Making Sounds with Numbers: A tutorial on music software dedicated to digital audio

Nicola Bernardini (1), Davide Rocchesso (2)
(1) Conservatorio "C. Pollini" di Padova, Italy
nicb@axnet.it
(2) Dipartimento Scientifico e Tecnologico, Università di Verona, Italy
rocchesso@sci.univr.it

Abstract

A (partial) taxonomy of software applications devoted to sounds is presented. For each category of software applications, an abstract model is proposed and actual implementations are evaluated with respect to this model.

1 Introduction

In recent years, the increased power and affordability of personal workstations has fostered a vast variety of software applications devoted to sounds and their musical use.

The aim of this tutorial is to present currently available applications in the field stressing the taxonomical aspect of different approaches and functions. The following categories will be introduced and developed:

- Languages for sound processing
- Inline sound processing
- Software to teach signal processing
- Processing libraries, plugins and toolkits

The tutorial will show abstract models for each category describing difficulties and problems encountered in actual implementations, while attempting to illustrate and evaluate user interaction and control in diverse situations. Examples of actual programs will be shown, stressing the functionalities they intended to fulfill by design and their usage in real-world situations.

Finally, some issues that still lie partially unresolved such as fast network interoperability, parallelization and musical control will be mentioned; future directions of development will be proposed.

2 Languages for sound processing

2.1 Abstract models

The most successful model for languages dedicated to sound processing dates back to the late fifties, when Max Mathews developed a collection of programs (called *Music I–II–…–N*) at the Bell Laboratories[1]. One of these programs is the *Music V* sound-synthesis language, which established a standard based on the concept of *unit generator* (UG). The UGs are primitive modules for generating, modifying, and acquiring audio or control signals: UGs that can perform cyclic or acyclic readings of sample tables are essential to produce audio signals whereas UGs for envelopes and low-frequency oscillators are available to produce control signals. To modify audio signals in the time or frequency domain, it is useful to have UGs which implement various forms of digital filters, such as delay lines or resonators.

According to the Music-N tradition, the UGs are connected as if they were modules of an analog synthesizer, and the resulting patch is called an *instrument*. The actual connecting wires are variables whose names are passed as arguments to the UGs. An *orchestra* is a collection of instruments. For every instrument, there are control *parameters* which can be used to determine the behavior of the instrument. These parameters are accessible to the interpreter of a *score*, which is a collection of time-stamped invocations of instrument events (called *notes*). Fig. 1 shows a schematic description of how Music-N languages work: `a)` is a Music-V source text[2] while `b)` is its graphical representation.

The orchestra/score metaphor, the decomposition of an orchestra into non-interacting instruments, and the description of a score as a sequence of notes, are all design decisions which were taken in respect of a traditional view of music. However, many musical and synthesis processes do not fit well in such a metaphorical frame. As an example, consider how difficult it is to express modulation processing effects

[1] For a good historical survey of sound and music languages we recommend the textbook by C. Roads [14].

[2] picked up from [10, page 45]

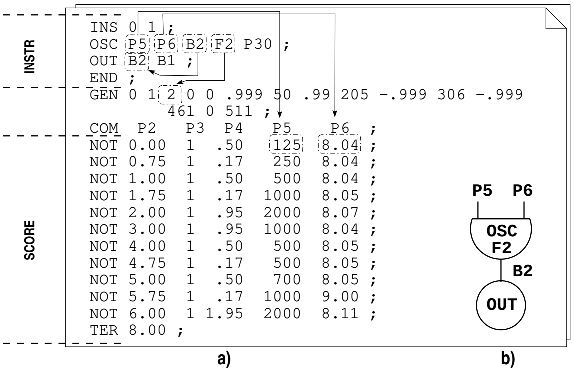

Figure 1: Music-V file description

that involve several notes played by a single synthesis instrument (such as those played within a single violin bowing): it would be desirable to have the possibility of modifying the intrument state as a result of a chain of weakly synchronized events (that is, to perform some sort of *per-thread* processing).

Currently, the most widely used language for sound synthesis is *Csound*[3], which is strictly adherent to the original dictates of Music V while improving it on the symbolic aspect and versatility. There are several graphic helpers running on different platforms to assist musicians in writing Csound orchestras and scores: the most widely used are *Cecilia*[4] and *WcShell*[5].

Another popular Music-N-like language is *Cmusic*[6] [11, 12], which we will mention again in section 5 as an example of integration of a sound-processing language into a software environment making extensive use of the facilities provided by unix operating systems.

Other models have been proposed for dealing with less rigid descriptions of sound and music events. One such model is tied to the language *Nyquist* [7] [4]. This language provides a unified treatment of music and sound events and is based on functional programming (Lisp language). Algorithmic manipulations of symbols, processing of signals, and structured temporal modifications are all possible without leaving a consistent framework. In particular, Nyquist exploits the idea of behavioral abstraction, i.e. time-domain transformations are interpreted in an abstract sense and the details are encapsulated in descriptions of behaviors [2]. In other words, musical concepts such as duration, onset time, loudness, time stretching, are specified differently in different UGs. Modern compositional paradigms benefit from this unification of control signals, audio signals, behavioral abstractions and continuous transformations.

Placing some of the most widely used languages for sound manipulation along an axis representing flexibility and expressiveness, the lower end is probably occupied by Csound while the upper one is probably occupied by Nyquist. Another notable language which lies somewhere in between is *Common Lisp Music* [8] (CLM), which was developed as an extension of Common Lisp [15]. If CLM is not too far from Nyquist (thanks to the underlying Lisp language) there is another language closer to the other edge of the axis, which represents a "modernization" of Csound. The language is called *SAOL* [9] and it is being adopted as the formal specification of structured audio for the MPEG-4 standard [17].

2.2 Design Issues

2.2.1 Samples Vs. Blocks

Since the introduction of early computer music software, there has been some debate about the way of computing samples by means of signal processing algorithms. Control signals vary with a rate lower than the audio sample rate and it makes sense to collect blocks of samples produced by tight loops and pass these blocks along the signal processing flowgraph. In general purpose architectures, this strategy introduces significant savings (up to a factor of 7 in Csound [5]) connected to the better use of registers[10]. Another source of savings comes from the fact that control signals are updated at control rate, typically at block boundaries. This coarse time-discretization often produces audible artifacts which can be reduced either by reducing the block size (thus loosing some of the benefits in performance) or by introducing interpolation in control signals. It appears that the most efficient strategy is to incorporate control-signal interpolation within UGs [5]. We think that this kind of signal degradation due to blocking is generally acceptable, especially if one uses large blocks for preparatory sketches and small or no blocking for the final rendering. However, there is a much more serious category of artifacts due to blocking that cause serious headaches to many sound designers [12, page 37]. These appear whenever there is feedback in a signal processing patch. For instance, explicit computation of a recursive filter often requires feeding back signals which are delayed by one sample only. This can not be done unless the block size is set to one. If the block size takes different values, the resulting spectra are wildly affected by imaging and aliasing. This second species of artifacts are particularly bad because they are all but the kind of gentle degradation that can be tolerated during preparatory sketches. Since these artifacts have a semantic nature and it is not easy to extend a language to deal with them, we prefer the solution taken by CLM and

[3]ftp://ftp.maths.bath.ac.uk/pub/dream/
[4]http://www.musique.umontreal.ca/electro/CEC/
[5]http://www.axnet.it/edison/wcshell.zip
[6]http://www-crca.ucsd.edu/cmusic/cmusic.html
[7]http://www.cs.cmu.edu/~rbd/nyquist.html

[8]http://www-ccrma.stanford.edu/CCRMA/Software/clm/clm.html
[9]http://sound.media.mit.edu/~eds/mpeg4
[10]Dannenberg and Thompson have shown that blocking is almost insensitive to the memory hierarchy due to prefetching of long cache lines [5]

SAOL where all audio signals are computed sample-by-sample. In particular, SAOL keeps the distinction between audio rate and control rate, using it only to have sparse updating of control signals and not for dividing audio-rate computations into blocks. This doesn't exclude the possibility of implementing block-oriented UGs (such as FFTs): they simply put a sample in a buffer every time they are called and compute the operation when the buffer is full.

2.2.2 Performance

Nowadays, general purpose computer architectures are fast enough to accomodate real-time performance of medium-size sound processing algorithms. In the recent past, it was thought that computation-intensive chores could take advantage of boards containing Digital Signal Processors. This was the approach taken by CLM, which could benefit of UGs coded in the assembly language of the DSP Motorola 56000, present in NeXT computers. Currently, the speed and processing power of CPUs has increased so much that floating-point C-language CPU implementations are often faster than fixed-point assembly-language implementations on DSPs. And of course it is much easier to implement a UG in C using floating-point arithmetic.

Benchmarks on implementations of various sound-processing languages show that there are no dramatic differences in performance [12, 3] when the same blocking policy is chosen. In particular, it is interesting to note that the expressiveness of Nyquist as compared to Csound is obtained with no efficiency losses. Just to mention a few of the advantages shown by Nyquist, sounds can be treated as any other argument in function calls and they can be temporally transformed on the fly by means of behavioral abstractions. Moreover, blocking in Nyquist is not subject to the restrictions of Csound, where note quantization matches the control rate [3]. However, Nyquist is designed to perform well when used with large block sizes, thus incurring in the semantic pitfall mentioned above where patches with feedback are designed. Problems are also likely to arise when using Nyquist in real-time sound processing, where tolerable I/O latencies impose small blocks, and interactive control (e.g. via MIDI) doesn't seem to agree too well with the internal representation of sounds as atomic entities [4].

So far, the designers of sound languages have not cared much of the characteristics of modern architectures when performing their optimizations. Again, a notable exception is found in [5]. Current computers have an organization of memory hierarchy such that: (i) reads are more expensive than writes, (ii) space locality in a memory reference pattern improves performance, (iii) access to tables whose addresses fit all in the Translation Look-aside Buffer is faster. These features can be exploited when designing the UGs. Other improvements, such as good use of the pipeline, loop unrolling, replacement of expensive operations, are typically performed by the compiler when it analyzes the code of UGs. The languages where instruments are actually compiled rather than interpreted can benefit from compiler-based global optimizations. In fact, the chances for optimization and extraction of parallelism increase with the increased mean length of basic blocks. For instance, CLM tries to translate instruments into C code which can then be compiled.

In the future, architectures having multiple CPUs will become widely available[11]. Therefore, it is important that sound languages can take advantage of multiprocessing by assigning loosely connected threads of computation (e.g. different notes) to different CPUs. This can be done fairly easily by using compilers that support parallel blocks, parallel loops, and shared variables. We can expect most future sound languages to be able to distribute computations automatically among the available CPUs.

2.2.3 Extensibility

A very important point driving the choice of a sound language is its extensibility. This feature is twofold: on the one hand it can be seen as the possibility of using syntactic structures of a high-level language within an instrument definition; on the other hand it can be seen as the possibility of enriching the set of UGs. The former feature is provided, among the languages mentioned so far, only by CLM and Nyquist, since any Lisp statement can be used within instruments. The latter feature is somewhat provided by all the languages, with varying degrees of flexibility. For example, extending the set of UGs of Csound is a matter of adding some C code and recompiling the sources. Unfortunately, this task can be daunting due to the intricacy of the source code. We also regret the fact that, since this is the only way of extending Csound, a plethora of extensions blossomed during the last decade, with the double effect of leading to unmanageable code and creating many incompatible versions. In this respect, an orthogonal approach is taken by the ISO Committee which is going to adopt SAOL as the standard language for specifying structured audio in MPEG-4: extensibility is limited by the fact that the set of UGs is standardized[12]. The hope of the proponents is that a sound-processing language "carved in stone" will be widely adopted by most multimedia-device manufacturers, as it happened with MIDI in the eighties.

In CLM, user-defined generators can be written in C language and interfaced by means of Lisp macros. Moreover, since general programming structures can be used within instruments and these can be translated into fast C code, the need of user-provided additions to the core language is alleviated.

[11] E.g. the forthcoming Intel-Merced architecture should support inexpensive and extensive multiprocessing

[12] However, new UGs can be defined using the standard SAOL syntax and a macro mechanism

In Nyquist, integrating user-defined generators implies using a tool which translates specifications of behaviors into actual C code, which is then compiled and linked into Nyquist. Without this tool, the complexity of behavioral abstractions would make it very difficult for a user to write a UG from scratch and to integrate it into Nyquist.

2.2.4 Dealing with Audio Effects

It is interesting to see how different sound languages deal with digital audio effects, i.e. instruments designed for modifying rather than generating sounds.

In Csound, effects are like any other instrument: they are invoked as "notes" from the score, and they receive input sounds through the use of global variables.

In CLM, any effect is assimilated to a reverberator. The macro `with-sound`, which is responsible for producing a note, operates a clear distinction between generation and processing. For example, the following statement instantiates a note from the instrument `sweep` and sends the result to the processing unit `echo`:

```
(with-sound
      (:reverb echo
             :reverb-data(1.0 0.1))
      (sweep "march.wav" :duration 25.0
             :freq-env '(0 0.0 100 1.0)))
```

Incidentally, note that the instrument `sweep` accepts as parameters a string representing a filename, a floating point number, and a list of points representing an envelope.

The approach to effect processing taken by Nyquist is the most elegant. Since scores and instruments are specified using the same high-level language, one can write functions that perform scores and use the result (which is a variable of type SOUND) as argument to functions that perform effects. For instance, with the properly defined functions `flanger` and `arpeggio`, the following statement would be valid in Nyquist:

```
(flanger  (arpeggio c2 e2 g2 c3))
```

If elegance and generality has been achieved in Nyquist by devising a rather complicated internal representation of sounds and behaviors, a neat representation of audio-processing connections can also be encapsulated within a Music-N framework. This is best shown by SAOL, which makes use of the metaphor of the mixing console with its "send" and "return" audio busses. The descriptions of complex audio patches turn out to be very terse and highly communicative, at least to someone previously exposed to some live audio-engineering practice (see sec. 6.2). As an example, consider the following code declaring a connection between a `generator` and an `echo`, the latter having two parameters, `delay` and `amplitude`:

```
route (bus1, generator);
//               delay   amplitude
send (echo;        0.1,         1.0; bus1);
```

It is worth noticing how a suite of programs based on piped communication, such as those described in sec. 5, leads naturally to an elegant way of expressing chains of sound effects.

3 Inline sound processing

A completely different category of music software deals with inline sound processing. The software included in this category implies direct user control over sound on several levels, from its inner microscopic details up to its full external form.

In its various forms, it allows the user to: (i) process single or multiple sounds (ii) build complex sound structures into a sound stream (iii) view different graphical representations of sounds. Hence, the major difference between this category and the one outlined in the preceding paragraphs lies perhaps in this software's more general usage at the expense of less 'inherent' musical capabilities: as an example, the difference between single event and event organization (the above-mentioned *orchestra/score metaphor* and other organizational forms) which is pervasive in the languages for sound processing hardly exists in this category. However, this software allows direct manipulation of various sound parameters in many different ways and is often indispensable in musical pre-production and post-production stages.

Compared to the Music-N-type software the one of this category belongs to a sort of "second generation" computer hardware: it makes widespread and intensive use of high-definition graphical devices, high-speed sound-dedicated hardware, large core memory, large hard disks, etc. . In fact, we will shortly show that the most hardware-intensive software in music processing - the digital live-electronics real-time control software - belongs to one of the subcategories exposed below.

3.1 Time-Domain Graphical Editing and Processing

The most obvious application for inline sound processing is that of graphical editing of sounds. While text data files lend themselves very conveniently to musical data description, high-resolution graphics are fundamental to this specific field of applications where single-sample accuracy can be sacrificed to a more intuitive sound event global view.

Most graphic sound editors allow to splice and process sound files in different ways.

As fig. 2 shows[13] the typical graphical editor dis-

[13]The editor in this example is the *Digital Audio Processor*, an open-source public-domain audio edit-

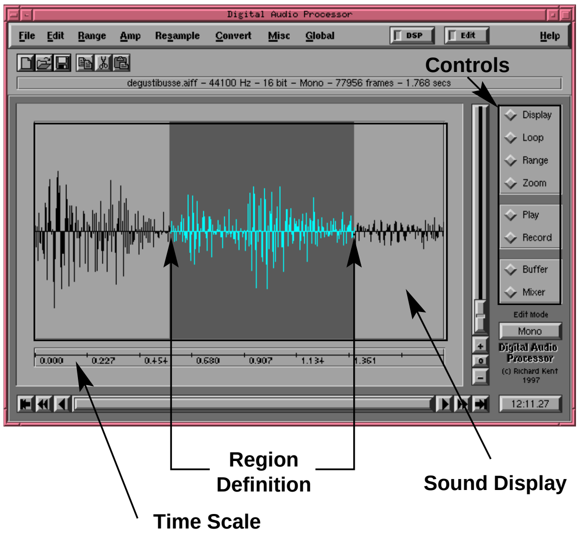

Figure 2: A typical sound editing application

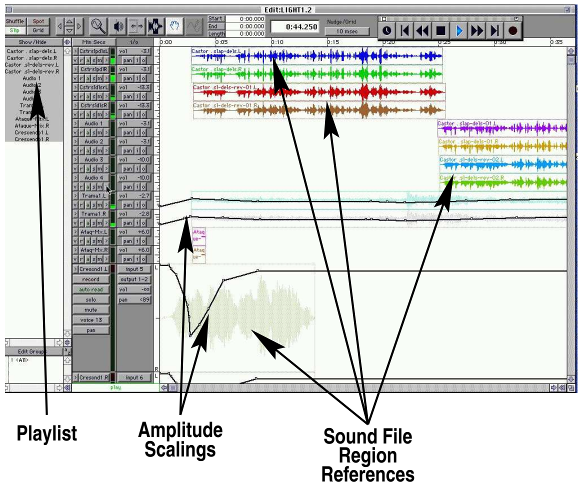

Figure 3: A snapshot of a typical *ProTools*© editing session

plays one or more soundfiles in the time-domain, allowing to modify it with a variety of tools. The important concepts in digital audio editing can be summarised as follows:

- regions - these are graphically selected portions of sound in which the processing and/or splicing takes place;

- in-core editing versus window editing - while simpler editors load the sound in RAM memory for editing, the most professional ones offer buffered on-disk editing to allow editing of sounds of any length: given the current storage techniques, high-quality sound is fairly expensive in terms of storage (ca. 100 kbytes per second and growing), on-disk editing is absolutely essential to serious editing;

- editing and rearranging of large soundfiles can be extremely expensive in terms of hardware resources and hardly lend themselves to the general editing features that are expected by any multimedia application: *multiple-level undos*, *quick trial-and-error*, *non-destructive editing*, etc.: several techniques have been developed to implement these features - the most important one being the *playlist*, which allows soundfile editing and rearranging without actually touching the soundfile itself but simply storing pointers to the beginning and end of each region. As can be easily understood, this technique offers several advantages being extremely fast and non-destructive;

In fig. 3, a collection of soundfiles is aligned on the time axis according to a playlist indicating the starting time and duration of each soundfile reference (i.e. a pointer to the actual soundfile). Notice the on-the-fly amplitude rescaling of some of the soundfiles[14]

Graphical sound editors are extremely widespread on most hardware platforms: while there is no current favourite application, each platform sports one or more widely used editors which may range from the US$ 10000 professional editing suites for the Apple Macintosh to the many free open-source programs for unix workstations. In the latter category, it is worthwile to mention the *snd* application by Bill Schottstaedt [15] which features a back-end processing in CLM (see sec. 2). More precisely, sounds and commands can be exchanged back and forth between CLM and snd, in such a way that the user can choose at any time the most adequate between inline and language-based processing.

3.2 Analysis/Resynthesis Packages

Analysis/Resynthesis packages belong to a closely related but substantially different category: they are generally medium-sized applications which offer different editing capabilities. These packages are termed *analysis/resynthesis* packages because editing and processing is preceded by an analysis phase which extracts the desired parameters in their most significant and convenient form; editing is then performed on the extracted parameters in a variety of ways and after editing, a resynthesis stage is needed to re-transform the edited parameters into a sound in the time domain. In different forms, these applications do: (i) perform various types of analyses on a sound (ii) modify the analysis data (iii) resynthesize the modified analysis.

Many applications feature a graphical interface that allows direct editing in the frequency-domain:

ing and processing application written by Richard Kent (http://www.cee.hw.ac.uk/~richardk/) for unix workstations.

[14] *ProTools*© is manufactured by Digidesign (http://www.digidesign.com)

[15] ftp://ccrma-ftp.stanford.edu/pub/Lisp/snd.tar.gz

the prototypical application in this field is *Audiosculpt* developed at the IRCAM[16] for the Apple Macintosh platform. Based on a versatile FFT-based phase vocoder called *SVP*, Audiosculpt is essentially a drawing program which allows the user to "draw" on the spectrum surface of a sound.

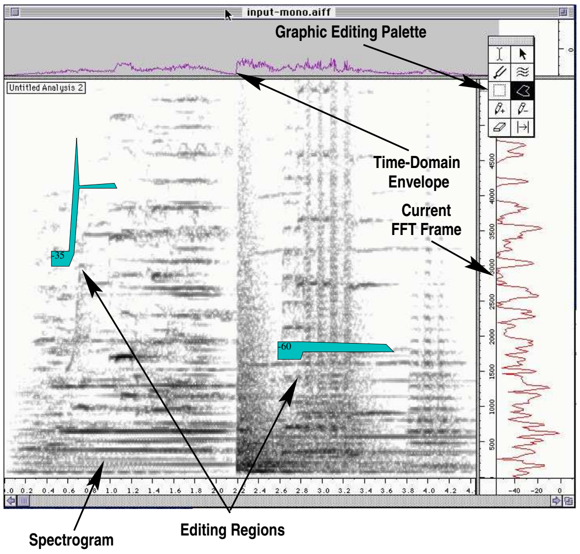

Figure 4: A typical *AudioSculpt* session

In fig. 4, some portions of the spectrogram have been delimited and different magnitude reductions have been applied to them.

Other applications, such as *Lemur*[17], running on Apple Macintoshes) [7] or *Ceres* (developed at NoTam[18]) perform different sets of operations such as partial tracking and tracing, logical and algorithmic editing, timbre morphing, etc.

The contemporary sound designer can also benefit from tools which are specifically designed to transform sound objects in a controlled fashion. One such tool is *SMS* [19], designed by Xavier Serra as an offspring of his and Smith's idea of analyzing sounds by decomposing them into stochastic and deterministic components [16] or, in other words, noise and sinusoids. SMS uses the Short-Time Fourier Transform (STFT) for analysis, tracking the most relevant peaks and resynthesizing from them the deterministic component of sound, while the stochastic component is obtained by subtraction. The decomposition allows flexible transformations of the analysis parameters, thus allowing good-quality time warping, pitch contouring, and sound morphing. SMS comes with a very appealing graphical interface under Microsoft Windows, with a web-based interface, and is available as a command-line program for other operating systems, such as the various flavors of unix. SMS uses an implementation of the Spectral Description Interchange Format[20], which could potentially be used by other packages operating transformations based on the STFT. As an example, consider the following SMS synthesis score which takes the results of analysis and resynthesizes with application of a pitch-shifting envelope and an accentuation of inharmonicity:

```
InputSmsFile march.sms
OutputSoundFile exroc.snd
FreqSine 0 1.2 .5 1.1 .8 1 1 1
FreqSineStretch 0.2
```

3.3 Interactive Graphical Building Environments

In recent times, several software packages have been written to ease the task of designing sound synthesis and processing algorithms. Such packages make extensive use of graphical metaphors and object abstraction reducing the processing flow to a number of small boxes with zero, one or more audio/control inputs and outputs connected by lines, thus replicating once again the old and well known modular synthesizer interface taxonomy.

Initially, many such packages where created to tame the daunting task of writing specialized code for dedicated signal processing tasks. In these packages, each object would contain some portion of DSP assembly code or microcode which would be loaded on-demand in the appropriate DSP card. With a graphical interface the user would easily construct, then, complex DSP algorithms with detailed controls coming from different sources (audio, MIDI, sensors, etc.). Several such applications still exist and are fairly widely used in the live-electronics music field (just to quote a few of the latest (remaining) ones): the *Kyma/Capybara* environment written by Carla Scaletti and Kurt Hebel[21], the *ARES* software developed by the software team at IRIS-Bontempi[22], the *Fly30* environment written by Michelangelo Lupone and Laura Bianchini at CRM-Rome[23], and the soon-to-be-released *Scope* package announced by the german firm Creamware[24].

While these specialized packages are bound to disappear with the rapid and manifold power increase of general purpose processors[25], the concept of graphic object-oriented abstraction to easily visually construct signal processing algorithms has spur an entire new line of software products.

The most widespread one is indeed the *Max* package suite conceived and written by Miller Puckette at

[16] http://www.ircam.fr
[17] http://datura.cerl.uiuc.edu/Lemur/
[18] http://www.NoTam.uio.no/
[19] http://www.iua.upf.es/~sms/
[20] http://cnmat.cnmat.Berkeley.edu/SDIF/
[21] http://www.symbolicsound.com
[22] http://aimi.dist.unige.it/IRIS/index.html
[23] http://www.axnet.it/crm
[24] http://www.creamware.de
[25] This is not a personal but rather a classic darwinian consideration: the maintenance costs of such packages added to the intrinsinc tight binding of such code with rapidly obsolescent hardware exposes them to an inevitable extinction.

IRCAM. Born as a generic MIDI control logic builder, this package has known an enormous expansion in its commercial version produced by Opcode Inc.[26] and maintained by Dave Zicarelli[27]. A recent extension to Max is *MSP* which features real-time signal processing objects on Apple PowerMacs (i.e. on general-purpose RISC architectures). Another interesting path is being currently followed by Miller Puckette himself who is now developing *Pure Data* (PD) [13], an open-source public domain counterpart of Max which handles MIDI, audio and graphics. PD is developed keeping the actual processing and its graphical display as two cooperating separate processes, thus enhancing portability and easily modeling its processing priorities (sound first, graphics later) on the underlying operating system thread/task switching capabilities. PD is currently a very early-stage work-in-progress but it already features most of the graphic objects found in the experimental version of Max plus several audio signal processing objects. Its tcl/tk graphical interface makes its porting extremely easy (virtually "no porting at all")[28].

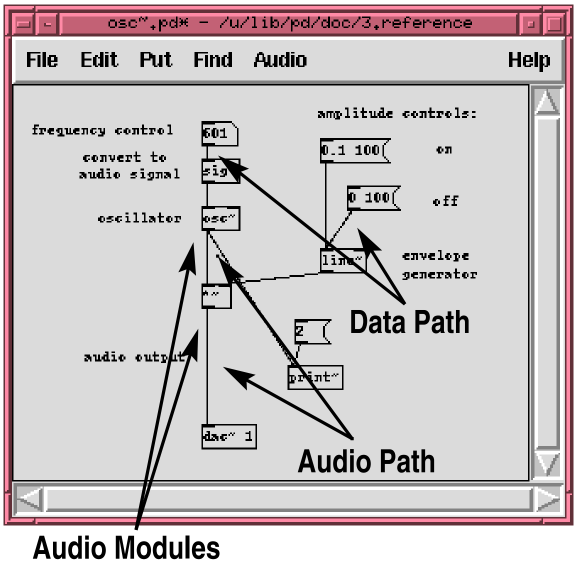

Figure 5: A *Pd* screen shot

4 Software to Teach Audio Signal Processing

Audio signal processing is essentially an engineering discipline. Since engineering is about practical realizations the discipline is best taught using real-world tools rather than special didactic software. At the roots of audio signal processing there are mathematics and computational science: therefore we strongly recommend using one of the advanced maths softwares available off the shelf. In particular, we experienced teaching with *Matlab*, or with its open-source counterpart *Octave* [29]. Even though much of the code can be ported from Matlab to Octave with minor changes, there can still be some significant advantage in using the commercial product. However, Matlab is expensive and every specialized toolbox is sold separately. True, there is an inexpensive student edition, but with a very severe limitation on vector size precluding any serious manipulation of sounds. On the other hand, Octave is free software distributed under the GNU public license. It is robust, highly integrated with other tools such as Emacs for editing and GNUPlot for plotting.

In Matlab/Octave, monophonic sounds are simply one-dimensional vectors, so that they can be transformed by means of matrix algebra, since vectors are first–class variables. In these systems, the computations are vectorized, and the gain in efficiency is high whenever looped operations on matrices are transformed into compact matrix-algebra notation [1]. This peculiarity is sometimes difficult to assimilate by students, but the theory of matrices needed in order to start working is really limited to the basic concepts and can be condensed in a two-hours lecture.

Matlab and Octave are great tools for illustrating the concepts of sampling, quantization, aliasing, windowing, etc. It is possible to visualize spectra and signals under different conditions with very small scripts. A Signal Processing Toolbox with plenty of routines for signal manipulation and filter design is purchasable for being used within Matlab. However, pedagogical needs can be largely satisfied by public-domain routines[30]. By using these routines it is possible, for example, to plot a time-frequency representation of a sound `S` by introducing the two lines:

```
SS = stft(S);
mesh(20*log10(SS));
```

A course reader (in italian) on sound processing with several examples produced using Octave is available on line [31].

An application devoted to digital filter design with pedagogical purposes in mind is called *ein* [9]. It is essentially a front end to a C/C++ library of functions dealing with signal processing in both time and frequency domain. Currently, an open-source implementation is available for Silicon Graphics workstations while a linux port is well on the way at the time of this writing[32].

[26] http://www.opcode.com

[27] http://www.cycling74.com

[28] *Pure Data* currently runs on Silicon Graphics workstations, on Linux boxes and on Windows NT platforms; sources and binaries can be found at ftp://crca-ftp.ucsd.edu/pub/msp/

[29] http://www.che.wisc.edu/octave/

[30] see, e.g. http://www.tsc.uvigo.es/GTS/Octave

[31] http://www.dei.unipd.it/ricerca/csc/roc/web/es.html

[32] a web page on ein may be found at http://www.music.princeton.edu/PSK/ein.html while the linux port will soon be available from ftp://musart.dist.unige.it/pub/CSOUND/ein-1.xx.tar.gz and ftp://mustec.bgsu.edu/pub/linux/

5 Processing Libraries, Plug-ins and Toolkits

Aside from complete and self-contained applications, another musical field in which several energies are being spent is that of tools and toolkits to perform specific processing tasks in a cooperative environment made of several small instances of such programs. These tools have different names (e.g. *libraries*, *plug-ins*, *toolkits*, etc.) and take up different forms (e.g. libraries of C functions and modules, dynamically linked modules, full-blown applications, etc.) but they essentially serve the same purpose. One of the most effective ways of providing extensibility to a software system is to design it as a suite of independent programs, and to let them communicate by means of mechanisms of the underlying operating system, e.g. unix pipes and Inter-Process Communication. An interesting example of this kind of design is found in *pipewave*, a suite of programs designed to arrange and analyze psychoacoustic tests. Data are passed between programs as ASCII streams using unix pipes. The choice of ASCII as a format for signals allows surgical operations via text editors, effective compression via regular compression tools, and manipulation by independently-conceived programs, such as Octave (see sec. 4). As an example of pipewave in action, consider the task of plotting and storing to file the lowpass-filtered version of a 1-second gaussian white noise sampled at 22050 Hz. This can be easily achieved by the following script, where filtering is performed in the frequency domain:

```
gaussian -s 22050 22050 | fft -p |
ffilt -c 0 -10 2000 -40 | ift |
store -o fnoise | plot
```

A strong advantage of using pipes is that the communicating programs can be written in entirely different programming languages and environments, so that it is possible to overcome the limitations of every single programming system.

Pipes have been used extensively in the *CARL* software environment of the University of California, San Diego too [12]. This environment includes the Cmusic sound-processing language and a plethora of tools such as reverberators, a spatializer, table generators, all communicating via floating-point streams passed through pipes. In practice, the Cmusic language can be extended without modifying the Cmusic program itself.

Cmix is another sound-processing system which is based on the idea of having many small programs rather than a monolithic software [33]. However, while in the CARL software environment there are satellites of a Music-N-like core, in Cmix there is not such a core, as there are just C functions which can be called within regular C programs. This greatly increases the possibilities of the sound designer, even though extensive knowledge of the C programming language is required. In particular, compact scores can be written using complex control structures, thus extending the crude note-list habit of Music-N-like languages. Cmix has been recently turned into a C++ system of classes, fitted with an efficient scheduler and with support for remote client requests via TCP/IP sockets. The new system is called *RTcmix*[34] [8], to emphasize the fact that sound computations can be done in real-time on stand-alone or networked workstations. A similar system is Perry Cook's *Synthesis Toolkit* (STK)[35]: a collection of sound-processing modules written as C++ classes which are particularly effective for representing complex sound synthesis patches, such as those found in physical models. STK runs under unix or Microsoft Windows, and supports real-time input/output audio and MIDI streaming.

Other widely diffused tools (in the Macintosh world) are the *GRM Toolkit* [6] and *Soundhack*, a stand-alone set of processing algorithms written by Tom Erbe[36]. The *GRM Toolkit* was initially designed as a stand-alone application to replicate in the digital domain the effects that were a special feature of the Groupe de Recherches Musicales active at the french radio since 1948. The effects included many types of filtering, delay and resampling modules and algorithmic splicing. In recent times these tools are being offered (commercially) as plug-in modules for popular Macintosh editors/sequencers like *ProTools*© and *Cubase VST*©. SoundHack is a well-designed stand-alone application (see fig. 6) which offers traditional tools coupled with less traditional processings like hybrid mutations, binaural placement and wavetable convolution.

6 Other issues

There are several general software development issues that do not belong to any particular music software approach or category. However, in music software these issues have often been underestimated: with few exceptions in both the commercial and the open-source domains music software is known to be badly designed, poorly developed and consequently often bugged by ill-defined problems.

6.1 Portability and availability under different platforms

Portability problems manifest themselves differently in the commercial and in the open-source domains.

[33] http://www.music.princeton.edu/cmix

[34] http://www.music.columbia.edu/RTcmix

[35] http://www.cs.princeton.edu/~prc

[36] http://shoko.calarts.edu/~tre/SndHckDoc

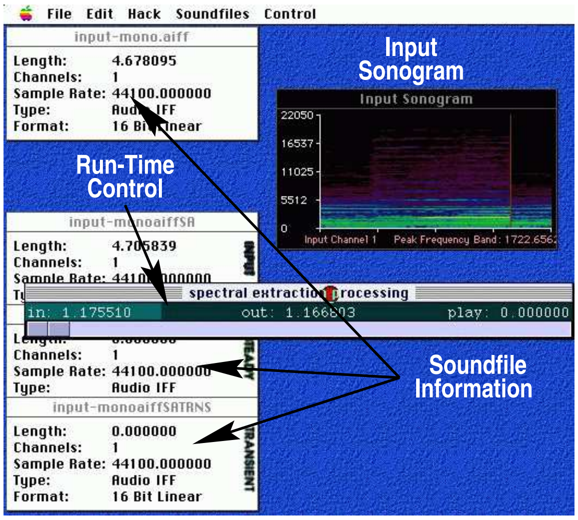

Figure 6: A *SoundHack* screen shot

In the commercial domain, up until very recently the software houses assumed that a musician would be such a computer illetterate to be forced to choose her/his computer platform according to the software used (and not the opposite, as normal logic would lead to think). In the best case, *interoperability*, *file exchange*, *inter-application communication* were words allowed only among the same firm's products on the same platform (and often the same version number too). With recent changes in commercial operating system trends and politics (combined with musicians becoming a little pickier about computers) these software houses were faced with some mandatory multi-platform software shift, which will probably lead, in some future, to better thought-out designs. Another problem that plagues the design of many commercial applications is the need to fulfill many radically different functions to satisfy the widest customer base: the applications are often too big, slow and non-homogeneous.

Since Computer Music is fairly diffused in the academic field, it is perfectly natural to find a large pool of high-quality open-source applications devoted to signal processing in the public domain (and indeed, a good 70% of the software mentioned in this document belongs to the open-source public domain). As usual, the very nature of successful open-source applications[37] implies a large base of users/debuggers/experts; hence, successeful open-source applications often grow more rapidly and are better debugged. However, even in this domain music applications do not yet follow the standards: while a lot of the open-source software constitute a model for coding and development, for some obscure reason many open source music software developers do not analyze nor code in a professional way and they refrain from using all the powerful development tools that the public-domain and the internet feature[38]. Professional music software development is still a rare (and welcome) instance in the open-source community. On the positive side, as all other open-source software, music applications stem from some precise real-world needs and are generally better suited to face sophisticated musical requests. Furthermore the widespread use of ASCII (text) data description encoding makes inter-application operation extremely easy and natural.

6.2 Integration with existing practice and working enviroments

Existing practice is a very diffused statement in music: existing practices are important in music composition, typography, interpretation, rehearsals, performance, analysis, concert habits - almost any musical field operates in conformance (or in opposition) to some existing practice[39]. Computer music software is no exception: even though it is a very recent field there are indeed several consolidated practices in electro-acoustic music performance (we even made some passing reference to some of them in this paper: the *orchestra-score* metaphor, the *modular synthesizer* model, etc.).

While it is often (wrongly) assumed that musicians will prefer simpler software with friendlier interfaces and lack of deeper complexity, it is interesting to notice (even if it could sound a bit obvious) that music software designed with existing practices in mind is often more easily accepted and used by musicians and therefore more successeful - no matter how complicate its operation is. Music production is a complex task full of many-layered implications and musicians are certainly not afraid of facing it every day - musical instruments are complicate interfaces which require years of practice at all levels: in this context, computers are certainly among the simplest instruments (even mere toys at times) and musicians are certainly eager to learn them if the software is designed with the musical context and the existing practices in mind.

References

[1] D. Arfib. Different ways to write digital audio effects programs. In *Proc. Conf. Digital Audio Effects (DAFX-98)*, Barcelona, Spain, Nov. 1998.

[2] R. B. Dannenberg. Abstract time warping of compound events and signals. *Computer Music J.*, 21(3):61–70, 1997.

[37] which is well explained in Eric Raymond's article *Homsteading the Noosphere*, available from http://sagan.earthspace.net/~esr/writings/homesteading/

[38] Perhaps a musician that devotes time to music thinks she/he doesn't have the time to learn software tools; unfortunately that very same time (and much more) gets spent hunting for hard-to-catch bugs and rewriting code.

[39] this probably means that the music world is very conservative.

[3] R. B. Dannenberg. The implementation of Nyquist, a sound synthesis language. *Computer Music J.*, 21(3):71–82, 1997.

[4] R. B. Dannenberg. Machine tongues XIX: Nyquist, a language for composition and sound synthesis. *Computer Music J.*, 21(3):50–60, 1997.

[5] R. B. Dannenberg and N. Thompson. Real-time software synthesis on superscalar architectures. *Computer Music J.*, 21(3):83–94, 1997.

[6] E. Favreau. Les outils de traitement GRM Tools. In *La Terra Fertile*, pages 95–98, L'Aquila, Italy, Sept. 1998.

[7] K. Fitz and L. Haken. Sinusoidal modeling and manipulation using lemur. *Computer Music J.*, 20(4):44–59, 1997.

[8] B. Garton and D. Topper. RTcmix – using CMIX in real time. In *Proc. International Computer Music Conference*, pages 399–402, Thessaloniki, Greece, 1997. ICMA.

[9] P. Lansky and K. Steiglitz. Ein: A signal processing scratchpad. *Computer Music J.*, 19(3):18–25, 1995.

[10] M. Mathews, J. E. Miller, F. R. Moore, J. R. Pierce, and J.-C. Risset. *The Technology of Computer Music.* MIT Press, Cambridge, MA, 1969.

[11] F. R. Moore. *Elements of Computer Music.* Prentice-Hall, Englewood Cliffs, N.J., 1990.

[12] S. T. Pope. Machine tongues XV: Three packages for software sound synthesis. *Computer Music J.*, 17(2):23–54, 1993.

[13] M. Puckette. Pure data. In *Proc. International Computer Music Conference*, pages 224–227, Thessaloniki, Greece, Sept. 1997. ICMA.

[14] C. Roads. *The Computer Music Tutorial.* MIT Press, Cambridge, Mass., 1996.

[15] B. Schottstaedt. Machine tongues XVII: CLM: Music V meets common lisp. *Computer Music J.*, 18(2):30–37, 1994.

[16] X. Serra and J. O. Smith. Spectral modeling synthesis: A sound analysis/synthesis system based on a deterministic plus stochastic decomposition. *Computer Music J.*, 14(4):12–24, 1990.

[17] B. L. Vercoe, W. G. Gardner, and E. D. Scheirer. Structured Audio: Creation, Transmission, and Rendering of Parametric Sound Representations. *Proc. IEEE*, 86(5):922–940, May 1998.