on untouched to the C compiler.

If a `tap` appears before the first use of its signal, the signal is called *feedforward*; otherwise, *feedback*.

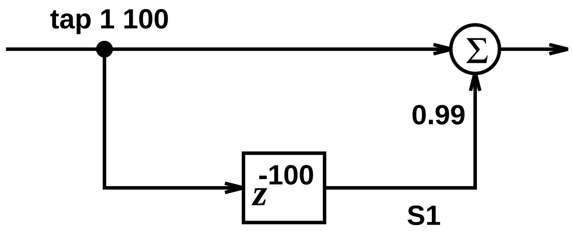

Fig. 1 Signal flowgraph of a simple feedforward filter.

To illustrate how `tap` is used, consider the simple inverse-comb filter defined by the following equation:

$$w_t = x_t + 0.99 * x_{t-100}$$

where x_t and w_t are the input and output signals, respectively. Figure 1 shows the corresponding signal flowgraph. What's important about visualizing the operation of the filter with a signal flowgraph is that it defines an order in which the calculations for each time sample are done. We can read these from left to right in Fig. 1: first we save the value of the input signal in a buffer with a delay of 100 samples, then we form the sum of the input signal and 0.99 times the delayed input. Thus, the EIN script is

```
tap 1 100
y = y + 0.99*S1;
```

The line `y = y + 0.99*S1;` does the arithmetic, while the `tap 1 100` line signals EIN to create a buffer to store the value of the signal at that point. Since the `tap` line defining `S1` appears before its first use, `S1` is a feedforward signal. When it is used in the next line, it provides a version of the input delayed by 100 samples.

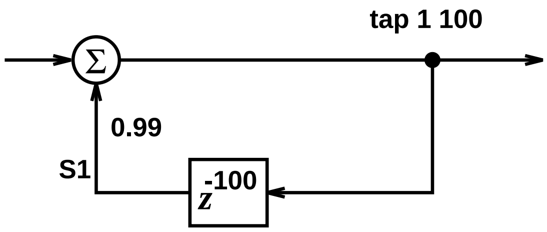

Fig. 2 Signal flowgraph of a simple feedback filter.

Figure 2 shows the signal flowgraph of a comb filter, an example of a feedback filter. Here the order of operations from left to right is reversed: first we form the sum of input and weighted delayed output, then we store the output for later use. The EIN script is therefore

```
y = y + 0.99*S1;
tap 1 100
```

which uses precisely the same two lines as the inverse comb script, but in reverse order.

EIN's `tap` command thus makes it possible to express traditional filter flowgraphs quite succinctly.

The Interface

The EIN interface is a NeXTStep application which incorporates a spectrum analysis and spectrogram of the generated signal as well as sound input and output facilities. Figure 3 shows a snapshot of a typical screen while using EIN.

The main window, in the lower left-hand corner of the screen, and shown enlarged in Fig. 4, contains a scrollview with the programming window, and buttons to compile, run, set the sampling rate, mono or stereo output, control input signals, and optionally include additional C programs or binaries in the compile step. As EIN compiles a script as a function, linking it with its driving program, additional C functions or libraries can be included by listing them in the form at the bottom of the window, labelled ‘‘include (.c/.o)’’.

The views include a zoomable amplitude/time plot (Fig. 5), an adjustable FFT (Fig 6), cued to the point of the cursor in the amplitude plot (the FFT changes as the cursor moves in the amplitude plot), and a spectrogram plot (Fig. 7): time is the horizontal axis, frequency the vertical axis, and grey-scale shows amplitude.

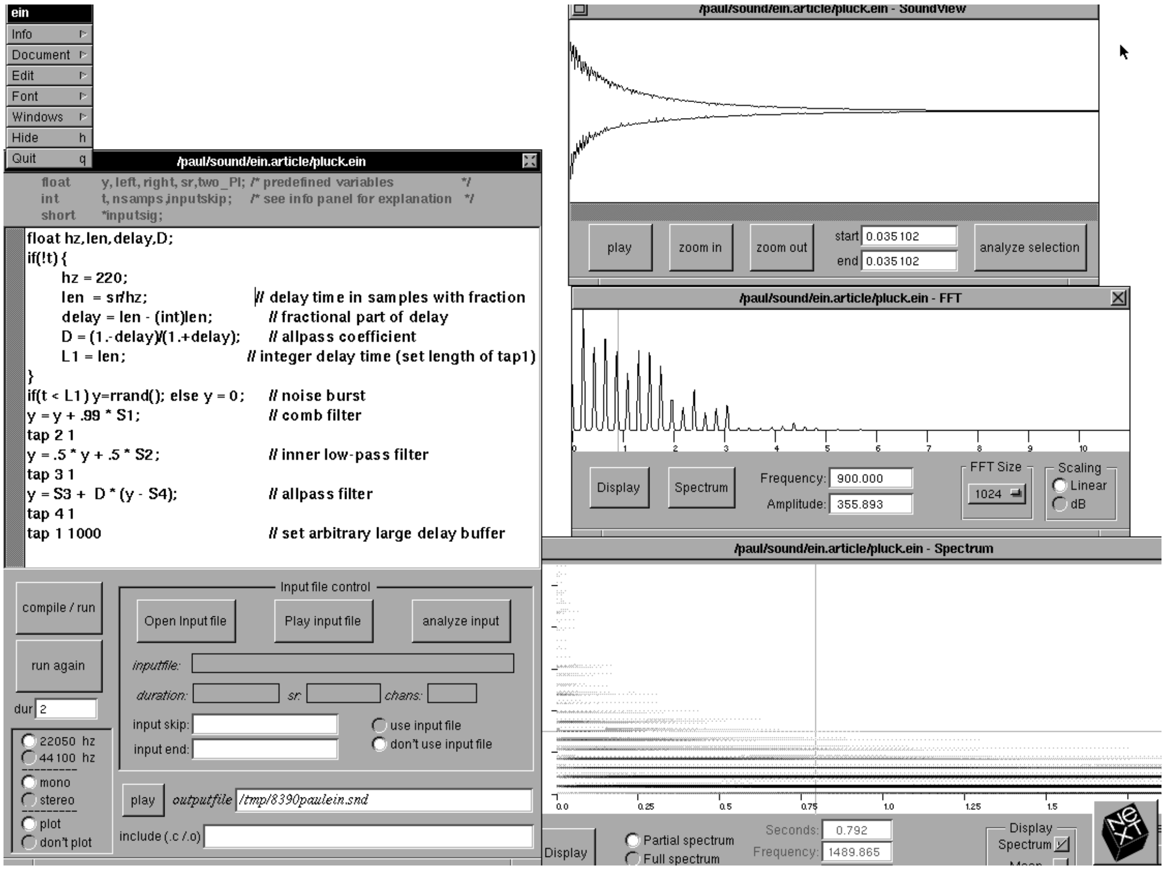

Fig. 3 Snapshot of a typical screen when using EIN.

Example: Tunable Plucked String

As a demonstration we will build a tunable plucked string filter (Jaffe and Smith 1983; Karplus and Strong 1983) in several stages. We begin with the comb filter illustrated in Fig. 2, with a resonant frequency of `sr/100` and a feedback gain coefficient of `0.99`. Its defining equation is

$$w_t = x_t + 0.99 * w_{t-100}$$

The corresponding EIN script, using a built-in white noise generator, is therefore

```
y = rrand();    // white noise generator built-in to EIN
y = y + 0.99*S1;
tap 1 100
```

Next, we create the classical plucked-string filter by limiting the length of the input signal to one pitch period and putting a feedforward lowpass filter within the feedback loop. The defining equation of the lowpass filter is

$$w_t = 0.5 * x_t + 0.5 * x_{t-1}$$

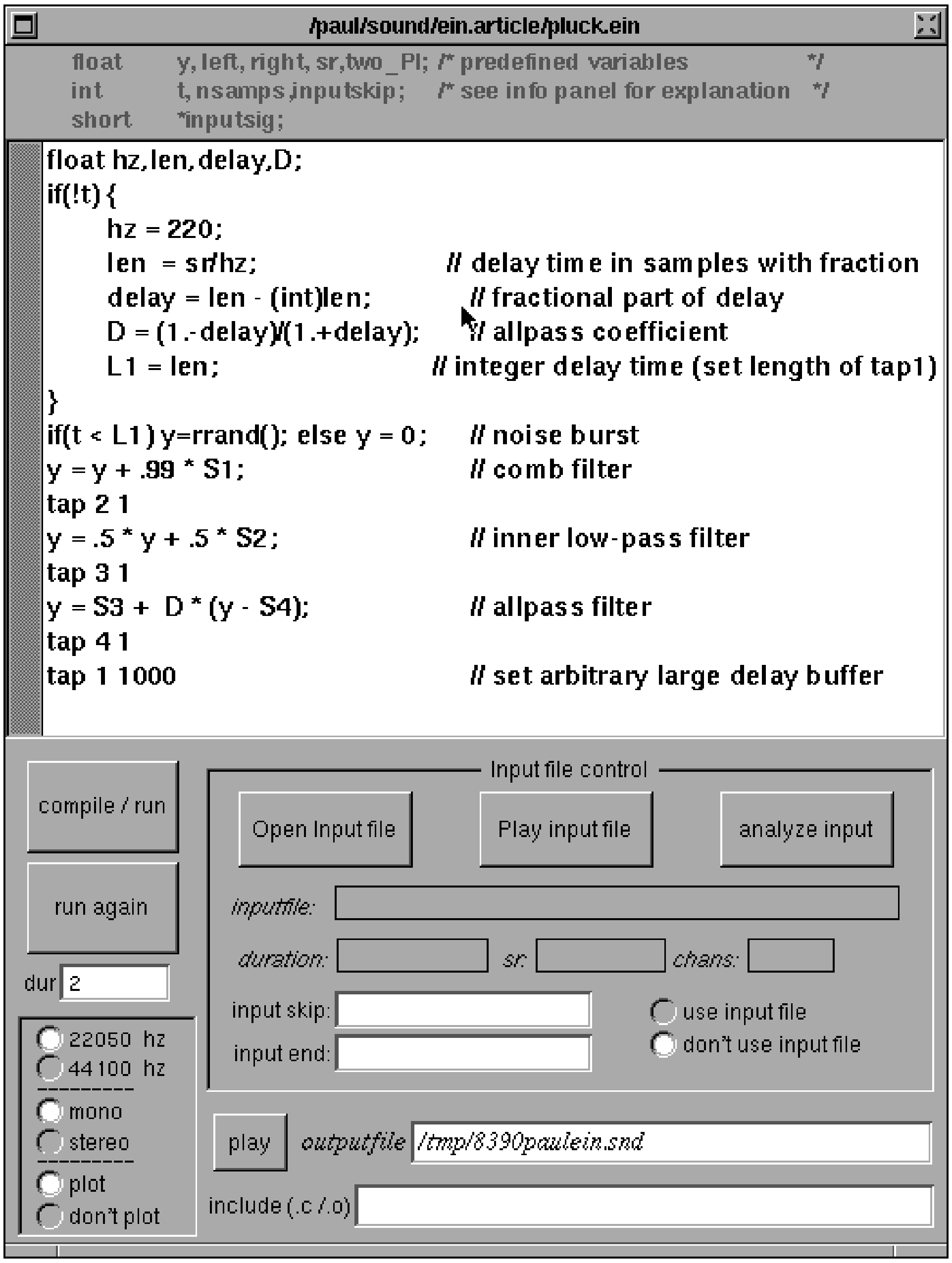

Fig. 4 Enlargement of the main window, containing script scrollview and main controls.

where *x* and *w* are its input and output respectively. Figure 8 shows the signal flowgraph of the entire plucked-string filter. The corresponding EIN script now includes a test which ensures that the input is turned off after 100 samples, and a new tap and assignment statement for the lowpass filter:

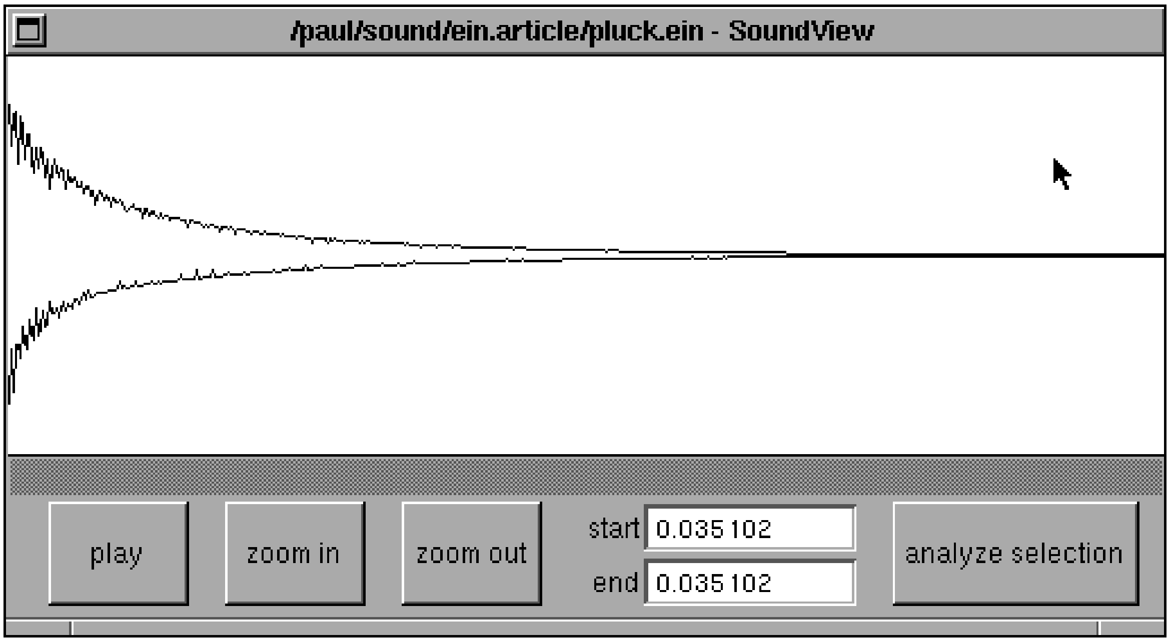

Fig. 5 Enlargement of the time waveform window.

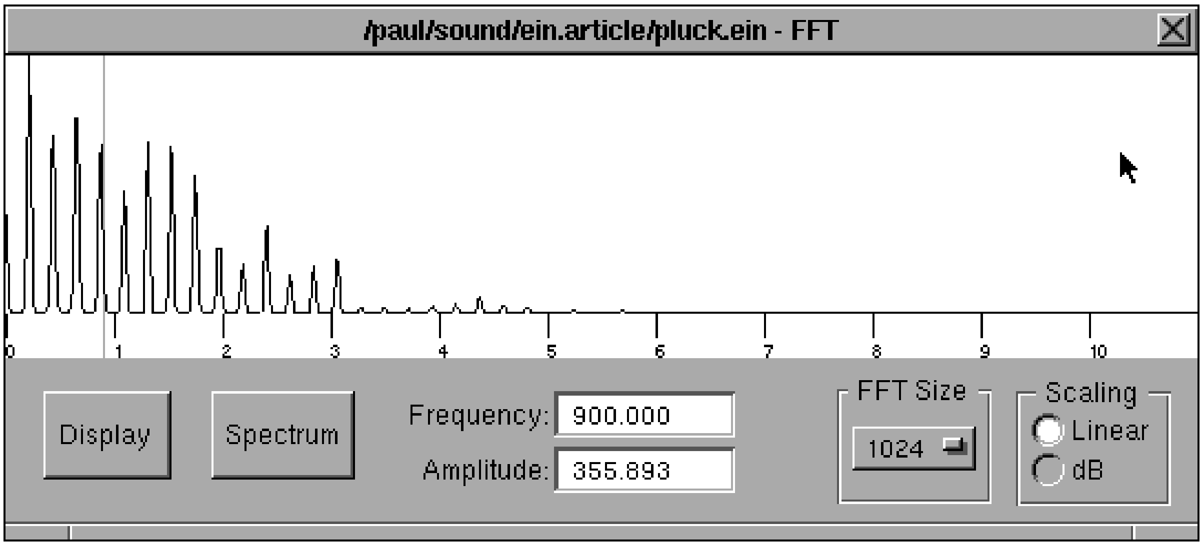

Fig. 6 Enlargement of the FFT window.

```
if(t < 100) y=rrand(); else y = 0;
y = y + .99*S1;
tap 2 1
y = .5*y + .5*S2;
```

```
tap 1 100
```

Since comb filters necessarily use an integer delay they can only resonate at frequencies that are integral divisors of the sampling rate. The standard way to tune a comb is to insert an allpass filter in the feedback loop to approximate any fractional part *d* of the desired delay. The allpass filter is implemented by the equation

$$w_t = D*(x_t - w_{t-1}) + x_{t-1}$$

Setting the allpass filter coefficient D to $(1-d)/(1+d)$ achieves a good approximation to delay d at low frequencies.

The signal flowgraph of the completed tuned plucked-string filter is shown in Fig. 9. The EIN script now includes a new feedforward and feedback `tap`, storing delayed signals `S3` and `S4` for the allpass filter.

In EIN the length of the `tap n` buffer is set to `Ln` internally. In this example we set this value ourselves to allow us to describe the pitch as a variable. The final EIN script is shown below:

```
float hz,len,delay,D;
if(!t) {                             // one-time-only initializations
        hz = 3100;                   // desired frequency
        len = sr/hz;                 // delay time in samples, with fraction
        delay = len - (int)len;      // fractional part of delay
        D = (1.-delay)/(1.+delay);   // allpass coefficient
        L1 = len;                    // integer delay (set length of tap1)
}
if(t < L1) y=rrand(); else y = 0;    // noise burst, one cycle long
y = y + .99*S1;                      // comb
tap 2 1
y = .5*y + .5*S2;                    // lowpass
tap 3 1
y = S3 + D*(y - S4);                 // allpass
tap 4 1
tap 1 1000                           // set arbitrary large delay buffer
```

Example: FM Synthesis

It is equally instructive and useful to use EIN to experiment with approaches to signal generation which do not use delays and feedforward or feedback filters. Here the value of the approach is in the conceptual simplicity of the language and the closeness between the algorithm and its expression. A simple FM sound, for example, can be expressed in four lines of code:

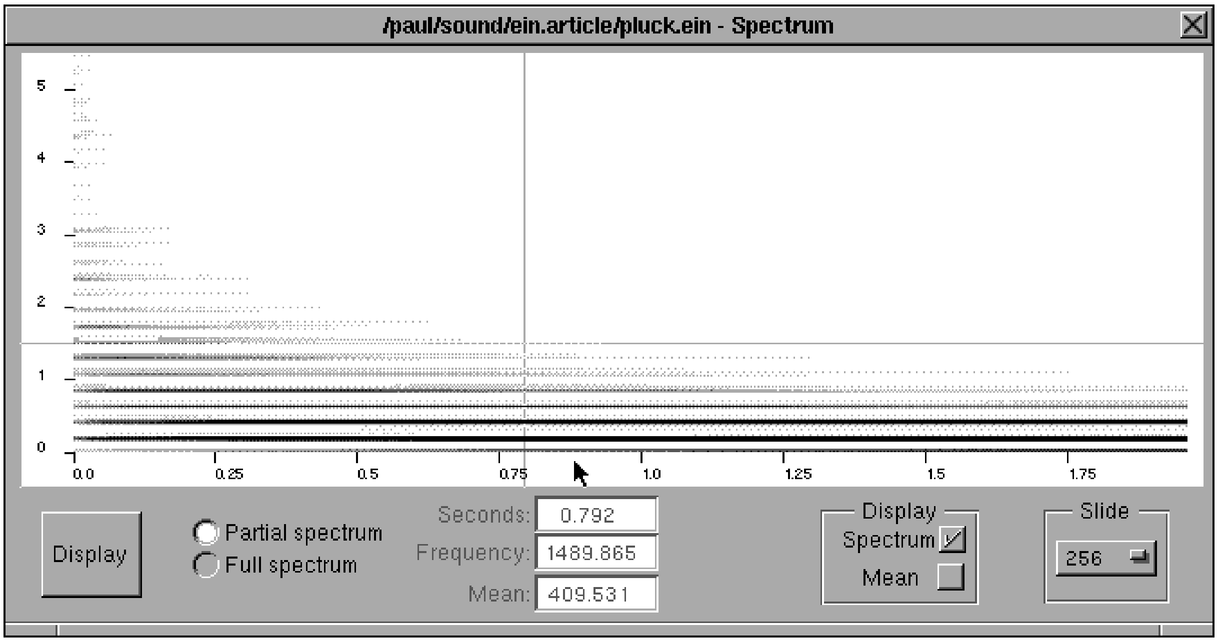

Fig. 7   Enlargement of the spectrogram window.

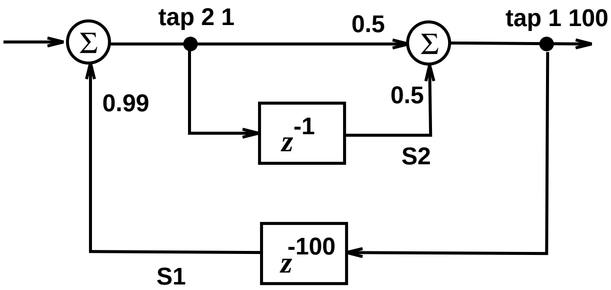

Fig. 8   Signal flowgraph of a plucked-string filter.

```
float env,line = (float)t/nsamps;   // line from 0 to 1
env = exp(-5.*(float)t/nsamps);     // exponential decay
y =10.*env*cos(133.*two_PI*t/sr);   // modulator and index of modulation
y = env*cos(y + 100.*two_PI*t/sr);  // carrier
```

Example: Cooks’ Slide Flute

Our final example is Perry Cook’s SlideFlute application (Cook 1992), a NeXT MusicKit program, expressed in the following EIN script:

```
float ysave,amp,randamp;
float line = (float)t/nsamps;
if(!t) {
        amp = 1;
        randamp = .04;
}
y=( rrand()/32767.)*randamp* amp;
y += amp;                           // breath pressure with rand dev
y = y + (S1 * -.35);                // mix  with pressure of returning wave
tap 2 10                            // embouchure, time to cross mouth hole
y = S2;
y = (y * y * y) - y;                // transfer differential across mouth hole
y = (.2*y) + (.95 * S1);            // jet stream plus full wave
ysave = y;                          // listen to output here
y = (.7 * y) + (.3 * S3);// low pass at end of tube
tap 3 1
tap 1 30                            // fundamental
y = ysave;
```

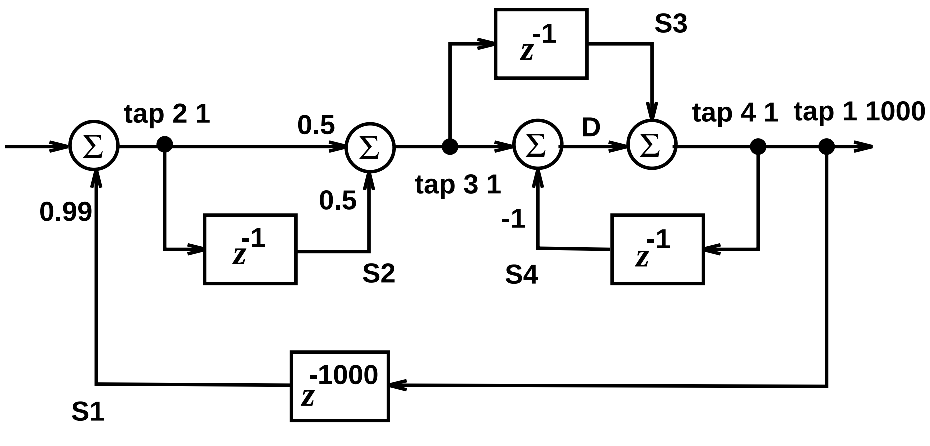

Fig. 9 Signal flowgraph of the plucked-string filter with an allpass filter added for tuning.

This is a model of a jet wind instrument. After initialization, the first two lines model a constant breath pressure with a slight random perturbation. This is then added to 35% of the phase-inverted previous returning wave, stored in tap 1. (The flute is considered a tube open at both ends. The length of tap 1 models the length of the flute — this example is really a model of a slide flute, which is constantly changing size, rather than a flute with finger holes.) The next line, $y = y^3 - y$, is a rough approximation to the action of the air blowing across the mouth hole, and 20% of this is added to 95% of the mixture of the left and right-going waves. Tap 2 models the time it takes the breath to flow across the mouth hole. Finally, a model is created of the lowpass effect created by the open end of the tube (tap 3). Cook’s signal flow graph is shown in Fig. 10.

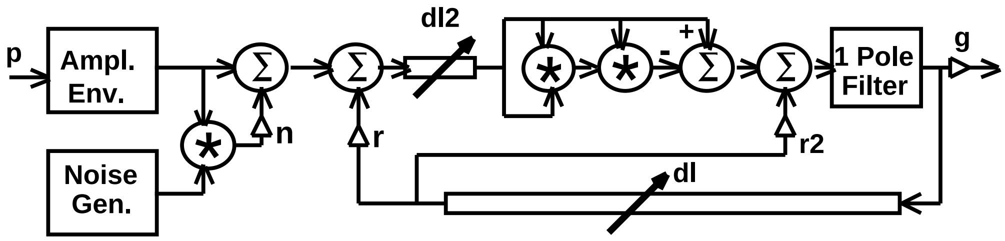

Fig. 10 Signal flow graph of Cook’s slide flute model. This picture is courtesy of Perry Cook.

Conclusion

We’ve found EIN to be useful both as a workbench for experimentation with filters and signals, and as a classroom tool while teaching introductory courses in computer music and digital signal processing. Its conceptual simplicity and freedom from elaborate syntactical rules allow it to be introduced at the earliest stages of a course. The use of multiple views of a sound, as well as auditory output, provide the student with feedback and reinforcement on several levels.

EIN is available by anonymous ftp in /pub/music on princeton.edu.

We conclude with an amusement. More than one beginning student has been tripped up by the following puzzle, whose solution we leave as an exercise for the reader. The assignment is to write an EIN script to generate a sine wave which will glissando from 2000 to 3000 Hz. The following script would seem to do the trick:

```
float line = 2000 + 1000.*(float)t/nsamps; // line from 2000 to 3000
y = sin( line*two_PI*t/sr );               // compute sine
```

but the spectrogram from EIN shown in Fig. 10 reveals that life is not always that simple...

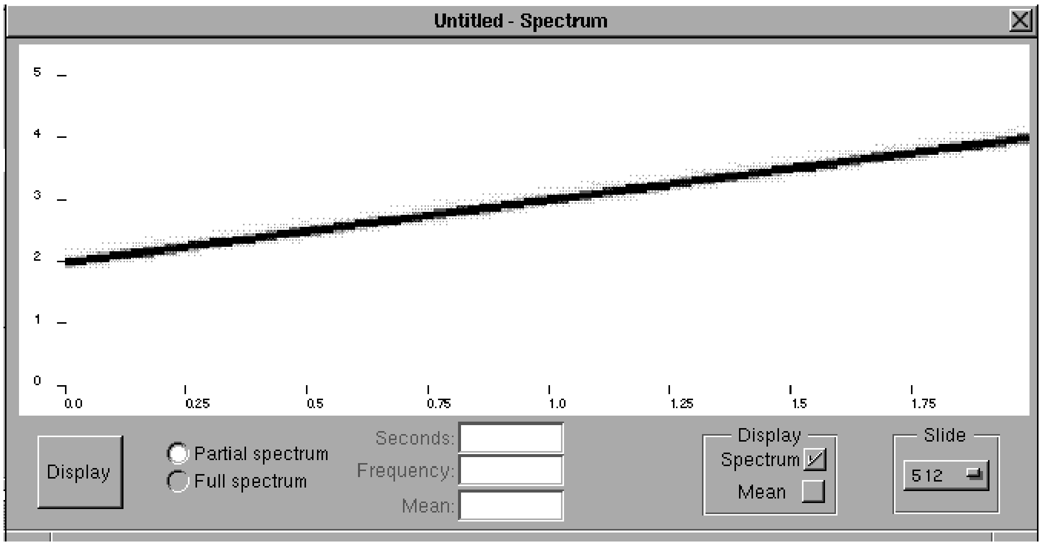

Fig. 11 Spectrogram of the signal generated by the concluding puzzle.

Acknowledgment

This work was supported by NSF Grants CDA91-15189 and MIP-9201484. The FFT and spectral displays were taken from Edsnd, by James Pritchett, with work by Steven M. Boker. Thanks to Perry Cook, who provided help with the slide flute example and permission to use Fig 10.

References

P. R. Cook. 1992. ‘‘A Meta-Wind-Instrument Physical Model and a Meta-Controller for Real Time Performance Control.’’ *Proceedings of the International Computer Music Association*, San Jose, California, Oct. 14-18, 1992, 273-276.

D. A. Jaffe and J. O. Smith. 1983. ‘‘Extensions of the Karplus-Strong Plucked-String Algorithm.’’ *Computer Music J.*, 7: 56-69.

K. Karplus and A. Strong. 1983. ‘‘Digital Synthesis of Plucked-String and Drum Timbres.’’ *Computer Music J.*, 7: 43-55.

M.V. Mathews. 1969. *The Technology of Computer Music*. Cambridge, Massachusetts: M.I.T. Press.

K. Steiglitz. 1990. ‘‘Ein Kleiner Filter Compiler.’’ Tech. Report CS-TR-279-90. Computer Science Department, Princeton University.

Appendix 1: Some Details about Buffers in EIN

The kernel of EIN, without the interface, is a very simple C pre-processor, originally only about 50 lines long (Steiglitz 1990). In a single pass of the user’s EIN script, each non-`tap` line is simply passed to the output, using the appropriate values from buffers for the `Si`. The only slightly delicate issue is the way the circular buffers for storing delayed signals are handled.

When a `tap i k` line is encountered, a new buffer `Bi` of length `N` is declared to be an array, together with the `int` index into that array, `IDi`. Then code is generated that stores the signal at that point in the buffer. For example, the script line `tap 1 100` used to save the feedback signal in a comb filter generates the code

```
B1[ID1++] = y;
ID1 = ID1%L1;
```

where the integer variable `L1` is equal to 100. The index `ID1` is incremented after it is used, and then taken modulo the length of the buffer.

The only tricky point concerns the definition of the length `N` of the buffer. Suppose `L` is the desired loop delay, the parameter in the EIN script. When the loop is a feedback loop, `N = L`; when the loop is a feedforward loop, `N = L + 1`. It is then not hard to verify that the buffer value `Bi[IDi]` provides a signal with the desired delay. The extra buffer storage location in the case of a feedforward loop is necessary because in that case the `tap` command is encountered during the same ‘‘clock’’ cycle that the signal is used.

It also follows from this arrangement that the case `L = 0` is allowed for a feedforward loop, corresponding to a buffer of length 1; the present signal value is simply saved for use during the same clock cycle of the filter. This is critical for the generality of the specification language, discussed in Appendix 4. A signal may be used both before and after its corresponding `tap`, but the user must remember that because the signal is used before its `tap`, the length of its buffer is that of a feedback signal, and uses after its `tap` are delayed one fewer sampling interval than the designated delay. As an example, if signal `S1` is used both before and after `tap 1 1`, uses before the `tap` are delayed 1 interval, but uses after are delayed 0 intervals.

Appendix 2: Warning

The example of the tuned plucked-string filter illustrated how the internally generated variable `L1` can be changed by the knowledgeable user. The other side of the coin is that the internally generated variable names of the form `Li`, `Bi`, and `IDi` are reserved by EIN and unwitting use of them will cause havoc. Besides that, the parsing for the signal variables `Si` in the user’s script is primitive in the current version of EIN, and no other names are allowed to begin with ‘‘`S`’’.

Appendix 3: Random Access to Input in EIN

Besides the signal and buffer variables just mentioned the following are all the reserved words in EIN:

```
float   y, left, right, sr, two_PI;
int     t, nsamps, inputskip;
short   *inputsig;
```

All the others besides `inputskip` and `*inputsig` were described at the beginning of this paper. These latter two variables provide random access to an input signal if one is being used.

The pointer `*inputsig` is the address of an array of short integers containing all the samples of an input soundfile. If the soundfile is mono, the sample numbers are equal to array locations. If the soundfile is stereo, the even indices represent the left channel and the odd indices the right channel. The sample number is then equal to `t/2` (`+1` for the right channel). If an input file has been opened the values of `y` are the current samples of the input signal, but the `inputsig[]` array can also be used to arbitrarily address parts of the input soundfile.

The integer `inputskip` is the sample number at which the input signal is taken to start. This will be a number other than 0 if it is specified in the ‘‘input skip:’’ form for an input file (see the screen snapshot in Fig. 3). Therefore, if a sound is accessed via the `*inputsig` array, its samples should be referred to by saying `inputsig[t + inputskip]`.

Appendix 4: A Remark about Generality

Any signal flowgraph G that represents a realizable digital filter can be represented in an EIN script using ordinary C code, plus `tap` statements.

The following informal argument should be convincing: Start with any signal flowgraph and remove the delay elements. What remains must be free of loops, because it represents the computation done during one sampling interval, and a loop would mean there would be no step-by-step procedure for finding the next output value.

The next claim is that the nodes in the flowgraph with the delays removed can be ordered from left-to-right, with no branches in the right-to-left direction. To do this, notice that there must be a node with no incoming branches. If there were not, there would be a loop, because we could start at a node and follow it backwards forever. Remove that node, putting it on the left end. Now repeating this process results in a left-to-right ordering that satisfies the claim.

In technical jargon, the flowgraph with the delay elements removed is a *directed acyclic graph (dag)*, and a dag can always be topologically sorted.

The left-to-right order of the nodes now determines an order in which an EIN script can evaluate the corresponding variables. The left-to-right arcs in the flowgraph without the delays are implemented with feedforward (left-to-right) arcs with delay zero, and the remaining arcs with feedforward arcs with delays $L \geq 1$ and feedback arcs with delays $L \geq 1$.