SCRIPTING AND TOOLS FOR ANALYSIS/RESYNTHESIS OF AUDIO

Stephen Travis Pope

CREATE, University of California, Santa Barbara

stp@create.ucsb.edu

DRAFT FOR REVIEW

ABSTRACT

Software tools for audio analysis, signal processing and synthesis come in many flavors; in general they fall into one of two categories: interactive tools with limited extensibility, or non-graphical scripting languages. It has been our attempt to combine the best features of these two worlds into one framework that supports both (a) the easy development of GUI-based applications for digital audio signal processing (DASP), and (b) an extensible text-based scripting language with built-in libraries for DASP applications. The goal is to combine the good performance of optimized low-level code for the signal processing number-crunching, with a powerful, flexible scripting language and GUI construction tools for application development. We investigate the solutions to this dilemma on the basis of four concrete examples in which DASP tools have been used together with the Siren music/sound package for Smalltalk.

1. INTRODUCTION

Analysis/resynthesis is one of the central topics in the field of digital audio signal processing (DASP), and the literature (e.g., as found in previous *ICMC Proceedings*) is full of useful algorithms and software tools for audio analysis, data processing (e.g., reduction, smoothing, mapping), and resynthesis. Examples include the many front-ends to FFT/IFFT-based sound processing (e.g., phase vocoders), tools for linear prediction of sound and LP-based cross-synthesis, and physical-model-based synthesis frameworks.

In most cases, one is presented either with a stand-alone graphical tool, or with a scripting language and no interactive tools. Examples of the first category are Michael Klingbeil's Sinusoidal Partial Editing Analysis and Resynthesis application (SPEAR, http://www.klingbeil.com/spear), the composer's Desktop Project's SoundLoom tool (http://www.composersdesktop.com), Douglas Scott's MiXViews (http://create.ucsb.edu/~doug/htmls/MiXViews.html) or the Princeton SndTools packages (http://sndtools.cs.princeton.edu).

Scripting language bindings for DASP also abound, such as the MATLAB signal processing toolbox (http://www.mathworks.com/products/signal), the Python-language front-end to Victor Lazzarini's SndObj (http://sndobj.sourceforge.net), aubio's scripting front-ends (http://aubio.piem.org), and the SWIG interface to Kelly Fitz' Loris package (http://sourceforge.net/projects/loris).

It poses a problem for many users, however, that these two sets of tools are mutually exclusive, that there appear to be very few tools that integrate graphical interactive data editors together with scripting languages for batch processing and easy extension (and even fewer where the graphical tools are themselves extensible).

There is a grey area in this otherwise-clean distinction that is populated by customizable scriptable applications such as Kyma (http://www.symbolic-sound.com) user-extensible editors like CommonLispMusic/SND (http://ccrma.stanford.edu/software/snd, http://ccrma.stanford.edu/software/clm) and spatial sound scripting/editors like SWONDER (http://sourceforge.net/projects/swonder), and by scripting languages with rudimentary GUI tools (e.g., Max and PD).

The user wants interactive, GUI-based tools for data exploration, input file preparation, and tuning of the analysis/synthesis processing, and then--for the later stages of realization--desire the simplicity and scalability of a scripting language for processing large numbers of source files.

The GUI tools are required to support flexible file I/O and data management, audio data display and (possibly) editing, analyzer configuration and execution with interactive monitoring of the results, and easy tool extension or customization.

As an extension language, it is convenient to have a simple, untyped language with a rapid turn-around compiler and an interpreter or shell-like interface. The scripting language front-end should support the same processing techniques and configuration parameters as the GUI-based tool, and should offer acceptable performance (often leaving the DSP up to C code). It should also provide comfortable file management facilities, so that scripts can be written quickly that iterate over large file sets running a given analysis process.

In this paper, we will discuss the design, implementation, and usage of an integrated framework for audio analysis, resynthesis, and cross-synthesis (morphing) that combines several DASP back-ends with a set of end-user GUI tools and a powerful and extensible scripting language for audio analysis/synthesis. The system is part of the Siren version 7.5 system (introduced elsewhere in these Proceedings), and uses libraries from the Loris (http://sourceforge.net/projects/loris), CSL (http://FASTLabInc.com/CSL), Aubio (http://aubio.piem.org), and Csound (http://csounds.com) packages as back-end DASP servers.

We will present a brief overview of the Siren system, then describe each of the models and back-ends separately, before evaluating the system as a whole.

2.   BACKGROUND

The Siren system is a Smalltalk framework for music/ sound processing (Pope 1987, 1993, 2002, 2007). It comprises about 375 classes in a group of object-oriented frameworks for signal description and processing, and includes external interface objects whose methods are actually proxies for C functions linked in from dynamically loaded libraries. This facility allows us to connect to external operating system resources such as sound and MIDI I/O, as well as to sound analysis/synthesis libraries such as are the topic of this paper.

Siren supports GUI construction and interactive tool development with the standard Smalltalk GUI library (a sophisticated extension of the model-view-controller framework), as well as with a set of custom components for DASP applications, such as reusable sound and score editor components.

SWIG (Simplified Wrapper and Interface Generator, http://www.swig.org) is an interface compiler that connects programs written in C and C++ with scripting languages such as Perl, Python, Ruby, and Tcl. It works by taking the declarations found in C/C++ header files and generating wrapper code that scripting languages need to access the underlying C/C++ code. There is a Smalltalk back-end to SWIG by Ian Upright (http://commonsmalltalk.wikispaces.com/SWIG+Documentation); with this, one has external interface objects whose methods call the functions created by SWIG, which themselves mirror the object methods of the source package that was fed into SWIG. This mechanism allows us to use SWIG APIs to access external libraries from within Siren, as we will describe below.

In the following sections, we will introduce four systems that combine scripting and batch processing with extensible GUIs for DASP-based tools within the Siren framework.

3.   USING LORIS IN SIREN

Loris is a sound analysis/resynthesis package written by Kelly Fitz that uses the model of time-adjusted bandwidth-enhanced partial lists to enable flexible resynthesis control and sound morphing. The central classes in Loris are the *Analyzer*, which reads a sound file and runs the multi-stage analysis process, and the *PartialList*, which represents the sound analysis results as a list of time-adjusted bandwidth-enhanced partials.

The analyzer has many configuration settings, and one central *analyze()* method, which runs the processing and data reduction and returns a partial list data structure. PartialList objects are composed of Partials (akin to tracks), which themselves consist of a list of Breakpoints (each with time, frequency, amplitude, phase, and bandwidth members). PartialLists support several kinds of iteration to process their components, and also support file I/O (in SDIF format) and resynthesis, saving a reconstituted sound file. There are a collection of Loris utility functions that operate on PartialLists to offer operations such as partial list distillation, collation, and time dilation.

Loris is a complex C++ framework that includes several stand-alone applications as well as a SWIG wrapper that allows users to script Loris processes in the Python language. Siren's SWIG-generated interface and glue code provide access to Loris analyzers and partial lists as Smalltalk objects. These low-level interfaces are then used to implement abstract model classes in Siren. The model class *LorisSound* (a subclass of *SampledSound*, see the class tree in Figure 1) has a sample array (the original sound), an analyzer configuration, and a partial list (the time-adjusted bandwidth-enhanced partial spectrum).
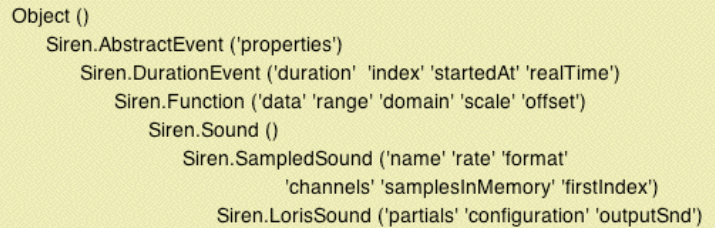


```
Object ()
    Siren.AbstractEvent ('properties')
        Siren.DurationEvent ('duration' 'index' 'startedAt' 'realTime')
        Siren.Function ('data' 'range' 'domain' 'scale' 'offset')
            Siren.Sound ()
                Siren.SampledSound ('name' 'rate' 'format'
                                    'channels' 'samplesInMemory' 'firstIndex')
                    Siren.LorisSound ('partials' 'configuration' 'outputSnd')
```

Figure 1. Hierarchy of Sounds and LorisSounds

In addition to simple accessor methods (getters and setters for the data members), class LorisSound provides the partial list iterators mentioned above, special sound-like methods (e.g., amplitude normalization), and the main analysis and resynthesis methods.

As an example, look at the following condensed Smalltalk code for a simple Loris processing task; the following method from class LorisSound will load a sound file and analyze it, saving the result as SDIF and playing the before and after sound files if the appropriate flags are set in the analyzer configuration.

```
LorisSound.analyze
    "Run the Loris analyzer using the receiver's configuration,
    then collate and distill; save SDIF and/or AIFF files and play
    the output, depending on the configuration"
"declare temp vars"  "declarations are optional in workspaces"
    | sndfile samples analyzer fenv dsize basenam durFloat |

"set up the analyzer"
    analyzer := Siren.Loris.Analyzer
            resolutionHz: configuration resolution.
    configuration amplEnv ifNotNil:
            [analyzer buildAmpEnv: configuration amplEnv].

"read the snd file"
    sndfile := Siren.Loris.AiffFile filename: basenam.
    samples := sndfile samples.
    durFloat := duration asSec value.
    dsize := (durFloat / analyzer hopTime) truncated.

"run the analyzer"
    partials := analyzer analyze: samples srate: rate.
"plug the results into the receiver"
    self loadEnvelopes: analyzer size: dsize.

"create freq ref"
    fenv := Siren.Loris.Loris createFreqReference: partials
            minFreq: configuration minFreq
            maxFreq: configuration maxFreq numSamps: dsize.
    self fRefEnv: (LinearFunction fromLorisData: fenv
            size: dsize duration: durFloat).
```

```
"channelize & process the data"
    Siren.Loris.Loris channelize: partials
        refFreqEnvelope: fenv refLabel: 1.
    self freqRefEnv: (LinearFunction fromLorisData: fenv
        size: dsize duration: durFloat).


"now process: sift, collate, distill"
    Siren.Loris.Loris sift: partials.
    Siren.Loris.Loris distill: partials.
    Siren.Loris.Loris collate: partials.


"save/play output files"
    configuration saveSDIF ifTrue: ["save"].
    configuration saveOutput ifTrue: ["save"].
    configuration playResults ifTrue: ["play"].
```

To actually run this example within Siren, one has to initialize the glue code object by sending an initialization message to the Loris object in the namespace Siren.Loris; this looks like,

```
    Siren.Loris.Loris initializeModule.
```

To create a LorisSound, plug in some analysis parameters, and run the process, a class factory method is provide that allows one to say,

```
    LorisSound fromFile: '1.2a1.aiff'
        configuration: LorisAnalysisConfiguration default.
```

The LorisSound class has a number of other methods (implemented using the low-level interface methods) for operations on partial lists, such as dynamic range compression, masking, frequency shifting, and morphing cross-synthesis. Multiple versions of the same LorisSound objects can be saved and restored to/from SDIF files or Siren *s7*-format files, based on settings in the analyzer configuration.

In the method above, we also see that several Siren classes have special methods for interfacing with Loris objects, the best example being the message,

```
    LinearFunction fromLorisData: fenv
        size: dsize duration: durFloat
```

which creates a Siren *LinearFunction* object given a Loris linear breakpoint envelope.

Given these facilities, the reader can hopefully imagine using Smalltalk as a scripting language for cross-synthesis with Loris. The Smalltalk class libraries include excellent support for file and directory manipulation, so that scripts are easily written that go through entire directories of source sounds analyzing each one with a collection of analyzer settings and saving the results in a database or in the user's folders as SDIF or AIFF files.

The next step would now be to provide some manner of interactive GUI displays for Loris objects, and to write controllers that provide access to the methods described above. The simplest tool is the analyzer configuration panel shown in the next Figure; it illustrates typical Loris analyzer settings (when set to a base resolution of 70 Hz).

The Smalltalk GUI construction tools make it a simple matter to build such dialog boxes, and to link their methods to a model class' setter methods. To build more

sophisticated GUIs for Loris, we need to use Siren's audio-specific widget set (see http://FASTLabInc.com/ Siren/Doc/Siren.GUI_2007.html for a tour).

The basic LorisEditor GUI (shown in Figure 3) has four main panes: two for sound views, one multi-function view, and a spectrum view. Depending on the application, the sound views might represent an original and a resynthesized sound (for tuning analysis parameters), or two different sounds (for morphing and cross-synthesis). The menu bar at the top of the editor view contains items for all the important Loris functions: analyzer configuration and analysis/resynthesis, separate operations on time, frequency, and amplitude components of the subject partial list, buttons to play the before or after sounds, and a button to bring up the view editor for rearranging the view's panes.
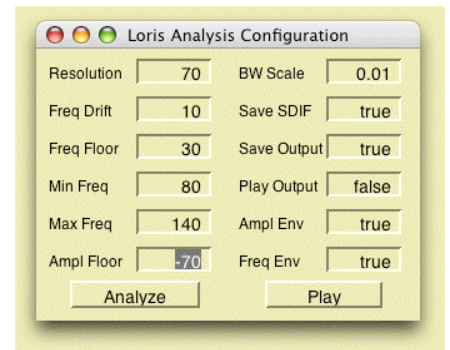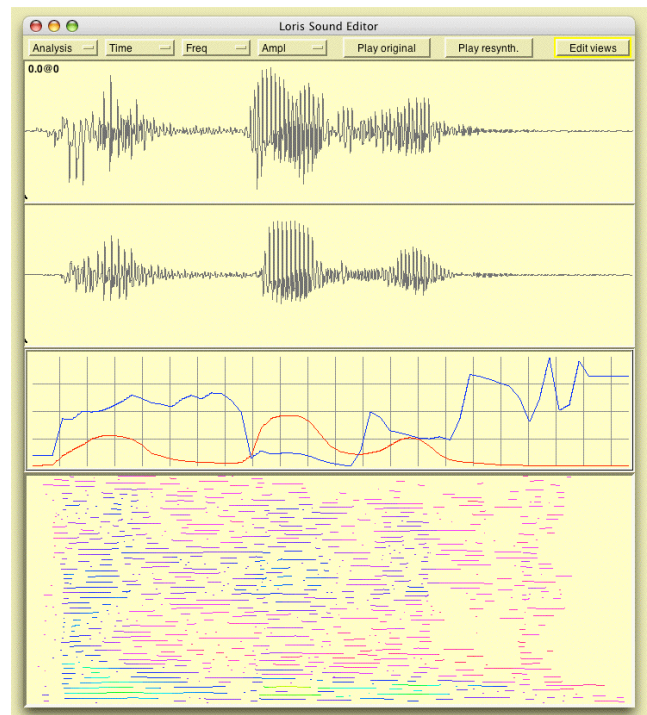

Figure 2. Loris Analyzer Configuration Panel

Figure 3. Loris Editor with before/after sound views, extracted feature functions, and partial spectrum editor, operations are in the menu bar

Because of the flexibility of the Smalltalk environment, one can add new menu buttons or menu items or change the display methods at run-time (i.e., add a new feature or change display colors while the app's running). This is especially useful for building new tools (the debugging phase), and for developing batch processing scripts based on methods in a GUI controller class. The Smalltalk compiler is also part of the runtime environment, so that user methods can

always create new classes or methods on-the-fly. There is much more to say about the Loris package in Siren (see http://fastlabinc.com/Siren/Workbook), but this brief introduction will suffice for the purposes of the current paper.

4.   USING CSL VIA SWIG

The CREATE Signal Library (CSL, pron "sizzle") is a framework for building digital audio synthesis and processing applications in C++. CSL has been used to write synthesis servers that respond to MIDI or OpenSoundControl inputs, and for sound analysis and feature extraction applications.

As detailed in our previous ICMC papers, CSL provides an object model based on traditional unit generator objects that are connected via their I/O and control ports into DASP graphs. A CSL graph typically has an IO object as its root, in which case the IO object periodically sends the message *nextBuffer()* up the graph to trigger CSL signal processing. The CSL class library includes a feature-rich set of unit generator classes for all common audio analysis, synthesis, processing , and multi-channel spatialization methods. The library relies heavily on C++ multiple inheritance, with lightweight "mix-in" classes for composing behaviors such as writeability, streamability, effects with inputs, and cacheing.

One typically uses CSL by writing a C++ application that uses the CSL class library and runs as a stand-aolne server or as a plug-in for some audio host API (e.g., VST or CoreAudio).

As with Loris, Siren uses a SWIG-generated interface and C-language glue code to create and trigger CSL object networks (DASP flow graphs) from within Smalltalk. Because the CSL model is based on the simple procedural unit generator model (like MusicN-style languages), Siren methods that create CSL graphs look very familiar to most users. As an example, the method below will construct a simple graph with a sine oscillator controlled by an ADSR envelope. The output is sent to a PortAudio IO object, which is then turned on to play for several seconds.

```
CSLGraph.sineWithADSR
    "Create and run a simple CSL DSP graph consisting of a
    Sine wave with an ADSR envelope."
"declare temp variables (optional)"
    | env sin out |
"Instrument: create a simple sine-with-envelope graph"
    env := ADSR dur: 2.0 att: 0.05 dec: 0.05 sus: 0.5 rel: 1.0.
    sin := Sine freq: 110 ampl: env.
"create an IO object and plug in the sine"
    out := PAIO sRate: 44100 bSize: 1024 root: sin.

"Score: open the output, trigger the envelope, and start"
    out open.
    out start.
    env trigger.
"sleep a bit (3 sec)"
    3 seconds wait.
"shut down nicely"
    out clearRoot.
    out stop
```

This example could be translated line-for-line into most other synthesis languages as an instrument definition and a score or play command. As a second example, the method below uses a Smalltalk loop control structure to add 50 inputs to a mixer; each input is a stereo panning sine oscillator with random-walk envelopes for the frequency and position.

```
CSLGraph.randomOscillatorBank
    "Create and run a CSL DSP graph with 50 sine waves with
    random-walk frequencies and stereo positions."
"declare temp variables"
    | out mix |
    out := PAIO sRate: 44100 bSize: 1024.   "IO guy"
    mix := Mixer chans: 2.                "stereo mixer"
    50 timesRepeat:                       "loop 50 times"
        [ | osc pos pan freq |            "loop temps"
        freq := RandEnvelope frequency: 1.0 amplitude: 200.0
            offset: 300.0 step: 50.0.     "create a rand env"
        osc := Sine freq: freq ampl: 0.005.  "sine osc"
        pos := RandEnvelope new.          "position envelope"
        pan := Panner in: osc pos: pos.   "panner"
        mix addInput: pan].               "send to mixer"
    out setRoot: mix.                     "set output"
    out open; start.                      "open and start output"
    30 seconds wait.                      "play for 30 sec"
    out clearRoot; stop                   "stop output"
```

As a final example, the following method creates a similar oscillator bank, but this time there are 16 oscillators and they are controlled by a MIDI fader/knob box (assumed to send out continuous controller messages for controllers in the range 48-63). It loops until its scheduler thread is stopped

```
CSLGraph.oscillatorBankWithMIDI
    "Create and run a CSL DSP graph with 16 sine waves under
    MIDI fader control."
"create output"
    mix := Mixer chans: 2.
    out := PAIO s_rate: 44100 b_size: 1024 root: mix.

"set up osc bank and array"
    oscBank := Array new: 16.             "list of sines"
    1 to: 16 do:                          "loop 16 times"
        [ :index | | osc pos pan |        "ma e a sine"
        osc := Sine frequency: 110.0 ampl: 0.05.
        pos := RandEnvelope new.          "randmo panner"
        pan := Panner input: osc position: pos.
        mix addInput: pan.                "added to the mixer"
        oscBank at: index put: osc].      "and to the osc array"
"play"
    out open; start.

"Now set up the MIDI faders to control the oscillators"
    midi := MIDIPort new.
    midi openInput: #PC1600X.             "controller name or #"
    midi startControllerCaching.
    midi startMIDIInput.
    ctrlrData := Array new: 16.           "current values"
    freqData := Array new: 16 withAll: 0.  "cached values"
    ctrlrData gcCopyToHeap.   "so that you can pass the ptr to C"
```

```
[EventScheduler isRunning] whileTrue:      "control loop"
                           "read controllers from MIDI driver"
    [midi readControllersFrom: 48 to: 63 into: ctrlrData.
    1 to: 16 do: [ :index |   "if controller has changed, update"
       (ctrlrData at: index) = (freqData at: index) ifFalse:
             [freqData at: index put: (ctrlrData at: index).
             (oscBank at: index) setFrequency:
                   (110.0 + ((ctrlrData at: index) * 4))]].
    (Delay forMilliseconds: 50) wait].    "loop delay"
"when done"
   out clearRoot; stop
```

The rapid-turn-around nature of the Smalltalk environment makes debugging DASP graphs like these (and finding appropriate ranges for the control variables) *much* easier than it would be in C++ or Java, and the Smalltalk code database tools support version management, roll-back, releases, incremental diffs, patches, etc. in an integral manner.
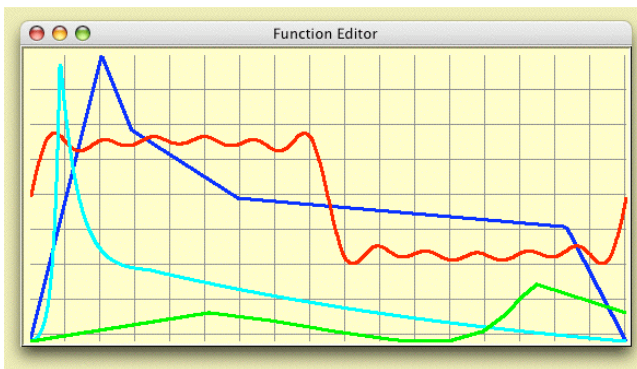

Figure 4. Siren Function Editor with ADSR, sum-of-sines, and spline functions on display

For CSL, we use the standard Siren sound and function GUI editors to prepare input data and display and edit sound results. The basic multi-function editor is illustrated in Figure 4 above, which shows linear and exponential break-point envelopes, a sum-of-sines function, and a spline curve. Figure 5 below shows the simple Siren sound editor and its pop-up menu.

Once configured, CSL graphs created from Siren can respond to MIDI or OSC messages, so that one can play a score using the standard Siren voices and have the commands processed by a CSL graph that shares Siren's address space!
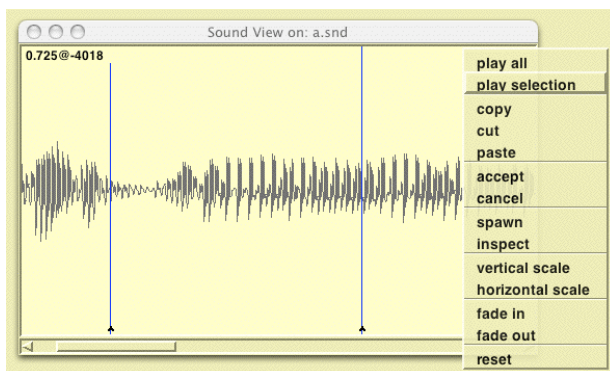

Figure 5. Siren Sound Editor with a selection and pop-up menu for sounds

5.   LPC

The two interfaces we will discuss next each use external programs and parse their output files in order to provide editing or scripting functionality using Siren's GUI tools and/or the Smalltalk language.

In the case of the LPC tools, we needed a simple editor for fixing the pitch estimates delivered by the Csound linear-prediction analyzer *lpanal*. It proved to be quite simple to parse the binary format used by Csound to store LPC analysis data; each frame includes reflection coefficients, residual data, error factors, and pitch estimates. The file-loading code reads the analysis file and creates a Siren *LPCSound* object (with a collection of *LPCFrame* members), which has a Siren *Function* object for its pitch estimate data. Given this, it was trivial to customize Siren's function editor (introduced in Figure 4 above) to add a few new operations that perform various kinds of data smoothing on pitch tracks.
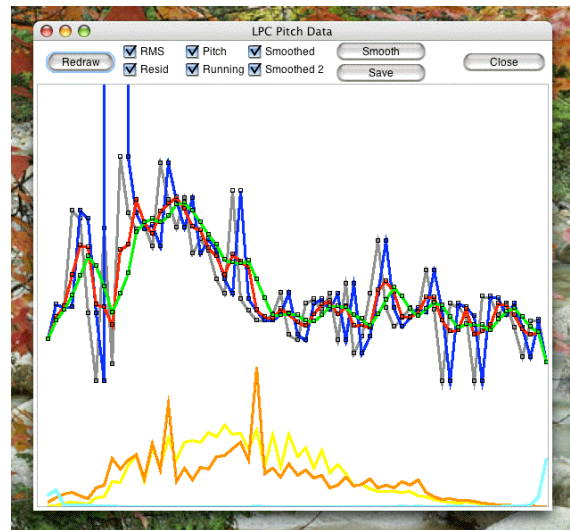

Figure 6. LPC Pitch Editor with several smoothing algorithms; the buttons select among reduced versions of the input data

The tool shown in Figure 6 has buttons to toggle the various function displays on and off, and to iteratively smooth the displayed data. The resulting massaged data can be rewritten to a formatted file for further processing with Csound or any other tool that supports their LPC data file format.

6.   AUBIO

As the final example, we have integrated several tools from the aubio package (http://aubio.piem.org) for audio feature extraction, labeling, and segmentation into Siren. This was done by writing methods (mostly in the class *SampledSound*) that run aubio programs and parse the results. Smalltalk supports executing UNIX shell commands and taking over their I/O streams, so it is easy, e.g., to take a musical sound and locate all the note onsets using the *aubioonset* program, setting the chosen note onset points as markers in the sound object (two sound markers are shown in the editor in Figure 5 above). The *aubionotes* utility reads a sound file and

generates a list of MIDI commands that correspond (if you're lucky) to the sound's pitch/time contour.

7. EVALUATION

The exposition above presented four very different methods of integration of scripting languages and graphical editors for existing DASP APIs. Each of these has different strengths and weaknesses, and each reflects both the back-end package it uses and the assumed application for which it was designed.

The first two examples (Loris and CSL) both provide flexible interactive scripting languages for sophisticated and complex C++ frameworks by mirroring the class hierarchy and core methods in Smalltalk using the SWIG interface generator. This indeed gives the user the best of both worlds: a rapid-turn-around scripting development tool with the performance of optimized C code for the numerical "heavy lifting." This does, however, come at some expense, the SWIG interfaces can be complex (or time-consuming) to develop, and the interface and glue code must be regenerated whenever the underlying source is changed. (Luckily, both Loris and CSL are mature and stable.) There are some problems with SWIG's support for multiple inheritance in its Smalltalk back-end (which is derived from the Java back-end). The current version simply ignores secondary superclasses in C++ classes that have them, meaning that special setter methods were needed in the Siren mirror classes for CSL.

For the case of Loris, advanced interactive tools have been built to support sound analysis, partial list management, and cross-synthesis control. Because of the Smoke models of sounds and spectra, and the audio widgets in Siren, this package amounts to 2000 lines of code (LOC), about 40% of it auto-generated GUIs.

As of this writing, CSL does not have any specially developed GUI tools, though a flow-chart "patch editor" would be easy, since there are quite powerful structured graphics editor packages for Smalltalk (e.g., HotDraw, http://st-www.cs.uiuc.edu/users/brant/HotDraw/Hot-Draw-applications. html).

In both of these packages, there were initial problems with memory integrity, which lead to a dangling pointer in the Loris code causing the Smalltalk virtual machine to crash (very upsetting to a Smalltalk programmer). These cases have *mostly* been addressed by extensive exception handling (exception raising, actually, with a rich exception hierarchy for external interfaces and shared data pointer management) on the Smalltalk side of the glue code.

Given the fact that both Loris and CSL already supported SWIG, it is obvious that they are intended to be used with scripting languages such as Python or Tcl. Smalltalk is differentiated from these by its minimalist yet scalable syntax, long-term stability, comprehensive yet simple class libraries (with namespaces and package support, and thousands of available libraries), and comprehensive integrated development environment (and GUI builder); all of these features are missing from most scripting languages. In contrast, however, to the scripting language *du jour*, Smalltalk has lamentably small market/mind share.

The latter two examples (Csound LPC and aubio's analysis/segmentation) demonstrate techniques for building on top of existing batch programs by preparing and processing their I/O files. These cases are much simpler than the former two; neither involves the rich data models, but rather are incorporated into methods of existing classes, as in the example of the aubio-enabled message aSound getNoteOnsets or simple model refinements as in LPC. The LPCSound/Frame/View/ Editor classes each have only a few methods, and the whole set comes to less than 800 LOC.

In these cases, we have used the Siren class hierarchy to distribute the new functionality according to how it fits into the Siren object model; the new features integrate well with other Siren methods and tools (e.g., take a Siren sound object and derive a Siren score from it). This technique might be appropriate for scripting languages for which sound/music libraries are available.

Siren supports both score description and sound processing, so that DASP processing scripts like those presented above can be integrated into scores (Siren event lists), and that special real-time driver objects (Siren voices) can be built for objects whose methods use external interface calls and shared data.

Comparing these tools to their nearest cousins, Kyma and SND, we find a number of similarities, and also several significant differences. Kyma's model certainly incorporates score and sound object creation and performance well, and its extensive set of graphical tools exhibit a remarkable collection of features. The extension language for new DASP methods is partially Smalltalk and partially DSP code. Kyma assumes a custom DASP server farm (the Capybara), however. The CCRMA clm/SND environment uses LISP as both the implementation and extension language, and has GUI editors for both sound and score data. While the tool set is sophisticated and cross-platform, it contains only a few tools and is rarely (if ever) extended.

8. CONCLUSIONS

With few exceptions, software tools for audio analysis, signal processing and synthesis fall into the categories of closed interactive tools or non-graphical scripting languages. It has been our attempt to combine the best features of these worlds into one framework that provides both easy development of GUI applications for digital audio signal processing (DASP), and an extensible text-based programming (scripting) language with built-in libraries for DASP applications.

We described four recent developments that integrate different DASP systems with the Siren music/sound framework in the Smalltalk programming environment. In two of the cases (Loris and CSL), we built mirror class hierarchies, with Smalltalk classes and methods derived from the back-end package's C++ design. These interfaces allowed us to use Smalltalk as a scripting language for the high-performance back-end functions, and to build or reuse GUI tools to operate on the DASP data. In the final two cases, we extended existing Smalltalk classes with methods that communicate with external processes either by reading formatted binary output files or by forking out-board shell commands to run batch processes and parse their output. In each of

these cases, new object models for scripting and custom GUI applications could be produced with alarmingly little code and quite a low cost of development using the best tools available.

We believe there is much more to be done in these areas, and that many of the techniques described here would translate quite well to other languages, *provided* they had Siren's rich object model and representations for music/sound objects and Smalltalk's cross-platform integrated development environment, look-and-feel agnostic GUI builder, and music/sound widget set.

9. REFERENCES AND LINKS

Pope, S. T. 1993."The Interim DynaPiano: An Integrated Tool and Instrument for Composers." *Computer Music Journal* 16:3.

Pope, S. T. 2002. "Music and Sound Processing in Squeak Using Siren." Invited Chapter in M. Guzdial and K.m Rose, eds. *Squeak: Open Personal Computing and Multimedia*. Prentice-Hall.

Pope, S. T. 2007. "The Siren 7.5 Music and Sound Package in Smalltalk" submitted to *Proceedings of the 2007 International Computer Music Conference.* also at http://FASTLabInc.com/Siren

Aubio. http://aubio.piem.org

CSL: CREATE Signal Library
 http://FASTLabInc.com/CSL

CRAM: CREATE Real-time Applications Manager
 http://create.ucsb.edu/CRAM

FFTW. http://www.fftw.org

libSndFile: http://www.mega-nerd.com/libsndfile

Loris: http://sourceforge.net/projects/loris

OSC: OpenSoundControl
 http://www.cnmat.berkeley.edu/OpenSound-Control

PortAudio: http://www.portaudio.com

PortMidi: http://www.cs.cmu.edu/~music/portmusic

SHARC. http://www.timbre.ws/sharc

SuperCollider. http://supercollider.sourceforge.net/

SWIG. http://www.swig.org