Somax 2: A Real-time Framework for Human-Machine Improvisation

Joakim Borg

19 December 2019

Abstract

In this paper a modular framework for real-time, human-machine improvisation is presented. The framework improvises over an audio or MIDI corpus by recombining its content, while listening and adapting to incoming audio or MIDI signals. The adaptation model consists of multiple layers in a tree-like structure, where each layer listens to a specific parameter (hard-filtering) and the matches in each layer are merged and scaled (soft-filtering) based on further parameters. The structure of the tree and choice of parameters is fully customisable by the user through a modular front-end written in MaxMSP.

1 Introduction

Over the past two decades, several real-time systems for machine improvisation that, to some extent, interacts or relates to a human musician have been developed. The term machine improvisation stems from the fact that the system has some sort of musical "understanding" of the content it is performing and that the output is being generated on the fly.

Among the more impactful examples of such a system is OMax [2], [3], which listens to an input, analyses its content and then improvises over it by recombining its content while maintaining its internal structure. Several systems for human-machine improvisation have been developed that to some extent stems from OMax. Among these are Somax [5], [6], which adds the concept of reactivity to the OMax model, allowing the system not only to draw its material from an input, but to react to a musician in real-time, hence creating a situation of co-improvisation. Another system, Improtek [10], [11], adds the concept of temporal scenarios and contextual awareness to the model. Recently, the DYCI2 project [12] was designed with the intention to merge these three systems into a single framework, but several other systems that to some extent stem from OMax have been developed over the years as well, for example the PyOracle [14] and Mimi [9].

In this paper, an updated implementation of the Somax system is presented. Here, the original architecture of the system has been modularised into a set of building blocks, allowing the user to construct a custom model for listening and reacting and even implement new behaviour. Section 2 outlines the theoretical framework of the system and largely reiterates the content of [6], but with a higher focus on implementation details. Section 3 presents the new architecture, which consists of a server written in Python and a set of MaxMSP objects used to construct a front-end, where the Python server and MaxMSP front-end are communicating with each other over the OSC-protocol [16].

2 The Somax Model

The architecture of Somax consists of three main components: a corpus or data model, created from one or multiple audio and/or midi files, from which the output will be generated, a set of listening modules, which analyses and mixes the input from one or multiple audio/and or midi sources, and a set of matching modules, that determines how the output generated from the corpus will be matched to the input.

2.1 The Corpus

The corpus is the data model from where Somax extracts its output. A corpus $\mathcal{C}$ is constructed from a set of (by the user provided) audio and/or midi files by segmenting the content of the file(s) along the time axis into a set of slices $\left\{\mathcal{S}^{(1)}, \dots \mathcal{S}^{(U)}\right\}$, where for midi slicing is done at each note-on (see figure 1), and for audio the slicing is done at each beat, where the positions of the beats are estimated with the dynamic programming approach described in [8]. For each slice, the onset time, duration, tempo, pitch and soft harmonic context is calculated and stored in the corpus.

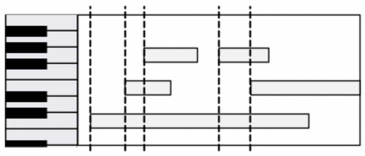

Figure 1: Example of temporal segmentation of midi notes into slices.

The soft harmonic context is computed by calculating the chromagram of the file and applying a low-pass filter along the time-axis, so that notes from each slice maintain a degree of impact on the consecutive slices. For audio files, a constant-Q chromagram $\boldsymbol{C}' = [\boldsymbol{c}'_0, \dots, \boldsymbol{c}'_{N-1}] \in \mathbb{R}^{12\times N}$ is calculated with a hop length of γ seconds and then filtered with a one-zero filter so that the soft harmonic context $\boldsymbol{C} = [\boldsymbol{c}_0, \dots, \boldsymbol{c}_{N-1}]$ is given by

$$\boldsymbol{c}_n = \frac{1-\gamma}{\tau}\boldsymbol{c}'_{n-1} + \frac{\gamma}{\tau}\boldsymbol{c}'_n \qquad (1)$$

where τ is the decay rate of the soft harmonic context.

For a midi corpus consisting of Q midi notes, a pseudo-chromagram is computed. Firstly, an amplitude (or score) $h_0^{(q)}[t]$ is computed for each midi note $\mathcal{M}^{(q)}$, with note number $m_0^{(q)} \in [0, 127]$, note-on at time $t_{\text{on}}^{(q)}$ and note-off at time $t_{\text{off}}^{(q)}$:

$$h_0^{(q)}[t] = \begin{cases} 1 - e^{-\frac{t-t_{\text{on}}^{(q)}}{\tau_a}} & \text{if } t \in \left[t_{\text{on}}^{(q)}, t_{\text{off}}^{(q)}\right] \\ h_0^{(q)}\left[t_{\text{off}}^{(q)}\right] e^{-\frac{t-t_{\text{off}}^{(q)}}{\tau_r}} & \text{if } t \in \left(t_{\text{off}}^{(q)}, t_{\text{end}}^{(q)}\right] \\ 0 & \text{otherwise,} \end{cases} \qquad (2)$$

i.e. as an attack-release envelope over time $t \in \left[t_{\text{on}}^{(q)}, t_{\text{end}}^{(q)}\right]$, where τ_a and τ_r are parameters controlling the slope of the attack and release respectively and $t_{\text{end}}^{(q)}$ is the end of the time interval from the note-off during which the note still influences the harmony, given by

$$t_{\text{end}}^{(q)} = t_{\text{off}}^{(q)} - \tau_r \ln \frac{0.05}{h_0^{(q)}\left[t_{\text{off}}^{(q)}\right]}. \qquad (3)$$

For each note $\mathcal{M}^{(q)}$, K harmonics are created with amplitudes

$$h_k^{(q)}[t] = 0.5^k h_0^{(q)}[t] \qquad k = 1, \dots, K \qquad (4)$$

and note numbers

$$m_k^{(q)} = m_0^{(q)} + \text{round}\left(12 \log_2 k\right). \qquad (5)$$

A MIDI pitch matrix $\boldsymbol{P} \in \mathbb{R}^{128\times N}$ is created and iteratively updated for each note $\mathcal{M}^{(q)}$ by inserting the amplitudes $h_k^{(q)}[t]$ as rows in $\boldsymbol{P}$, i.e.

$$(\boldsymbol{P})_{m_k^{(q)},n} := \max\left((\boldsymbol{P})_{m_k^{(q)},n}, h_k^{(q)}[n]\right) \qquad (6)$$
$$k = 0, \dots, K, \quad n = 0, \dots N-1, \quad q = 1, \dots, Q.$$

Finally, the chromagram $\boldsymbol{C} \in \mathbb{R}^{12\times N}$ is computed by summing all rows corresponding to each pitch class, i.e.

$$\boldsymbol{C} = \mathcal{I}\boldsymbol{P}, \tag{7}$$

where $\mathcal{I} \in \mathbb{Z}^{12\times 128}$ and

$$(\mathcal{I})_{i,j} = \begin{cases} 1 & \text{if } j \equiv_{12} i \\ 0 & \text{otherwise.} \end{cases} \tag{8}$$

For each slice $\mathcal{S}^{(u)}$ in the corpus, its soft harmonic context $\boldsymbol{c}^{(u)}$ is set to the column of $\boldsymbol{C}$ corresponding to the time of the note-on (in the case of midi) or to the element-wise mean of the columns corresponding to the time interval of the slice (in the case of audio). Each slice is assigned a chroma label $\theta_c^{(u)} \in [0, 121]$ by comparing it with a pre-labelled set of chroma vectors (see [6] for details on the labelling), where the label of $\mathcal{S}^{(u)}$ is given by the label of the vector in the set that minimises the euclidean distance to $\boldsymbol{c}^{(u)}$.

Finally, the slice is assigned a pitch label $\theta_p^{(u)}$. For audio corpora, $\theta_p^{(u)} \in [0, 11]$ and

$$\theta_p^{(u)} = \text{argmax}\left(\boldsymbol{c}^{(u)}\right). \tag{9}$$

For MIDI corpora, $\theta_p^{(u)} \in [0, 140]$ where the range $[0, 127]$ correspond to MIDI note numbers, the range $[128, 139]$ correspond to virtual fundamentals [15] of pitch classes $[0, 11]$ and 140 correspond to silence. The label is determined by the number of notes in the slice. If the slice contains zero notes, it will be assigned the label 140. If the slice contains a single note, $\theta_p^{(u)} = m$, where $m \in [0, 127]$ is the note number. If the slice contains multiple note, the virtual fundamental will be used. It is also possible for the user to bypass the virtual fundamental and use the top note in the slice as pitch label.

2.2 Listening

The listening modules analyses one or multiple incoming MIDI and/or audio signals in real-time and uses the results of the analysis to determine which slice of the corpus to output at a given time. The incoming signals are analysed with respect to pitch, chroma, onset and tempo.

For pitch, the procedure is identical to above for MIDI signals, but uses the Yin algorithm [7] for audio, as it assumes each incoming signal is monophonic. The generated analysis will be segmented into discrete pitch labels when the quality of the estimation is above a certain threshold. The procedure of estimating chroma is identical to the soft harmonic context outlined in section 2.1. Onsets are detected from either the Yin onsets or `bonk` [13] for audio, while for MIDI each note-on triggers a new onset. Finally, tempo is computed by the beat tracking module described in [4].

2.3 Mapping Listening to the Corpus

The mapping is done by matching the incoming labels from the listening modules to corresponding labels in the corpus. The matching model consists of multiple layers, where each layer matches a single label type (for example chroma or pitch) to corresponding labels in the corpus. At each match in the model, a peak is generated with a temporal position corresponding to the relative time of the slice in the corpus. The peaks of each layer are scaled, weighted and finally merged together based on parameters specified by the user, where the merged peaks are used to determine which slice in the corpus to output. A simplified description of the entire procedure is given below.

(1) A corpus $\mathcal{C}$ is constructed according to the procedure described in section 2.1. The corpus has a duration of $\Xi \in \mathbb{R}_+$ beats and consists of $U \in \mathbb{Z}_+$ slices, where each slice $\mathcal{S}^{(u)}$, $u = 1, \ldots, U$ is assigned a temporal position $\xi^{(u)} \in [0, \Xi)$.

(2) While the system is running, an internal scheduler with beat $t \in \mathbb{R}_+$ is continuously updated based on a tempo set either by the user or by the beat tracker. Note that there is no relation between two temporal positions t (scheduler time domain) and ξ (corpus time domain), but the corpus time domain is scaled by the tempo of the scheduler so that $\Delta t = \Delta\xi$ for an elapsed interval Δt. In both cases, an interval of $\Delta t = \Delta\xi = 1$ correspond to one beat.

(3) When a label θ is received from the listener at scheduler time t, each layer will attempt to match θ to its data model, should the type (for example pitch or chroma) correspond to the type of that layer, and

at each match generate an influence λ. The data model used is an n-gram, matching the labels of the n previous influences to corresponding sequences of labels in $\mathcal{C}$, returning the last slice $\mathcal{S}^{(u)}$ of the sequence matched. For more details on the n-gram implementation, refer to [6].

(4) In each layer, at time $\xi^{(u)}$ of each matched influence $\mathcal{S}^{(u)}$, a peak with coordinates $\left(\xi^{(u)}, \alpha\right)$ is created and inserted, where the amplitude $\alpha = 1$. Any existing peak from previous influences λ_{prev} are at this point decayed and shifted with an interval Δt corresponding to the elapsed scheduler time since last influence, so that

$$\alpha_{\lambda_{\text{prev}}} := \alpha_{\lambda_{\text{prev}}} e^{-\Delta t/\tau} \tag{10}$$

$$\xi_{\lambda_{\text{prev}}} := \xi_{\lambda_{\text{prev}}} + \Delta t, \tag{11}$$

where τ is set by the user to control the slope of the decay. If any previous peak occur at the same time as an inserted peak, these will be merged into a single peak with amplitude

$$\alpha := \alpha + \alpha_{\lambda_{\text{prev}}}. \tag{12}$$

This behaviour ensures that previous influences maintains an impact on the output for a certain amount of time and that consecutive matches in the same region accumulates into larger peaks, hence enhancing the otherwise rather primitive matching model of the n-gram.

(5) The peaks from each layer are merged into a single set of peaks, where once again peaks occurring simultaneously are summed together so that influences occurring in multiple layers result in larger peaks than those that occur in only a single layer, and all peaks in each layer are scaled according to a (user-controlled) weight specific for each layer, effectively allowing the user to control the relative balance between different layers, for example between harmonic and melodic influences.

(6) The merged peaks are further scaled according to a set of user-defined functions. An example of such a function is the phase adjustment function Φ, defined as

$$\Phi(\alpha, t, \xi) = \alpha \cdot \exp\left[\cos\left(2\pi\left(t - \xi\right)\right) - 1\right] \tag{13}$$

where α is the amplitude of the peak, t the scheduler time when the function is called and ξ the temporal position in the corpus time domain of the peak. As this function is periodic over the fractional part of the differences between the time domains, i.e. $\{t - \xi\} \in [0, 1]$, and the length of a beat correspond to a duration of 1, this de-emphasises peaks occurring at a different subdivision level (or phase) of the beat than the scheduler's current beat.

(7) Finally, a peak is selected and the content of the slice corresponding to the largest peak is output. By default, the slice $\mathcal{S}^{(u)}$ whose time $\xi^{(u)}$ minimises the distance to the largest peak is used to generate the output, i.e.

$$\min_{u \in [1,U]} \left|\xi^{(u)} - \xi_{\max}\right|, \tag{14}$$

where $\xi_{\max}$ is the temporal position of the largest peak. If $\mathcal{S}^{(u)}$ is an audio slice, the original audio file is played from the time corresponding to the onset $\xi^{(u)}$ to its end $\xi^{(u+1)}$, time-stretched with respect to the scheduler's time. for a MIDI slice, any note-on or note-off occurring within the slice will be played at its original position (scaled with respect to the scheduler's tempo). As a slice may contain multiple notes and some of these may be held from a previous slice (for example the D in the third slice of figure 1) or held into the next slice (same D but from the perspective of the second slice), this will be accounted for by triggering note-offs for any notes held from a previously played slice and triggering note-ons for any note occurring in the slice but not held by the previously played slice. In other words, each slice $\mathcal{S}^{(u)}$ has four types of note information: a set of note-ons, $\mathcal{N}_{\text{on}}^{(u)}$ a set of note-offs $\mathcal{N}_{\text{off}}^{(u)}$ a set of notes held from the (in the corpus) previous slice $\mathcal{S}^{(u-1)}$: $\mathcal{N}_{\text{from}}^{(u)}$ and a set of notes held into (in the corpus) the next slice $\mathcal{S}^{(u+1)}$: $\mathcal{N}_{\text{to}}^{(u)}$. When the system is transitioning from slice $\mathcal{S}^{(v)}$ to slice $\mathcal{S}^{(w)}$, $v, w \in [1, U]$, the output can be described by

$$\text{note-ons} = \mathcal{N}_{\text{on}}^{(w)} \cup \left(N_{\text{to}}^{(w)} \setminus N_{\text{from}}^{(v)}\right) \tag{15}$$

$$\text{note-offs} = \mathcal{N}_{\text{off}}^{(w)} \cup \left(N_{\text{from}}^{(v)} \setminus N_{\text{to}}^{(w)}\right) \tag{16}$$

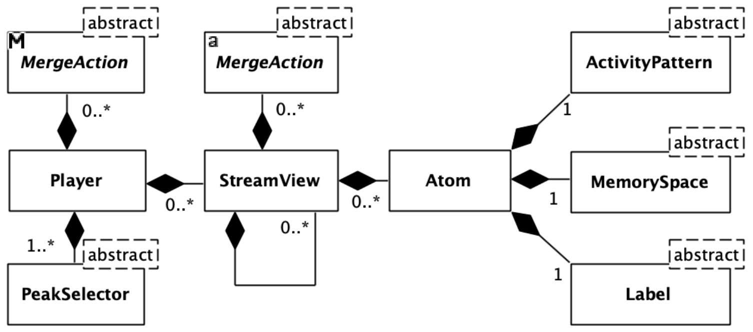

Figure 2: Simplified class diagram of the Python architecture.

Note that steps 2-7 describes the procedure to output a single slice based on an input, but this procedure is repeated as soon as a slice is finished playing, which in most cases means several times per second, hence creating a recombined output of the original corpus $\mathcal{C}$ based on where the peaks generated from the input occur. Also note that in the actual system, steps 2-4 (input, i.e. influencing and peak generation) are decoupled from steps 5-7 (output, i.e. selecting and outputting slices) to allow multiple modes of interaction between input and output.

3 Somax as a Framework

While the system described in section 2 largely mimics the Somax implementation described in [6], the main contribution in this article is the redesign of the architecture from a fixed system into a toolbox of objects and functions that can be combined dynamically. The toolbox consists of two main components: (a) a server, written in Python, handling the offline construction of the corpus described in section 2.1 as well as real-time mapping described in section 2.3, and (b) one or multiple clients, written in MaxMSP, handling the real-time analysis described in section 2.2 as well as the final audio/MIDI output. Each of those have a highly modular design, allowing the user to freely recombine the modules within a fixed architecture and easily extend them to implement new behaviour.

3.1 Python Architecture

Figure 2 shows a simplified view of the class architecture of the Python implementation. The core of the architecture is the `Player`, which contains any number of `StreamViews`, which in turn contain any number of `StreamViews` and any number of `Atoms`. This corresponds to the multiple layers referred to in section 2.3, but should rather be described as a tree (with `Player` as the root, `StreamView` as branches and `Atom` as leaves). Around them are five abstract classes, (`MergeAction` is duplicated in the figure), each of those roughly corresponding to the behaviour described in each step in section 2.2 and open for the user to extend to create new behaviour. At runtime, the user can specify the structure of the three and which instances of the abstract classes to use at each depth. This section will once again describe the steps 3-7 from section 2.3, but in terms of the architecture.

(3.1) When a piece of label information is received, each implemented `Label` class will attempt to categorise it accordingly. There are currently three implemented labels: `MelodicLabel`, corresponding to labels $\theta_p \in [0, 140]$, `PitchClassLabel`, corresponding to labels $\theta_p \in [0, 11]$ and `HarmonicLabel`, corresponding to labels $\theta_c \in [0, 121]$. The `Label` class could also be extended to implement matching (hard-filtering) of other discretisable parameters, for example instrumentation (which could be discretised as MIDI channel or for audio as a MFCC-based labelling algorithm), articulation, dynamics, etc. The user can choose to influence a specific `Atom` in the tree or (by

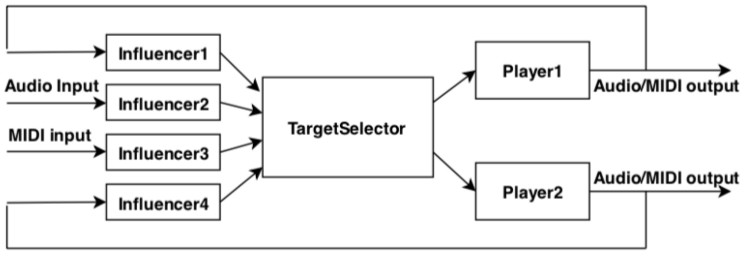

Figure 3: Simplified diagram of the objects used to model the previous version of Somax.

default) influence all `Atoms` in the tree whose `Label` class correspond to the incoming label.

(3.2) The incoming label is matched, together with previously received labels, to the data model of the `Atom` implemented in the `MemorySpace` class, forwarding a list of matched slices to the `ActivityPattern` of the atom. Currently, the only data model implemented is the `NGramMemorySpace` class outlined in section 2.3, but more sophisticated models such as suffix trees, the factor oracle [1] or even deep-learning based models could be used to extend the framework.

(4) The `ActivityPattern` determines how slices matched from the `MemorySpace` of each atom are converted into peaks and how the peaks are shifted (corresponding to equation 11) and decayed (equation 10).

(5, 6) At each `StreamView` (each branch in the tree), the peaks are merged and scaled (soft-filtered) according to the `MergeActions` declared in the `StreamView`. The merging is done with the `DistanceMergeAction` and two other classes have been implemented for scaling: `PhaseMergeAction`, described in equation 13, and `StateMergeAction`, which slightly boosts peaks close (in the corpus time domain) to the temporal position of the previously output slice. While not yet implemented, technically the `MergeAction` class could allow any type of parametric soft-filtering of peaks, for example on velocity, duration, register or note density.

(7) Finally, the `PeakSelector` class is implemented to allow the user to choose which slice to output from the final set of peaks. Currently, only the `MaxPeakSelector`, which selects the slice closest to the largest peak (equation 14) has been implemented.

The general idea is that the combination of multiple layers of hard-filtering slices to input data, in combination with soft-filtering of the results based a set of musical parameters, should allow the user to create a specific model for matching, as well as give the user detailed real-time control of the parameters of the model to allow the performer(s) expressive control of the generated output. While only a few filters (`MergeActions`) have been implemented so far, the core architecture parses any information needed with python's `inspect` module, so that implementing a custom `MergeAction` can be done without any modification to the original code, and its parameters will be available in the MaxMSP interface.

3.2 MaxMSP Architecture

The MaxMSP architecture consists of a set of abstractions for communicating with the Python server as well as analysing incoming audio and midi signals, as described in section 2.2. Each abstraction has a graphical user interface implemented, but can also be used with a custom interface and controlled with normal Max messages. The implemented abstractions are: `somax.audioinfluencer`, which analyses an incoming audio signal with respect to pitch, chroma and onsets, `somax.midiinfluencer`, which similarly analyses an incoming midi signal with respect to these parameters, `somax.player` which directly corresponds to a `Player` object on the server (see figure 2), `somax.targetselector` which is used to route and mix messages from the influencers to a corresponding player (or a specific `StreamView` or `Atom` in the player's tree), `somax.beattracker`, which implements the beat tracker described in [4] to control the scheduler's tempo, and finally `somax.paramselector`, which exposes all parameters of each player on the server, and is used to modify these in real-time.

These objects are designed to allow the user to quickly create an interface matching the wanted the interaction between audio sources, midi sources and players. In the original Somax model (again, see [6]), the interface consisted of one input (audio or midi) and two players, where the players could be influenced by the input as well as by each other. A similar structure with the new architecture can be seen in figure 3. This structure has been implemented as part of the new toolbox for evaluation and demo purposes.

4 Conclusion

This paper presented a modular framework for real-time human-machine improvisation. Based on the reactive system in [6], the new framework allows the user to create a highly customisable model for listening and reacting to human-controlled audio or MIDI input in multiple layers.

All components of the framework are modular and extendable so that the user can implement new modes of parameter matching, either through soft- or hard-filtering, as well as control other aspects of the model. While implementing new parameters is done in the Python code base, all other aspect of the system, including customising the model and controlling any parameter, can be done directly through the MaxMSP front-end, which also is highly modular, thus requiring no other programming skills.

References

[1] Cyril Allauzen, Maxime Crochemore, and Mathieu Raffinot. Factor oracle: A new structure for pattern matching. In *International Conference on Current Trends in Theory and Practice of Computer Science*, pages 295–310. Springer, 1999.

[2] Gérard Assayag, Georges Bloch, Marc Chemillier, Arshia Cont, and Shlomo Dubnov. Omax brothers: a dynamic topology of agents for improvization learning. In *Proceedings of the 1st ACM workshop on Audio and music computing multimedia*, pages 125–132. ACM, 2006.

[3] Gérard Assayag and Shlomo Dubnov. Using factor oracles for machine improvisation. *Soft Computing*, 8(9):604–610, 2004.

[4] Laurent Bonnasse-Gahot. Donner à omax le sens du rythme: vers une improvisation plus riche avec la machine. *École des Hautes Études en sciences sociales, Tech. Rep*, 2010.

[5] Laurent Bonnasse-Gahot. Prototype de logiciel d'harmonisation/arrangement à la volée: Somax v0. 2012.

[6] Laurent Bonnasse-Gahot. An update on the somax project. *Ircam-STMS, Tech. Rep*, 2014.

[7] Alain De Cheveigné and Hideki Kawahara. Yin, a fundamental frequency estimator for speech and music. *The Journal of the Acoustical Society of America*, 111(4):1917–1930, 2002.

[8] Daniel PW Ellis. Beat tracking by dynamic programming. *Journal of New Music Research*, 36(1):51–60, 2007.

[9] Alexandre RJ François, Elaine Chew, and Dennis Thurmond. Performer-centered visual feedback for human-machine improvisation. *Computers in Entertainment (CIE)*, 9(3):13, 2011.

[10] Jérôme Nika and Marc Chemillier. Improtek: integrating harmonic controls into improvisation in the filiation of omax. In *International computer music conference (ICMC)*, pages 180–187, 2012.

[11] Jérôme Nika, Marc Chemillier, and Gérard Assayag. Improtek: introducing scenarios into human-computer music improvisation. *Computers in Entertainment (CIE)*, 14(2):4, 2016.

[12] Jérôme Nika, Ken Déguernel, Axel Chemla, Emmanuel Vincent, Gérard Assayag, et al. Dyci2 agents: merging the" free"," reactive", and" scenario-based" music generation paradigms. 2017.

[13] Miller S Puckette, Miller S Puckette Ucsd, Theodore Apel, et al. Real-time audio analysis tools for pd and msp. 1998.

[14] Greg Surges and Shlomo Dubnov. Feature selection and composition using pyoracle. In *Ninth artificial intelligence and interactive digital entertainment conference*, 2013.

[15] Ernst Terhardt. Calculating virtual pitch. *Hearing research*, 1(2):155–182, 1979.

[16] Matthew Wright. Open sound control: an enabling technology for musical networking. *Organised Sound*, 10(3):193–200, 2005.