cage: a high-level library for real-time computer-aided composition

Andrea Agostini
HES-SO, Geneva
and.agos@gmail.com

Éric Daubresse
HES-SO, Geneva
eric.daubresse@hesge.ch

Daniele Ghisi
HES-SO, Geneva
danieleghisi@gmail.com

ABSTRACT

This paper is an introduction to *cage*, a library for the Max environment [1] including a number of high-level modules for algorithmic and computer-aided composition (CAC). The library, in the alpha development phase at the time of writing, is composed by a set of tools aimed to ease manipulation of symbolic musical data and solve typical CAC problems, such as generation of pitches, generation and processing of melodic profiles, symbolic processes inspired by digital signal processing, harmonic and rhythmic interpolations, automata and L-systems, tools for musical set theory, tools for score generation and handling. This project, supported by the *Haute École de Musique* in Geneva, has a chiefly pedagogical vocation: all the modules in the library are abstractions, lending themselves to be easily analyzed and modified.

1. INTRODUCTION

This article describes some of the main concepts and components of the *cage* [2] library for Max, containing several high-level modules for computer-aided composition (CAC). Some of these modules have already been discussed in [1] (in French); in this paper we complete the overview of the library, and provide a more comprehensive view on its goals.

cage is entirely based upon the *bach: automated composer's helper* library, which is developed by two of the authors [2, 3]. *bach* is a library of about 200 Max externals and abstractions, aimed to bring within Max a set of 'primitives' for the manipulation of symbolic musical data, along with some GUIs for their graphical representation and editing. Data within *bach* are invariantly represented through specialized uses of a generic data structure, the *llll* ('Lisp-like linked list'), which as the acronym suggests is essentially a tree structure in the form of a nested list, directly inspired by the Lisp programming language. Subsequently, most *bach* modules are tools for low-level manipulation of *llll*s (performing operations such as rotations, substitutions or retrieval of single elements) or for more complex but conceptually basic operations such as constraint solving or rhythmic quantization. Differently from *bach*, *cage* modules in general perform higher-level tasks, with a compositional rather than strictly technical connotation (e.g. melodic material generation, or computation of symbolic frequency modulation). Still, some basic mechanisms and principles are inherited by *cage* from *bach*, including the fact that communication between the different modules happens mostly by means of *llll*s.

Two main criteria have informed the conception of the library.

The first is the idea at the very root of *cage* itself: building a library of ready-to-use modules, implementing a number of widely used CAC processes. As a consequence, a part of the library is openly inspired by libraries already existing for other programs (namely the *Profile* [4] and *Esquisse* [5, 6] libraries for *Patchwork*, which have been subsequently ported to *OpenMusic*); on the other hand, another part of the library is addressed to problems and practices typically associated with real-time interaction (such as *cage.granulate*, the symbolic granulation engine).

Secondly, the project has a strong pedagogical connotation [3]: all the modules of the library are abstractions, lending themselves to be easily analyzed and modified. It is not difficult, for the user wishing to learn how to treat musical data, to copy, edit or adjust the patches to his or her own needs. In this regards, all the tools in the library are intrinsically 'open source': although each implemented process is conceived for a typical, somehow standard usage, the advanced user will easily start from these abstractions and modify their behavior. This pedagogical connotation is completed by the fact that the library will be thoroughly documented by help files, reference sheets and a collection of tutorials.

2. A REAL-TIME APPROACH TO COMPUTER-AIDED COMPOSITION

The real-time paradigm deeply influences the nature itself of the compositional process. For example, composers who work in the domain of electro-acoustic music often need the computer to react immediately to each parameter change. Similarly, composers working with symbolic data may wish the computer to adapt within the shortest delay to a new configuration of the data themselves. *cage*'s underlying paradigm is ultimately the same that informed the *bach* library: creating and editing symbolic musical data is not necessarily an out-of-time activity, but it follows the temporal flow of the compositional process, and adapts to

[1] *http://cycling74.com*
[2] *www.bachproject.net/cage*

[3] The *cage* library is supported by a grant from HES-SO.

it (see also [3, 7, 8]).

3. COMPOSITION OF THE LIBRARY

The library is composed by several families of modules. In the following paragraphs we will briefly describe them, in order to give an idea of the scope of the work. Of course, there is no ambition of completeness in the choice of the processes that have been implemented. Computer-aided composition is a vast domain, and practices are personal and specific to each single composer more often than not. Still, it seems to us that some general typologies of approaches, as well as some commonly used specific operations, can be discerned. We attempted to exemplify at least some of them, hoping that our work will be useful to composers wishing to implement their own individual processes and operations.

3.1 Pitch generation

The first family of modules that will be discussed is aimed at generating pitches according to different criteria: *cage.scale* and *cage.arpeggio* can generate respectively scales and arpeggios within a given pitch range. The types of chords and scales can be expressed either through symbolic names or midicents patterns. Scale and chord names can contain quartertones and eight-tones as well. *cage.harmser* generates harmonic series starting from a given fundamental, with an optional distortion factor.

Other modules generate pitches on a one-by-one basis: *cage.noterandom* generates random notes from a given reservoir, optionally according to different predefined probability weights, which can be defined, for instance, through *cage.weightbuilder*; *cage.notewalk* generates an aleatory path in a given reservoir, according to a list of allowed steps. In both cases, the result of the operation is meant to be used in combination with *bach.transcribe*, which will transcribe the incoming stream of notes in real time. Also, in both cases the randomly chosen element can be validated by the user via a *lambda loop*. [4]

3.2 Generation and treatment of melodic profiles

A family of modules is specifically aimed at generating and treating melodic profiles, in a similar fashion to the *Profile* library in OpenMusic and PatchWork [4]. A breakpoint function can be converted in a sequence of pitches (a melodic profile) through *cage.profile.gen*. This profile can be edited in different ways: it can be compressed or stretched (with *cage.profile.stretch*), reversed (with *cage.profile.mirror*), approximated to an harmonic grid or a scale (with *cage.profile.snap*), forced into a pitch range (with *cage.profile.rectify*), randomly perturbed (with *cage.profile.perturb*) or filtered (with *cage.profile.filter*). Profile filtering is achieved through application of an average, median or custom filter, the latter being definable by the user through a *lambda loop* (see also Fig. 1).

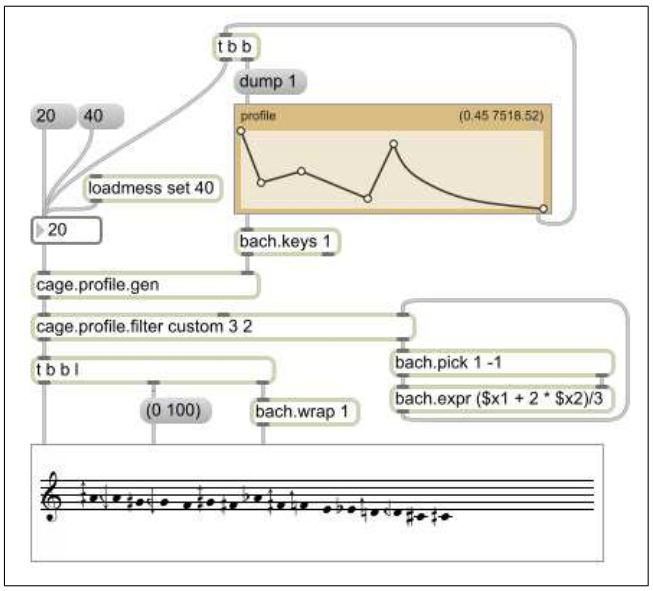

Figure 1. A melodic profile is built from a function defined inside a *bach.slot* object and sampled over 20 points. Then, the profile is filtered by a process expressed through a *lambda loop*, which operates on three-note windows; each window is replaced by a single value, the average of the first and last element of the window itself weighted by the weights $(1, 2)$. This filtering process is repeated twice. It can be observed that, because of the windowing, the result contains four notes less than the original sampling.

3.3 Processes inspired by electro-acoustic practices

cage contains a group of modules dedicated to symbolic emulation of processes belonging to the domains of sound synthesis and digital audio processing.

cage.freqshift is a tool allowing transposition of materials linearly on the frequency axis, as in single-sideband ring modulation. Because of the strict similarity of the two processes, *cage.pitchshift* is considered as belonging to the same category, although a pitch shifting operation applied to musical notation is just a simple transposition.

cage.rm and *cage.fm* deal respectively with ring modulation and frequency modulation. The idea underlying these techniques, widely employed by composers associated with the spectral movement, is the following: starting from two chords (a 'carrier' and a 'modulating' chord), whose notes are considered as a simple sine tones, the spectrum obtained by modulating with each other these two groups of sinusoids is calculated. Each component of the resulting spectrum is then represented as a note of the resulting chord. This operation requires a number of approximations and trade-offs which can make its result significantly different from the actual product of the corresponding audio treatment: nevertheless, it is a very effective approach in generating rich harmonic families from

[4] A *lambda loop* in *bach* and *cage* is a symbolic feedback configuration: objects supporting this behavior have one or more dedicated *'lambda'* outlets returning data for acceptance or modification; these data are processed in a specific section of the patch whose resulting value is fed back into a dedicated *'lambda'* inlet of the first object. This configuration is often employed within *bach* in order to define custom behaviors for specific operations (e.g. a sorting criterium, or a process to be applied to every element of an *llll*). The name 'lambda' hints to the fact that this configuration somehow allows to pass a section of a patch as a pseudo-argument of an object. Indeed, this is nothing more than an allusion: there is no lambda calculus or interpreted functions involved in the process.

simple materials, hence its compositional interest. Although the direct inspiration for *cage.rm* and *cage.fm* is taken from the *Esquisse* library [5, 6] for *OpenMusic*, their operational paradigm and some computational details are different. In particular, being conceived to work in time, these two modules can accept not only simple chords, but also chords sequences representing variations of 'carriers' and 'modulating chords' in time. In this case, the process will return a new score containing the result of these variations in time (see Fig. 2). For what concerns the actual internal computation, the two modules take into account an estimate of the phase oppositions generated by the modulation, and the relative component elision, differently from what happens in the *Esquisse* library. For this reason, the results of the same process in the two environments can be significantly different.

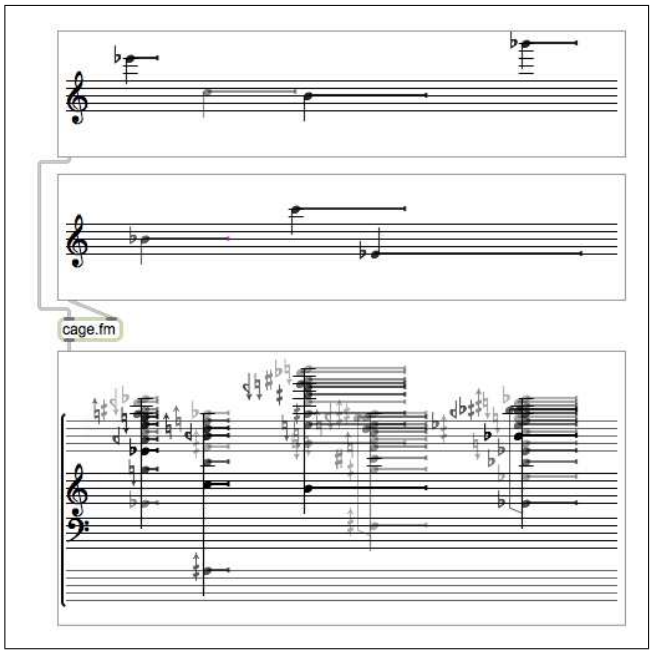

Figure 2. An example of frequency modulation of two scores, achieved through the *cage.fm* abstraction. The 'carrier' and 'modulating' are on top, the result below. The note velocity (treated as the amplitude of the corresponding sinusoidal components) is represented in grayscale.

cage.virtfun returns one or more estimates of the virtual fundamental frequency of a chord, as perceived for example at the output of a waveshaping process. The implementation is very simple: the sub-harmonic series of the lowest note of the chord is traversed until a frequency whose harmonics approximate all the notes of the given chord, within a given tolerance, is found. *cage.virtfun* can also be applied to a sequence of chords in time; in this case, the result will be the sequence of the virtual fundamentals. On the other hand, the numerical operation performed by *cage.virtfun* has a broader range of applications: it can be considered a computation of an approximate greatest common divisor of a set of numbers. As such, it is called for example by *cage.accrall* to establish a 'reasonable' minimal rhythmic unit in a non-measured score.

cage.delay and *cage.looper* extend the concept of delay line with feedback in the symbolic domain. Their aim is creating loops and repetitive structures in which the material can be altered at each pass through a *lambda loop*. The difference between the two lies in the musical unit that is passed to the *lambda loop*: a single chord in *cage.delay*, a whole section of the score in *cage.looper*. In both cases, the delay time itself can be changed for each repetition. In principle there is no limitation to the richness of the processes that can be applied to the material in the *lambda loop*: the musical result can therefore be much more complex than a simple iteration.

cage.cascade∼ and *cage.pitchfilter* extend the principle of filtering to the symbolic domain. The former applies a chain of two-pole, two-zeros filters to a score, as the *biquad*∼ and *cascade*∼ Max objects, by emulating the actual frequency response of a digital IIR filter. The latter operates directly on pitches, rather than frequencies, by applying to a score a filter defined by a breakpoint function obtained for example from a *function* or a *bach.slot* object. In both cases, the MIDI velocity of each note is modified according to the filter response, and notes whose velocities fall below a given threshold are removed. Interpolation between different filter configurations in time is also possible (see fig. 3).

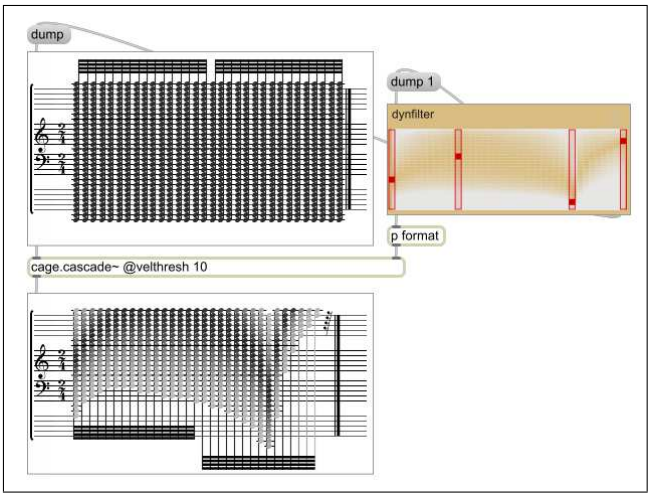

Figure 3. An example of dynamic filtering of a score obtained through cage.cascade∼ driven by a *dynfilter* slot in a *bach.slot* object. Whenever the filter parameters are edited through the interface, the result is automatically updated in real time.

cage.granulate is a symbolic granulation engine. The parameters of the granulation are the same as in the corresponding electro-acoustical process: the time interval between two grains, the size of each grain, the beginning and the end of the temporal region from which the grain must be extracted. Based upon these parameters, *cage.granulate* fills in real time a *bach.roll* object connected to its outlet.

3.4 Harmonic and rhythmic interpolation, formalization of agogics

The *cage.chordinterp* abstraction performs a linear harmonic interpolation between a set of chords, through the assignment of different weights to each of them. In the same

way, a rhythmic interpolation can be obtained through the module *cage.rhythminterp*.

cage.timewarp on the other hand performs a temporal distortion of a score, obtained through a function (in the usual form of a *lambda loop*) that is applied to the onset of each discrete event of the score. Among the other things, this provides a flexible way to perform any kind and shape of rallentando or accelerando through the definition of the appropriate function - a task that is eased by the *cage.accrall* abstraction, allowing to express agogics through a set of high-level parameters such as total resulting duration or starting and ending speed.

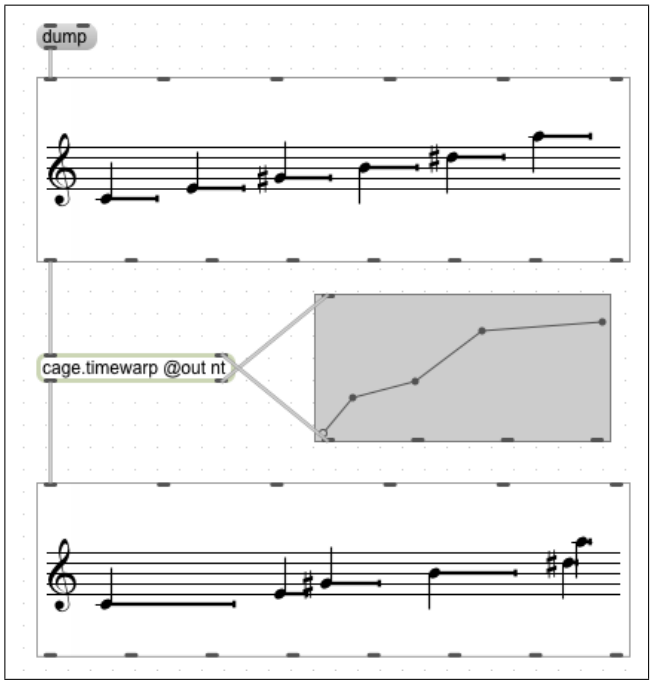

Figure 4. An example of temporal distortion performed through *cage.timewarp*. The function in the *lambda loop* associates time in the original score (above), represented on the x-axis, to time in the resulting score (below), represented on the y-axis.

3.5 Automata, L-systems, etc.

The *cage.chain* abstraction implements one-dimensional cellular automata and L-systems. It performs rewrites of a given list according to a set of rules defined by the user through either messages or a *lambda loop*. Substitutions can take place on single elements (e.g. a certain letter or note is substituted by a list of letters or notes), or overlapping sequences of elements with a fixed length (e.g., each couple of elements is replaced by one or more different elements); in the latter case, *cage.chain* will manage the behavior at the boundaries according to the values of some specific attributes (*pad*, *align*). In summary, this module makes it easy to build cellular automata, or fractals by substitution.

cage.life deals with two-dimensional cellular automata (the most famous example being John Conway's 'game of life'). The rules for these automata are defined through a *lambda loop*. The order of the substitution sub-matrices can be defined by the user as well.

An abstraction closely related to the two previous ones is *cage.lombricus*, implementing a way to build rule-based generative systems. The module accepts a set of starting elements grouped into families, with a weight assigned to each family. The task of the abstraction is creating a sequence of an arbitrary number of elements, trying to match the relative number of occurrences of elements of each family to the weight associated to the family itself. At runtime, the *lambda loop* of the abstraction is fed with proposals of elements to be chained to the existing sequence, along with the whole sequence built so far; each proposal can be refused, or accepted and assigned a score according to custom-definable rules: among the accepted elements, a 'winner' will be chosen according to the score and the weights of the family to which it belongs. If at some point a suitable element cannot be found, the abstraction is capable to backtrack on the sequence built so far, and substitute a previously chosen element with a different one with a lower score but potentially allowing a longer chain to be built. It should also be pointed out that the element needs not to be copied literally in the resulting sequence: for example, the user might want to provide the system with a set of intervals as starting elements, and obtain a melodic sequence at the end of the process: the substitution can be performed within the *lambda loop* described above. In summary, the underlying mechanism of the *cage.lombricus* abstraction shares some features of cellular automata and L-systems on one hand (in particular, a rule-based constructive behavior allowing rewrites), and constraint satisfaction problems on the other (the ability to make choices according to weights and the backtracking behavior), without strictly belonging to either category. Although this process may appear cumbersome, a thorough investigation on our own compositional practices as well as those of other composers (and firstly Michaël Jarrell's) suggested us that it is well-suited to model a wide array of real-life musical formalization techniques.

3.6 Musical set theory tools

A group of modules in *cage* deals with pitch representations typical of the set theory: *cage.chroma2pcset* and *cage.pcset2chroma* convert between pitch class sets and chroma vectors (see [9]); *cage.chroma2centroid* and *cage.centroid2chroma* convert between chroma vectors and spectral centroids, the latter being obtained through the transform described by Harte and Sandler [10]. Going from chroma vector to centroid causes a loss of information, therefore the conversion is not univocal: a single chroma vector, among all those having the input vector as their centroid, is returned.

3.7 Scores

cage contains a set of modules for the global processing of scores: *cage.rollinterp* interpolates between the contents of two bach.roll objects, according to an interpolation curve or a single value in the case of static interpolation.

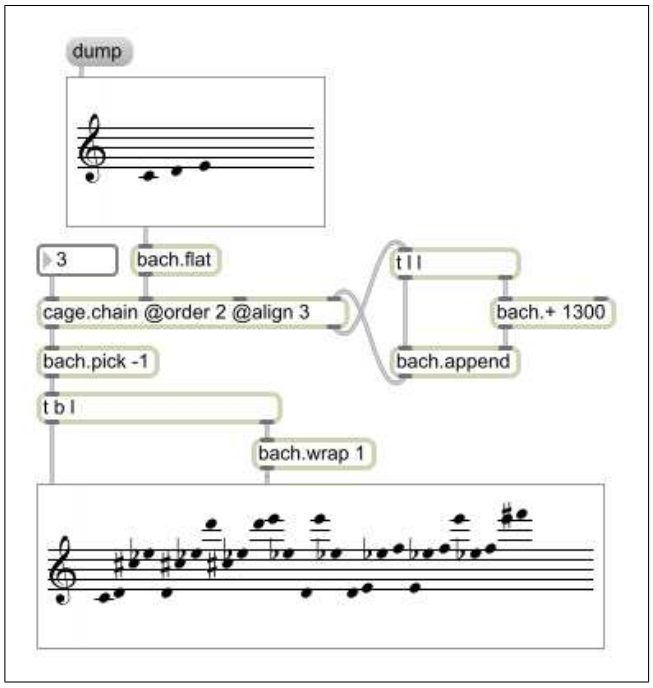

Figure 5. The bottom *bach.roll* shows the sequence of notes produced starting from the top *bach.roll*, and by applying three steps of the substitution rule given in the lambda loop. Such substitution rule states appends to every couple of overlapping notes (*order* is 2) the same couple transposed by one octave plus one semitone. For instance, at the first step, the couple **C4 D4** is substituted with **C4 D4 C#5 Eb5**, and the couple **D4 E4** is substituted with **D4 E4 Eb5 F5**, yielding the sequence **C4 D4 C#5 Eb5 D4 E4 Eb5 F5**; the following steps do the same with the result obtained from the previous step. *cage.chain* then outputs the whole sequence of steps; only the ending one is displayed.

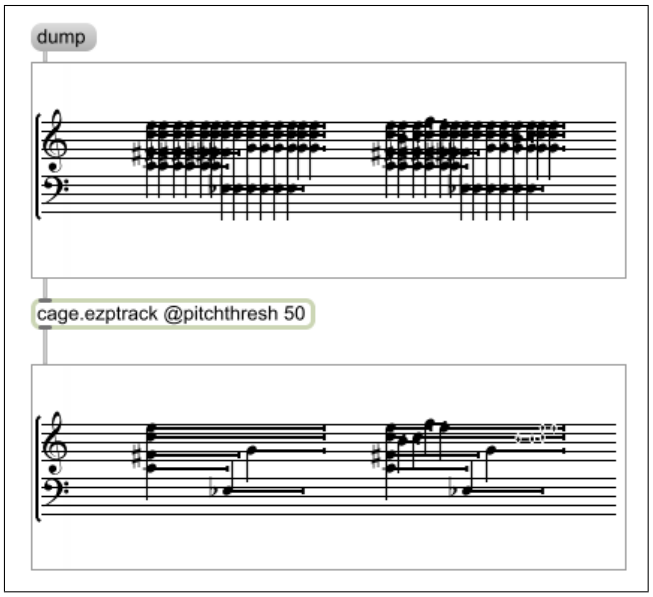

Figure 6. Partial tracking on sequences of chords can be quickly and easily be obtained via *cage.ezptrack*. Here, the pitch threshold to link two consecutive peaks is 50mc. Notice the presence of pitch breakpoints in at the end of the lower *bach.roll*, due to the fact that at the end of the upper roll some notes were not perfectly snapped to the semitone grid.

cage.envelopes represents a family of functions synchronized to the total duration of a score, aiding real-time editing of the score with respect to the values of the curves at each instant. *cage.scissors* divides the score contained in a *bach.roll* object according to vertical (time) and horizontal (voice) split points, and returns a matrix containing the resulting score excerpts. *cage.glue* performs the opposite operation: fills a single *bach.roll* with the contents of a matrix of smaller scores, according to the temporal and voice disposition implicit in the matrix itself, or to an explicitly set disposition. *cage.ezptrack* takes a sequence of chords and attempts to reconstruct musical voices, in a similar way to what partial trackers do with harmonic analysis data. (see Fig. 6).

3.8 SDIF files support

A set of modules in *cage* is designed to ease the reading and writing of SDIF files [11, 12]. This family contains sub-families for some of the most common analyses and descriptors, namely fundamental frequency, peaks, partial tracking, markers.

Starting from version 0.7.4, *bach* supports reading and writing SDIF files through the *bach.readsdif* object, a low-level tool reading all the information contained in a SDIF files and structuring it into an *llll*, and the corresponding *bach.writesdif* object, allowing to write SDIF files starting from their *llll* representation. This representation is complete, meaning that feeding the output of *bach.readsdif* into *bach.writesdif* produces an SDIF file perfectly equivalent, if not identical, to the original one. On the other hand, this very completeness makes the representation itself difficult for the user to manipulate.

For this reason, *cage* includes a set of modules implementing a number of basic operations upon the contents of SDIF files. Some directly convert SDIF data into *bach.roll* syntax, for instance *cage.sdif.ptrack.toroll* (see Fig. 7). Other abstractions rearrange SDIF data in an easily accessible form. As an example, *cage.sdif.fzero.unpack* looks for 1FQ0 (fundamental frequency estimate) frames and outputs onsets, frequencies, confidences, score and amplitudes from different outlets as *llll*s structured by stream. Two abstractions deal with partial tracking (*cage.sdif.ptrack.resolve* and *cage.sdif.ptrack.assemble*), allowing to switch between a time-wise and an index-wise representation of the data.

In general, we did not consider the writing of SDIF files starting from symbolic data a common usage scenario, with one possible exception: markers. For this reason, the only abstraction providing a direct translation from a notation object to a SDIF *llll* is *cage.sdif.markers.fromroll*, transferring into it all the markers of a *bach.roll* object, each with its time position and name.

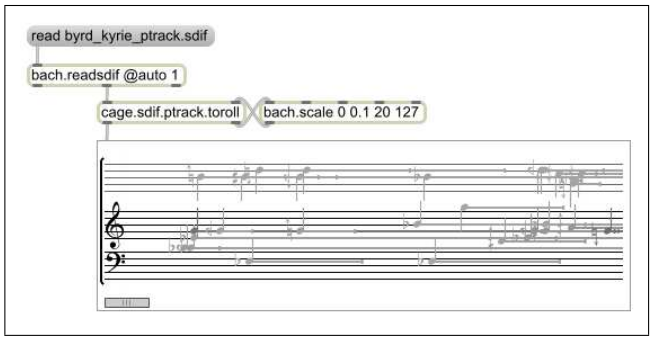

Figure 7. A SDIF partial tracking analysis is imported in a *bach.roll* via *cage.sdif.ptrack.toroll*. The lambda loop is used to define a custom velocity mapping (if no lambda loop is provided, a default mapping will be used).

3.9 Audio rendering

In addition to the previously described proper CAC tools, *cage* contains a set of utilities aimed to make quick prototyping and verification of musical solutions easier. In particular, two modules of the *cage* library perform audio rendering of *bach* scores: *cage.ezaddsynth∼* (a basic additive synthesis engine) and *cage.ezseq∼* (a basic sound file sampler). Like *bach.ezmidiplay*, both are designed to be directly connected to the 'playout' outlet of the *bach.roll* and *bach.score* objects.

The additive synthesis engine addresses the need of a quick-and-dirty audio rendering, overcoming the limitations of MIDI instruments: this may be useful for example when working with non-standard microtonal grids, or when amplitude envelopes, panning or glissandos cannot be ignored. Envelopes should all be defined inside slots. [5]

The sampler addresses the need of using *bach.roll* and *bach.score* as 'augmented sequencers': *cage.ezseq∼* takes into account file names, amplitude envelopes, panning, playback speed, audio filtering, playback starting time (all defined inside slots). *cage.ezseq∼* is also capable to preload audio files, if a given directory is assigned. If requested, the *cage.ezseq∼* module can transpose each sample without temporal alteration (via the *gizmo∼* Max object) according to the pitch of the associated note.

4. CONCLUSIONS

At the time of writing, the library is in an ongoing phase of development. A public alpha version will be available in May 2014: not all the features might be implemented at this point, and the documentation will not be complete. Nonetheless, most modules will already be functional. The first complete version of the library will be made available in October 2014, on the occasion of a public presentation that will take place in Geneva. The library will be freely downloadable. Starting from the academic year 2014-2015, *cage* will be taught within the courses of composition and electronic music at the *Haute École de Musique* in Geneva and in a number of partner institutions.

[5] Slots are metadata of various kind associated to individual notes (see [2]).

Acknowledgments

cage is a research project taking place within the center of electroacoustic music of the *Haute École de Musique* in Geneva, supported by the music and arts domain of the scene of the *Haute École Specialisée* of Western Switzerland. The name *cage*, which in the context of German note names represents the famous expansion of *bach*, is also an acronym acknowledging this support: *composition assistée Genève* (*Geneva computer-aided composition*).

5. REFERENCES

[1] A. Agostini, E. Daubresse, and D. Ghisi, "cage: une librairie de haut niveau dédiée à la composition assistée par ordinateur dans Max," in *to appear*, 2014.

[2] A. Agostini and D. Ghisi, "*bach*: an environment for computer-aided composition in Max," in *Proceedings of the International Computer Music Conference (ICMC 2012)*, Ljubljana, Slovenia, 2012, pp. 373–378.

[3] ——, "Real-time computer-aided composition with *bach*," *Contemporary Music Review*, no. 32 (1), pp. 41–48, 2013.

[4] M. Malt and J. B. Schilingi, "Profile - libreria per il controllo del profilo melodico per Patchwork," in *Proceedings of the XI Colloquio di Informatica Musicale (CIM)*, Bologna, Italia, 1995, pp. 237–238.

[5] J. Fineberg, "Esquisse - library-reference manual (code de Tristan Murail, J. Duthen and C. Rueda)," Ircam, Paris, 1993.

[6] R. Hirs and B. G. editors, *Contemporary compositional techniques and OpenMusic*. Delatour/Ircam, 2009.

[7] A. Cont, "Modeling Musical Anticipation," Ph.D. dissertation, University of Paris 6 and University of California in San Diego, 2008.

[8] M. Puckette, "A divide between 'compositional' and 'performative' aspects of Pd," in *Proceedings of the First Internation Pd Convention*, Graz, Austria, 2004.

[9] M. Müller, *Information Retrieval for Music and Motion*. Springer Verlag, 2007.

[10] C. Harte, M. S, and M. Gasser, "Detecting harmonic change in musical audio," in *In Proceedings of Audio and Music Computing for Multimedia Workshop*, 2006.

[11] M. Wright, R. Dudas, S. Khoury, R. Wang, and D. Zicarelli, "Supporting the sound description interchange format in the max/msp environment," in *Proceedings of the International Computer Music Conference*, 1999.

[12] M. Wright, A. Chaudhary, A. Freed, S. Khoury, and D. Wessel, "Audio applications of the sound description interchange format standard," in *Proceedings of the Audio Engineering Society 107th Convention*, 1999.