TITLE TODO

Subtitle todo

Joakim Borg, Gérard Assayag, + more?

..

Abstract

abstract abstract abstract todo (or maybe skip?)

1 Transforms

A fundamental problem with the model used in Somax is the fact that the quality of the output heavily relies on how well the influence (input) matches the corpus. This lies in the nature of its corpus-based approach - if the n-gram model is unable to find any slices matching a given series of influence labels, the output cannot be generated in relation to the given input. For example, if a corpus is using only the pitches of the D major scale while the influence is strictly limited to the pitches of the Db major scale, finding harmonic or melodic matches between the two will be impossible as they have no tones in common. What they do have in common is however intervallic relationships between the pitches, and if we could use this in the matching algorithm, the number of matches would increase without reducing the quality of the individual matches. For this purpose, the concept of transforms was introduced.

1.1 Definition

A transform $\mathcal{T}$ is a set of functions $\Lambda\colon \mathbb{V} \to \mathbb{V}$ for an arbitrary domain $\mathbb{V}$ that fulfil a number of conditions. More specifically, $\mathcal{T} = \{\Lambda_{\mathcal{S}}, \Lambda_{\theta 1}, \dots, \Lambda_{\theta q}\}$ where

(i) $\Lambda_{\mathcal{S}}$ is defined for any slice $\mathcal{S}_u^{\mathcal{C}}$, $u = 1, \dots U$ given a corpus $\mathcal{C}$ of length U so that the content of $\mathcal{S}_u^{(\mathcal{T})} = \Lambda_{\mathcal{S}}\left(\mathcal{S}_u^{(\mathcal{C})}\right)$ is defined and can be played in real-time,

(ii) Each function $\Lambda_{\theta q}$, $q = 1, \dots, Q$ is defined for one trait $\theta^{(q)}$,

(iii) Each function $\Lambda_{\mathbf{v}}$ has an inverse $\Lambda_{\mathbf{v}}^{-1}$ defined so that for any arbitrary element $\mathbf{v} \in \mathbb{V}$,

$$\Lambda_{\mathbf{v}}^{-1}\left(\Lambda_{\mathbf{v}}\left(\mathbf{v}\right)\right) = \Lambda_{\mathbf{v}}\left(\Lambda_{\mathbf{v}}^{-1}\left(\mathbf{v}\right)\right) = \mathbf{v}. \tag{1}$$

Given a transform $\mathcal{T}$ fulfilling property (i) and (ii), we can transform each slice $\mathcal{S}_u^{(\mathcal{C})}$ in the corpus and given a sequence of influences look for matches in both the original sequence $\left\{\theta_1^{(q}, \dots, \theta_U^{(q)}\right\}$ as well as the transformed sequence $\left\{\Lambda_{\theta q}\left(\theta_1^{(q)}\right), \dots, \Lambda_{\theta q}\left(\theta_U^{(q)}\right)\right\}$ as of property (ii), while ensuring that a slice $\mathcal{S}_u^{(\mathcal{T})}$ can be output (as of property (i)) and matches the influence. The third property exists for performance reasons and will be discussed in the following sections.

1.2 Transposition

Given that all currently implemented classifiers focus on aspects of the pitch domain, either in terms of the pitches of certain notes (Top Note Classifier, Pitch Class Classifier) or chroma (SOM Chroma Classifier, GMM Chroma Classifiers) [7], the only transform implemented so far is the Transposition transform. The intuition behind this transform is simple - given our previous example with a corpus in D and influence in Db, we simply transpose our corpus to Db.

More formally, we define a transposition $\mathcal{T}^{(n)}$ of $n \in [-5, 6]$[1] semitones as the set

$$\mathcal{T}^{(n)} = \left\{\Lambda_{\mathcal{S}}^{(n)}, \Lambda_{\theta Pt}^{(n)}, \Lambda_{\theta c}^{(n)}\right\} \tag{2}$$

where

(i) The function $\Lambda_{\mathcal{S}}^{(n)}\left(\mathcal{S}_u^{(\mathcal{C})}\right)$ is for MIDI[2] corpora defined as the set of notes $\mathcal{N}_u^{(\mathcal{T}n)} = \left\{m_1^{(\mathcal{T}n)}, \dots, m_k^{(\mathcal{T}n)}\right\}$ (per definitions in section 3.4.2 of [7]) where $m_k^{(\mathcal{T}n)}$ is the k:th note in $\mathcal{N}_u^{(\mathcal{C})}$ with its pitch altered by n, i.e.

$$m_k^{(\mathcal{T}n)}.\texttt{pitch} = m_k.\texttt{pitch} + n, \quad \forall m_k \in \mathcal{S}_u^{(\mathcal{C})}. \tag{3}$$

(ii.1) The function $\Lambda_{\theta Pt}^{(n)}$ given a pitch $\theta^{(Pt)} \in \mathbb{Z}_{[0,127]}$ is defined as the pitch shifted by n, i.e.

$$\Lambda_{\theta Pt}^{(n)}\left(\theta^{(Pt)}\right) = \theta^{(Pt)} + n \tag{4}$$

(ii.2) The function $\Lambda_{\theta c}^{(n)}$ given a chroma vector $\boldsymbol{\theta}^{(c)} \in \mathbb{R}_{[0,\infty]}^{12}$ is defined as the chroma vector rotated by n, i.e.

$$\Lambda_{\theta c}^{(n)}\left(\theta^{(c)}\right) = \mathbf{M}^{(n)}\boldsymbol{\theta}^{(c)}, \tag{5}$$

[1]The domain $[-5, 6]$ was chosen to allow transpositions to all 12 pitch classes while ensuring that the transposed content is as close to the original timbre of the input as possible

[2]While audio corpora are not yet implemented in the current version of Somax, the same behaviour could be implemented for audio corpora by pitch shifting the temporal interval of the original audio file by a value corresponding to the number of semitones n.

where the rotation matrix $\boldsymbol{M}^{(n)} \in \mathbb{Z}_{[0,1]}^{12\times 12}$ is given by

$$\left(\mathbf{M}^{(n)}\right)_{i+1,j+1} = \begin{cases} 1 & \text{if } i = j+n \mod 12 \\ 0 & \text{otherwise} \end{cases}, \quad i,j = 0,\dots,11. \tag{6}$$

(iii) Finally, the inverses are defined identically to equations 3-6 but with all n-terms substituted by $-n$.

1.3 Procedure

The concept of transforms is by no means a novelty in Somax. In fact, it was already present in an early beta version of Somax 2 released back in 2018. The main problem of this version was however a huge decrease in performance, increasing the latency of the system by up to several hundreds of milliseconds (on a 2018 MacBook Pro 13" with a corpus of length $U < 10000$) when using multiple of transforms. This section will outline how the new implementation of transforms has altered the three main processes of the system: (1) constructing and modelling the corpus, (2) influencing and (3) generating output[3], which will be evaluated in terms of performance in section 1.4.

For constructing and modelling the corpus, the implementation of transforms does in fact not introduce any changes. In section 1.2, the procedure was described as a transposition of the entire corpus, which would mean clustering, classification and constructing N parallel models of the corpus given N transforms. The problem with this approach is that it would increase load time significantly and modifying parameters of the clustering/classification or modelling modules would become too computationally heavy to handle at runtime. This approach would also lead to a significant increase in runtime computation time as the size of the corpus would effectively be N times as large. Instead, we're utilizing property (iii) to inverse-transform the incoming influences $\theta^{(\mathcal{K})}$ for each transform before classification, i.e. that for a transform $\mathcal{T}^{(n)}$ and a slice $\mathcal{S}_u^{(\mathcal{C})}$ containing the trait $\theta_u^{(\mathcal{S})}$ we're utilizing the fact that if

$$\Lambda_\theta^{(n)}\left(\theta_u^{(\mathcal{S})}\right) = \theta^{(\mathcal{K})} \tag{7}$$

then

$$\left(\Lambda_\theta^{(n)}\right)^{-1}\left(\theta^{(\mathcal{K})}\right) = \theta_u^{(\mathcal{S})}. \tag{8}$$

Then for each inverse-transformed influence, we're computing the same steps as in section 2.3 of [7] for each of the n inverse-transformed influences, with the only new addition being that a peak $\boldsymbol{p}$ which previously was defined as

$$\boldsymbol{p} = \begin{bmatrix} t^{(\mathcal{C})} \\ y \end{bmatrix} \tag{9}$$

[3]For a thorough description of each of these sections, refer to chapter 2 in [7].

now contains additional information about which transform was applied,[4] i.e.7

$$\boldsymbol{p} = \begin{bmatrix} t^{(\mathcal{C})} \\ y \\ \mathcal{T}^{(n)} \end{bmatrix}. \tag{10}$$

Finally, for the process of merging peaks and generating output, the former process is identical to the description in section 2.4.2 in [7] except that the concatenated peak matrix $\boldsymbol{P}_w$ is split into n matrixes - one per transform - so that peaks corresponding to similar corpus time $t^{(\mathcal{C})}$ but different transforms are not merged with each other. The latter process, the generation of output, contains an additional step, that of transforming the selected output slice $\mathcal{S}_w^{(\mathcal{Y})}$ in accordance with its corresponding peak's applied transform, i.e. the final output slice $\mathcal{S}_w^{(\mathcal{T}n)}$ is defined as

$$\mathcal{S}_w^{(\mathcal{T}n)} = \Lambda_{\mathcal{S}}^{(n)} \left(\mathcal{S}_w^{(\mathcal{Y})} \right) \tag{11}$$

where $\Lambda_{\mathcal{S}}^{(n)}$ is the function corresponding to the transform of the selected peak $\hat{\boldsymbol{p}}_w$ in the case where such a peak exist. If no peak exists, the default output is the slice following the previously output slice with the same transform applied, i.e. given the previous output[5]

$$\mathcal{S}_{w-1}^{(\mathcal{T}n)} = \Lambda_{\mathcal{S}}^{(n)} \left(\mathcal{S}_{w-1}^{(\mathcal{Y})} \right) \quad \text{where} \quad \mathcal{S}_{w-1}^{(\mathcal{Y})} = \mathcal{S}_u^{(\mathcal{C})} \tag{12}$$

for some $u \in [1, U]$, we get

$$\mathcal{S}_w^{(\mathcal{T}n)} = \Lambda_{\mathcal{S}}^{(n)} \left(\mathcal{S}_{u+1}^{(\mathcal{C})} \right). \tag{13}$$

1.4 Runtime Performance

To evaluate the increase in computation time (i.e. runtime latency) caused by the addition of transforms, a sequence of integration tests were carried out. In these tests, the computational cost of the two main runtime processes, influence and generate, were evaluated separately on a corpus consisting of $U = 7966$ slices under four different conditions: no additional transform applied (formally one transform of 0 semitones), three additional transforms applied, six additional transforms applied and 11 additional transforms applied. The same corpus was used both for the model and the influence, i.e.

$$\mathcal{S}_u^{(\mathcal{C})} = \mathcal{S}_w^{(\mathcal{K})} \quad w = u = 1, \dots, U, \tag{14}$$

and iterated over linearly 10 times, in total yielding 79660 influence-generate steps, but at each step in the iteration, $p = 10^0, 10^1, \dots, 10^4$ additional peaks

[4] For practical reasons, this transform is denoted $\mathcal{T}$ here but in the actual implementation it is stored as an integer reference, which means that it indeed can be stored in a vector.

[5] see section 2.4.4 in [7]. Also see section 2 of this report for a further update on the output selection process.

with random scores, times and transforms were inserted to evaluate how the system performs under different conditions. The entire process was evaluated on a MacBook Pro 13-inch 2018 with a 2.3 GHz Intel Core i5 processor running MacOS 10.15.

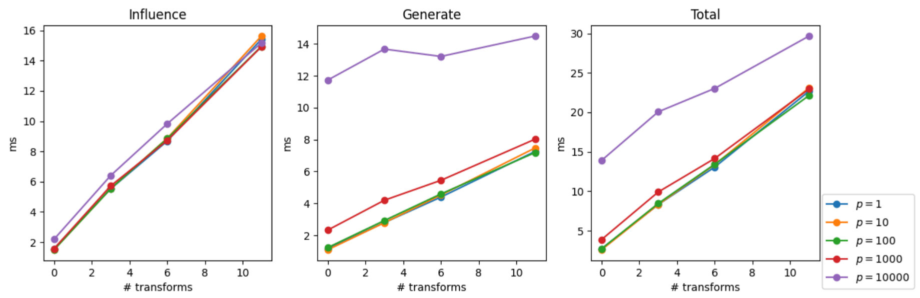

Figure 1: Average computational cost with respect to number of transforms.

As depicted in figure 1, the average computational cost increases linearly with the number of transforms and remains fairly constant independently of the number of added peaks.[6] For the generation process we also see a linear increase with the number of transforms, but the increase with relation to number of transforms is much less steep, while the computational cost is determined by the number of peaks to a much higher degree. In the third figure, we see the average cost of an entire influence-generation cycle, i.e. the time elapsed from when a performer presses a key to when the system responds. Without going too deep into psychoacoustics we can assume that most performers would consider a latency of 3.897 ms (0 additional transforms, $p = 1000$) as close to imperceivable, a latency of 9.911 ms (3 additional transforms, $p = 1000$) most likely acceptable, but above that, we run into issues of perceivable latencies, for example with 6 additional transforms at $p = 1000$ (14.16 ms), or even with 0 additional transforms at $p = 10000$ (13.91 ms).

The relation between computation time of the influence process and generate process are also important to take into account when further expanding the system. With the current relationship between estimated number of peaks and the number of classifiers, this implementation of transforms is ideal for runtime purposes, but should the system in the future move towards a higher degree of parallel classifiers, it's important to take this into account and possibly find other strategies for handling transforms in relation to classification.

2 Additional Control Parameters

A number of new features and other architectural improvements have been added to the main runtime framework to improve the quality of the output

[6]The cost increases slightly with a large number of peaks, likely due to reallocation of memory being slightly more expensive in the final concatenation of peaks, but this cost does not have a significant impact on the overall result

as well as give the user a higher degree of parametric control. This chapter will briefly recapitulate the behaviour of the main runtime framework to provide a context for the new features, which will be presented afterwards.

2.1 Recapitulation of the Runtime Architecture

As described in sections 2.2-2.4 in [7], there are three main processes in the Somax system: (i) building and modelling a corpus, (ii) influencing and (iii) generating output.

In (i), a MIDI file is sliced (i.e. segmented into vertical slices $\mathcal{S}_u^{(\mathcal{C})}$, $u = 1, \ldots, U$ into a corpus $\mathcal{C}$ of length U) where each slice is analyzed with respect to a number of traits, and R n-gram-based models $\mathcal{M}^{(r)}$, $r = 1, \ldots, R$ are constructed by clustering and classifying the sequence of slices in R layers, each with respect to a single trait. This step is computed once and not part of the main runtime architecture (though the steps of clustering, classifying and modelling may be recomputed when altering certain parameters during runtime).

While the system is running, the processes of influencing (ii) and generating output (iii) are continuously computed in an interleaved manner. Given input from a continuous MIDI and/or audio stream (e.g. a musician playing in real-time, a pre-recorded audio file, etc.), this stream is sliced (continuously), analyzed with respect to the same traits as above and classified into discrete labels in each of the R layers. By comparing the sequence of most recent labels generated by this process with the n-gram model, a number of matches are found, each match generating a peak $\boldsymbol{p}$, which basically is a coordinate along the temporal axis of the corpus with a certain score corresponding to the quality of the peak (with the new addition of transform, as per equation 10). These peaks are combined with peaks from previous influence, that have been shifted and decayed corresponding to the amount of time that has passed since the previous influence, into a matrix of peaks, $\boldsymbol{P}_w^{(r)}$, for each layer $r = 1, \ldots, R$ at time step w.

The final process is the generation of output (iii), which as mentioned is computed interleaved with the influence process, either directly after each received influence or asynchronous based on its own scheduling system (depending on which mode the user has selected). In this step, the peak matrices from each layer are collected, scaled and merged into a single matrix $\boldsymbol{P}_w$ (at each time step w). After that, the peaks are scaled individually with regards to a number of musical parameters (which was labelled as ≪fuzzy filtering≫ in [7]) and finally, the slice corresponding to the highest peak is selected as output and played accordingly.

This entire procedure is thoroughly explained in [7] and has not been significantly altered apart from what was described in section 1 of this report. However, all of the improvements that will be presented in this section revolve around the two final steps, so these will need some further explanation.

More formally, the individual scaling of a peak matrix $\boldsymbol{P}_w = \begin{bmatrix}\boldsymbol{p}^{(1)} & \ldots & \boldsymbol{p}^{(K)}\end{bmatrix}$,

$\boldsymbol{P}_w \in \mathbb{R}^{3\times K}$ at time step w was in [7] described as a function $\Gamma\colon \mathbb{R}^{3\times K} \to \mathbb{R}^{3\times K}$ with a number of side effects. While this remains true, it is easier to describe it in terms of a scale operation of individual peaks, i.e. $\Gamma\colon \mathbb{R}^3 \to \mathbb{R}^3$ for each peak $\boldsymbol{p}^{(k)} \in \boldsymbol{P}_w$, $\boldsymbol{p}^{(k)} \in \mathbb{R}^3$ for $k = 1, \dots, K$. This function is at time step w defined as

$$\boldsymbol{p}^{(*)} = \Gamma\left(\boldsymbol{p}, t_w^{(\mathcal{Y})}, \mathcal{S}_{\boldsymbol{p}}^{(\mathcal{C})}, \mathcal{H}_w^{(q)}\right), \qquad \forall\, \boldsymbol{p} \in \boldsymbol{P}_w \tag{15}$$

where $t_w^{(\mathcal{Y})} \in \mathbb{R}_+$ denotes the (scheduler's) time at generation time step w, $\mathcal{S}_{\boldsymbol{p}}^{(\mathcal{C})}$ denotes the slice in the corpus corresponding to the time $t^{(\mathcal{Y})}$ of the peak (per equation 10) and $\mathcal{H}_w^{(q)}$ is the history of output slices from the previous $q \in \mathbb{Z}_+$ time steps, i.e.

$$\mathcal{H}_w^{(q)} = \left\{S_{w-1}^{(\mathcal{Y})}, \dots, S_{w-q}^{(\mathcal{Y})}\right\}, \tag{16}$$

where $\mathcal{S}_{w-m}^{(\mathcal{Y})}$ denotes the generated output m time steps ago.

This behaviour was previously handled at each level in the architecture by the `MergeAction`, which handled both merging peaks from multiple layers as well as scaling them individually, but has now been moved to the `ScaleAction` class and is only computed once at the top level (`Player` class). Do note that there are no limitations to the number of scale actions that can be applied, so the final value of each peak $\boldsymbol{p}^{(*)}$ can for J scale actions be defined as

$$\boldsymbol{p}^{(*)} = \left(\Gamma^{(J)} \circ \Gamma^{(J-1)} \circ \cdots \circ \Gamma^{(1)}\right)\left(\boldsymbol{p}, t_w^{(\mathcal{Y})}, \mathcal{S}_{\boldsymbol{p}}^{(\mathcal{C})}, \mathcal{H}_w\right) \tag{17}$$

where $\circ$ denotes function composition. Each of the scale actions described in the following sections are optional and can be added/removed dynamically by the user.

The final step in the entire generation process is to select an output slice $\mathcal{S}_w^{(\mathcal{Y})}$ from the generated and scaled peak matrix $\boldsymbol{P}_w^{(*)}$ at time step w. This was in section 2.4.4 of [7] defined as the slice corresponding to the peak with the highest score (or if no peaks exist, simply the slice in the corpus following the previous output slice), but can be generalized to a selection function $\Omega\colon \mathbb{R}^{3\times K} \to \mathcal{S}$ defined as

$$\mathcal{S}_w^{(\mathcal{Y})} = \Omega\left(\boldsymbol{P}_w^{(*)}, \mathcal{C}, \mathcal{H}_w\right). \tag{18}$$

This will be further elaborated on in section 2.6, where two new functions for output selection are presented.

2.2 Transposition Consistency

One concern related to the introduction of transpositions is that all transforms are given equal probability. In certain (tonal) cases, this may mean that unbiased and too frequent jumping between transpositions will cause issues between the original harmonic function of a particular chord and its harmonic function in the new real-time context. The Transposition Consistency scale action was introduced as an attempt to remedy this by simply scaling all peaks with a user controlled parameter $\gamma \in \mathbb{R}_{[0,1]}$ if the peak's transposition does not correspond to the transposition of the previous output slice, i.e.

$$\boldsymbol{p}^{(*)} = \begin{cases} \boldsymbol{p} & \text{if } \mathcal{T}_{w-1} = \mathcal{T}_{\boldsymbol{p}} \\ \begin{bmatrix} 1 & 0 & 0 \\ 0 & \gamma & 0 \\ 0 & 0 & 1 \end{bmatrix} \boldsymbol{p} & \text{otherwise} \end{cases} \tag{19}$$

where $\mathcal{T}_{w-1}$ denotes the transform of the previous output slice $\mathcal{S}^{(\mathcal{Y})}_{w-1}$ and $\boldsymbol{p}$ is defined (according to equation 10) as

$$\boldsymbol{p} = \begin{bmatrix} t^{(\mathcal{C})}_{\boldsymbol{p}} \\ y_{\boldsymbol{p}} \\ \mathcal{T}_{\boldsymbol{p}} \end{bmatrix}. \tag{20}$$

2.3 Taboo and Auto-Jump

Taboo and auto-jump are two behaviours that existed in earlier versions of Somax but were not originally included in the 2.0 rewrite. With this update, they have been reintroduced as scale actions in the new architecture. Taboo is a simple but efficient way to prevent a player from getting stuck in a short loop by reducing the score $y_{\boldsymbol{p}}$ of all peaks whose temporal coordinates $t^{(\mathcal{C})}_{\boldsymbol{p}}$ corresponds to the $q \in \mathbb{Z}_+$ (user-controlled parameter) most recent output slices to 0, i.e.

$$\boldsymbol{p}^{(*)} = \begin{cases} \begin{bmatrix} t^{(\mathcal{C})}_{\boldsymbol{p}} \\ 0 \\ \mathcal{T}_{\boldsymbol{p}} \end{bmatrix} & \text{if } \mathcal{S}^{(\mathcal{C})}_{\boldsymbol{p}} \in \mathcal{H}^{(q)}_{w} \\ \boldsymbol{p} & \text{otherwise.} \end{cases} \tag{21}$$

The purpose of the auto-jump scale action is to prevent too long sequences of the original corpus to be replicated in the output by gradually increasing the probability of a jump. More precisely, it will gradually decrease the score of peaks corresponding to the in the corpus following slice within a user-defined interval $[a, b], a, b \in \mathbb{Z}_+, b > a$ number of generated output events by a factor $\gamma \in \mathbb{R}_{[0,1]}$ calculated according to the following formula:

$$\gamma = \begin{cases} 1 & \text{if } \tau \leq a \\ 0.5^{\tau - a} & \text{if } a < \tau < b \\ 0 & \text{otherwise,} \end{cases} \tag{22}$$

where τ corresponds to the number of in the corpus consecutive slices computed by algorithm 1, where the function $u\left(\mathcal{S}^{(\mathcal{Y})}\right)$ returns the index $u \in [1, \ldots, U]$ of $\mathcal{S}^{(\mathcal{Y})}$ in its original corpus $\mathcal{C}$. The result of this calculation is only applied to the score of any peak corresponding to the in the corpus next consecutive slice,

Algorithm 1 Computing number of consecutive slices τ

```
$\tau := 0$
$u := u\left(\mathcal{S}_{w-1}^{(\mathcal{Y})}\right)$
$i := 2$
**while** $u - u\left(\mathcal{S}_{w-i}^{(\mathcal{Y})}\right) = 1$ **do**
    $\tau := \tau + 1$
    $u := u\left(\mathcal{S}_{w-i}^{(\mathcal{Y})}\right)$
    $i := i + 1$
**end while**
```

i.e.

$$\boldsymbol{p}^{(*)} = \begin{cases} \begin{bmatrix} t_{\boldsymbol{p}}^{(\mathcal{C})} \\ \gamma y_{\boldsymbol{p}} \\ \mathcal{T}_{\boldsymbol{p}} \end{bmatrix} & \text{if } u\left(\mathcal{S}_{\boldsymbol{p}}^{(\mathcal{C})}\right) = u\left(\mathcal{S}_{w-1}^{(\mathcal{Y})}\right) - 1 \\ \boldsymbol{p} & \text{otherwise.} \end{cases} \tag{23}$$

2.4 Parametric Filters

Somax has up to this point mainly been operating on two musical dimensions when selecting its output: chroma and pitch. In this update, a number of scale actions were introduced to give the user some degree of control over other musical dimensions such as loudness, spectral distribution and various temporal aspects of the generated output. Similarly to other scale actions, all of these filters operate by scaling the score of the peak according to a parameter $\gamma \in \mathbb{R}_{[0,1]}$, i.e.

$$\boldsymbol{p}^{(*)} = \begin{bmatrix} t^{(\mathcal{C})} \\ \gamma y_{\boldsymbol{p}} \\ \mathcal{T}_{\boldsymbol{p}} \end{bmatrix} \tag{24}$$

where γ can be described as a function operating on a trait θ defined in the corpus. More specifically, each of those filters (apart from the last one) take the form of a gaussian defined so that

$$\gamma = \gamma\left(\theta, \bar{\theta}\right) = \exp\left[-\frac{(\theta - \bar{\theta})^2}{2s^2}\right] \tag{25}$$

where $\bar{\theta}$ (unless otherwise specified) defines the desired value for the targeted musical trait and s (in all cases) defines the desired spread of the targeted parameter, effectively controlling how steeply peaks corresponding to slices whose traits fall outside the requested value should be attenuated. Both $\bar{\theta}$ and s are (unless otherwise specified) user-controlled parameters. As Somax cannot construct corpora from audio files at the moment, all of these filters are only defined for corpora based on MIDI files.[7] Note that none of these scale actions will remove any peaks no matter how far away their corresponding traits fall from the requested value, they will at most drastically reduce their probability. The idea behind this is to to give the user control over the overall direction, where occasional fluctuations occur, ideally resulting in a more dynamic output. Also note that if no peaks exist for a given time step w, the scale action will not have any impact on the resulting output.

2.4.1 Loudness

The Loudness scale action allows the user to specify a requested maximum velocity within a slice. The parameters for the γ-function are defined as

$$\theta = \max\left(\left\{\frac{\mathcal{N}\texttt{.velocity}}{128} \mid \mathcal{N} \in \mathcal{S}_{\boldsymbol{p}}^{(\mathcal{C})}\right\}\right), \quad \theta \in R_{[0,1]} \tag{26}$$

i.e. the maximum normalized (MIDI-) velocity where $\mathcal{N}$ denotes each note within the corresponding slice $\mathcal{S}_{\boldsymbol{p}}^{(\mathcal{C})}$, and $\bar{\theta} \in \mathbb{R}_{[0,1]}$ is the by the user requested normalized maximum velocity.

2.4.2 Vertical Density

The Vertical Density scale action allows the user to control how many notes there ideally should be in the output. The parameters for γ are given by

$$\theta = \left|\mathcal{S}_{\boldsymbol{p}}^{(\mathcal{C})}\right|, \quad \theta \in \mathbb{Z}_+ \tag{27}$$

and $\bar{\theta} \in \mathbb{Z}_+$ is the by the user requested number of notes.

2.4.3 Duration

The Duration scale action gives the user control over the average duration of each slice in the output. γ is defined so that

$$\theta = d_{\boldsymbol{p}} \tag{28}$$

where $d_{\boldsymbol{p}} \in \mathbb{R}_+$ denotes the duration in ticks of slice $\mathcal{S}_{\boldsymbol{p}}^{(\mathcal{C})}$ and $\bar{\theta} \in \mathbb{R}_+$ is user-defined.

2.4.4 Tempo Consistency

When using a corpus consisting of multiple sections with varying tempi, one problem is that the tempo changes too often, sometimes with every slice, which may obscure the user's perception of beat. The Tempo Consistency scale action is designed to remedy this behaviour. Unlike the previous filters in this section,

[7] but similar solutions exist in the audio domain and the architecture fully supports adding such solutions, should this change in the future.

it is not designed so that the user defines a requested tempo (as this can already be done in the scheduler), rather it serves as a low-pass filter based on the history of previous tempi so that

$$\theta = \zeta_{\boldsymbol{p}}, \quad \theta \in \mathbb{R}_+ \tag{29}$$

$$\bar{\theta} = \frac{1}{q}\sum_{k=1}^{q} \zeta_{w-q}, \quad \bar{\theta} \in \mathbb{R}_+ \tag{30}$$

where $\zeta_{\boldsymbol{p}}$ denotes the tempo in BPM of slice $\mathcal{S}_{\boldsymbol{p}}^{(\mathcal{C})}$, ζ_{w-q} denotes the tempo of previous output slice $\mathcal{S}_{w-q}^{(\mathcal{Y})}$ and q is a user controlled parameter corresponding to the length of the filter, i.e. how many previous output slices to take into account when applying the filter.

2.4.5 Spectral Distribution

The final filter in this category, the Spectral Distribution scale action, is intended to give the user some control over the spectral distribution, which in this case corresponds to the number of MIDI notes in each octave, and is the only filter in this category that does not apply the above procedure of scaling based on a gaussian. Here $\gamma \in \mathbb{R}_{[0,1]}$ is defined as a function $\gamma\left(\boldsymbol{\theta}, \bar{\boldsymbol{\theta}}\right)$ where the user specifies a vector $\hat{\boldsymbol{\theta}} \in \mathbb{R}_{[0,1]}^{11}$ where the value at $\hat{\boldsymbol{\theta}}[i]$ corresponds to the weight to apply to the i:th octave, $i \in [0, 11)$.[8] A similar value $\boldsymbol{\theta} \in \mathbb{R}_{[0,1]}^{11}$ is calculated for each slice $\mathcal{S}_{\boldsymbol{p}}^{(\mathcal{C})}$ corresponding to a peak using algorithm 2:

Algorithm 2 Computing spectral distribution $\boldsymbol{\theta}$ for slice $\mathcal{S}_{\boldsymbol{p}}^{(\mathcal{S})}$

```
$\boldsymbol{\theta} := \begin{bmatrix} 0 & \dots & 0 \end{bmatrix}$
for $\mathcal{N} \in \mathcal{S}_{\boldsymbol{p}}^{(\mathcal{C})}$ do
    $b := \left\lfloor \frac{\mathcal{N}.\texttt{pitch}}{12} \right\rfloor$
    $\boldsymbol{\theta}[b] := \boldsymbol{\theta}[b] + 1$
end for
$\boldsymbol{\theta} := \left(\max_{\theta \in \boldsymbol{\theta}} \theta\right)^{-1} \boldsymbol{\theta}$
```

Finally, γ is given by the Euclidean distance between the requested distribution and the actual, i.e.

$$\gamma\left(\boldsymbol{\theta}, \bar{\boldsymbol{\theta}}\right) = \left\|\boldsymbol{\theta} - \bar{\boldsymbol{\theta}}\right\|_2 . \tag{31}$$

[8]Octave numbers start from MIDI note number 0, e.g. octave 0 corresponds to note numbers 0-11, octave 1 note numbers 12-23, etc., resulting in a total of 11 octaves, even if the bottom and top octaves are very rarely used in musical contexts.

2.5 Regional Masking

The final scale action is the Regional Masking scale action, which allows the user to specify a region of indices within the corpus that the output should be selected from, e.g. to select only one of multiple movements in a multi-movement corpus. More specifically, the user specifies an interval $[a, b],\ a, b \in \mathbb{Z}_{[1,U]},\ a < b$ where U denotes the size of corpus $\mathcal{C}$, and applies a parameter γ (according to equation 24) defined as

$$\gamma = \begin{cases} 1 & \text{if } a \leq u\left(\mathcal{S}_{\boldsymbol{p}}^{(C)}\right) \leq b \\ 0 & \text{otherwise.} \end{cases} \tag{32}$$

2.6 Output Selection Modes

This report introduces two new modes for output selection: Probabilistic output selection and Quality Threshold. As mentioned in section 2.1, the default mode for selecting the output slice $\mathcal{S}_w^{(\mathcal{Y})}$ at time step w is to select the slice corresponding to the peak with the highest score through a selection function $\Omega\colon \mathbb{R}^{3\times K} \to \mathcal{S}$. To provide a better framework for the new classifiers, we will briefly revisit this default output selector and introduce some additional definitions, followed by the introduction of the new modes.

2.6.1 Output Selection in the General Case

As per equation 18, we define the output slice $\mathcal{S}_w^{(\mathcal{Y})}$ at time step w as a function of the peak matrix $\boldsymbol{P}_w^{(*)}$, the corpus $\mathcal{C}$ and the history of previous output slices $\mathcal{H}_w$. More specifically, we can for all output selection modes define a set $\Xi = \{\boldsymbol{p}_1, \ldots, \boldsymbol{p}_H\}$ of viable peak candidates with the corresponding set of slices $\xi = \left\{\mathcal{S}_{\boldsymbol{p}_1}^{(\mathcal{C})}, \ldots, \mathcal{S}_{\boldsymbol{p}_H}^{(\mathcal{C})}\right\}$. For each slice, a probability ρ_h is defined so that

$$\rho_h = P\left(\mathcal{S}_{\boldsymbol{p}_h}^{(\mathcal{C})}\right), \quad \forall\, \mathcal{S}_{\boldsymbol{p}_h}^{(\mathcal{C})} \in \xi. \tag{33}$$

Given a stochastic variable $X \sim \mathrm{U}[0, 1]$ the output slice is selected through a sampling function $g(\rho)$ so that

$$g(\rho) = \begin{cases} \mathcal{S}_{\boldsymbol{p}_1}^{(\mathcal{C})} & \text{if } X < \rho_1 \\ \mathcal{S}_{\boldsymbol{p}_2}^{(\mathcal{C})} & \text{if } \rho_1 \leq X < \rho_2 \\ \vdots & \\ \mathcal{S}_{\boldsymbol{p}_{H-1}}^{(\mathcal{C})} & \text{if } \rho_{H-2} \leq X < \rho_{h-1} \\ \mathcal{S}_{\boldsymbol{p}_H}^{(\mathcal{C})} & \text{otherwise} \end{cases} \tag{34}$$

In the case of the default output selection mode («maximum»), we define Xi as the set of peaks corresponding to the peak with the highest score ($H > 1$

if multiple peaks have the same maximum value), i.e.

$$\Xi = \left\{ \boldsymbol{p} \,\middle|\, \boldsymbol{p} \in \boldsymbol{P}_w^{(*)} \wedge \left(y_{\boldsymbol{p}} = \max_{\boldsymbol{p}_j \in \boldsymbol{P}_w^{(*)}} y_{\boldsymbol{p}_j} \right) \right\}, \tag{35}$$

with corresponding probabilities

$$\rho_h = P\left(\mathcal{S}_{\boldsymbol{p}_h}^{(\mathcal{C})}\right) = \frac{1}{|\Xi|}, \quad \forall\, \mathcal{S}_{\boldsymbol{p}_h}^{(\mathcal{C})} \in \xi. \tag{36}$$

The output slice $\mathcal{S}_w^{(\mathcal{Y})}$ is given by

$$\Omega\left(\boldsymbol{P}_w^{(*)}, \mathcal{C}, \mathcal{H}_w\right) = \begin{cases} g(\rho) & \text{if } \Xi \neq \varnothing \\ \mathcal{S}_{u+1}^{(\mathcal{C})} & \text{otherwise,} \end{cases} \tag{37}$$

where $g(\rho)$ is defined according to equation 34 and $\mathcal{S}_{u+1}^{(\mathcal{C})}$ denotes the slice following the previous output slice, i.e. the slice $\mathcal{S}^{(\mathcal{C})} \in \mathcal{C}$ fulfilling the condition

$$u\left(\mathcal{S}^{(\mathcal{C})}\right) = u\left(\mathcal{S}_{w-1}^{(\mathcal{Y})}\right) + 1 \qquad \bmod U. \tag{38}$$

2.6.2 Probabilistic Output Selection

The Probabilistic output selection mode will, unlike the default output selection mode, take all peaks into account and create a probability density function where the probability is defined by the score of each peak, i.e.

$$\Xi = \left\{ \boldsymbol{p} \,\middle|\, \boldsymbol{p} \in \boldsymbol{P}_w^{(*)} \right\} \tag{39}$$

and

$$\rho_h = P\left(\mathcal{S}_{\boldsymbol{p}_h}^{(\mathcal{C})}\right) = \frac{y_{\boldsymbol{p}_h}}{\sum_{i=1}^{H} y_{\boldsymbol{p}_i}}, \tag{40}$$

where $g(\rho)$ and Ω are defined according to equations 34 and 37 respectively. In this mode, the probability of matches with lower quality (i.e. peaks with a lower score) increases, but the variation increases. Due to the latter, it's less likely to get stuck in deterministic loops when interacting with the system.

2.7 Quality Threshold

The Quality Theshold output selection mode is an extension that can be combined with either of the two previous modes, in which the user defines a threshold $\psi \in \mathbb{R}_+$, where all peaks with scores below this value will be discarded, i.e.

$$\Xi = \left\{ \boldsymbol{p} \,\middle|\, \boldsymbol{p} \in \boldsymbol{P}_w^{(*)} \wedge \left(y_{\boldsymbol{p}} = \max_{\boldsymbol{p}_j \in \boldsymbol{P}_w^{(*)}} y_{\boldsymbol{p}_j} \right) \right\} \cap \left\{ \boldsymbol{p} \,\middle|\, \boldsymbol{p} \in \boldsymbol{P}_w^{(*)} \wedge y_{\boldsymbol{p}} \geq \psi \right\} \tag{41}$$

when combined with the default (≪maximum≫) mode and

$$\Xi = \left\{ \boldsymbol{p} \,\middle|\, \boldsymbol{p} \in \boldsymbol{P}_w^{(*)} \right\} \cap \left\{ \boldsymbol{p} \,\middle|\, \boldsymbol{p} \in \boldsymbol{P}_w^{(*)} \wedge y_{\boldsymbol{p}} \geq \psi \right\} \tag{42}$$

when combined with the probabilistic mode. The output selection function is in both cases defined as

$$\Omega\left(\boldsymbol{P}_w^{(*)}, \mathcal{C}, \mathcal{H}_w\right) = \begin{cases} g(\rho) & \text{if } \Xi \neq \varnothing \\ \text{undefined} & \text{otherwise,} \end{cases} \tag{43}$$

where ρ is defined as equations 36 and 40 respectively and $g(\rho)$ as equation 34. More importantly, as seen in equation 43, the output is undefined if no peaks exists in Ξ, either because no matches were found in earlier stages of the generation process so that $\boldsymbol{P}_w^{(*)}$ is empty or because no peaks fulfil the condition $y_{\boldsymbol{p}} \geq \psi$. When undefined, no output will be generated for this particular generation step w. In other words, if the quality of the slices corresponding to the existing peaks does not match the requested quality (as defined by the threshold ψ), no output will be generated at all, hence introducing a degree of sparsity in the generated material.

3 The Wireless Paradigm

The user interface of Somax, which was discussed briefly in [6] and [7], is implemented in the Max [2] programming language. The user interface was originally designed as a thin client, where all of the computation is handled on the Python server (apart from the real-time signal processing required for audio signals used as influences). As most readers likely are aware of, Max is a visual programming language where the default means of programming is by connecting objects using patch cords. In most cases, the readability of a Max program is determined by how easy it is to follow the cords throughout the program. The Somax user interface was originally designed with this in mind to promote readability on both micro and macro levels of the program, but is from version 2.3 using wireless communication (`send` and `receive`) between objects on a macro level. While this approach to some extent obscures the readability (or at least the global signal flow) of the system, the benefits are manifold. First of all, the architecture becomes easier for the user to extend - adding new players can be done with a single keypress - and objects can dynamically select which other objects to interact with without having to modify the architecture. This new architecture and its implications will be presented in section 3.1.

Another purpose of this redesign is to make the system integrable into Ableton Live. Somax can as a system be described as a ≪function≫ that reads one or multiple audio and/or MIDI streams and outputs one or multiple MIDI streams. For compatibility with Live, this has to be split into several smaller objects based on Live's syntax of ≪instruments≫ (function that reads one MIDI stream and outputs one audio stream), ≪audio effects≫ (function that reads one audio stream and outputs one audio stream) and ≪MIDI effects≫ (function that reads one MIDI stream and outputs one MIDI stream). Using a wireless architecture, this goal is possible to achieve for Somax. Do note that the wireless paradigm presented in this report is just a prerequisite for the Max for Live version of Somax; the latter it is still work in progress and will be presented in a future report.

To accommodate these changes, the Python back-end has been updated to drastically increase the performance when using multiple players. This will be presented in section 3.2.

3.1 The Wireless Max Architecture

The Max architecture currently consists of four main objects: `AudioInfluencer`, `MidiInfluencer`, `Player` and `Server`. The `AudioInfluencer` and `MidiInfluencer` read a continuous stream of audio or MIDI data respectively and perform the slicing and trait analysis steps mentioned in section 2.1. In the user interface, the data resulting from this process is called an `influence`, which is routed to a player (and the Python back-end) to compute the following steps in the influencing process. The `Player` object is essentially a client for the corresponding `Player` class in the Python back-end, which handles all of the runtime architecture described in section 2.1. Finally, the `Server` object handles communication with the `Server` class in the Python back-end, which is the root of the entire system, handling all players as well as the transport (i.e. the master clock of the event scheduling). So far, this is identical to the architecture presented in [6], apart from the fact that that there is no routing object in the middle, which is now rather handled wirelessly and will be described further down in this section.

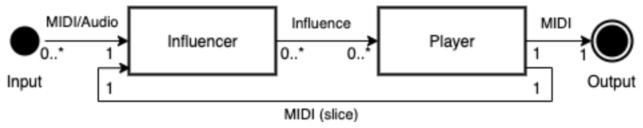

Figure 2: Generalized interaction model of the Somax user interface.

Figure 2 depicts the interaction model of the user interface on a large-scale level. Each input is sent into an influencer, which computes discrete influences from the data and forwards it to one or multiple players. An influencer may send influences to multiple players and a player may receive influences from multiple influences (hence, a many-to-many relationship). Each player generates an output MIDI stream based on said influences, which is also sent into an influencer to allow further influencing of other players with the generated output.

A simplified diagram over the entire wireless system can be seen in figure 3. Here we see that the only objects that do not have corresponding objects in the Python back-end are the influencers. Each influencer is given a name by the user, which the system ensures is unique, and this name will serve as the address on which the influencer sends its influences to players.

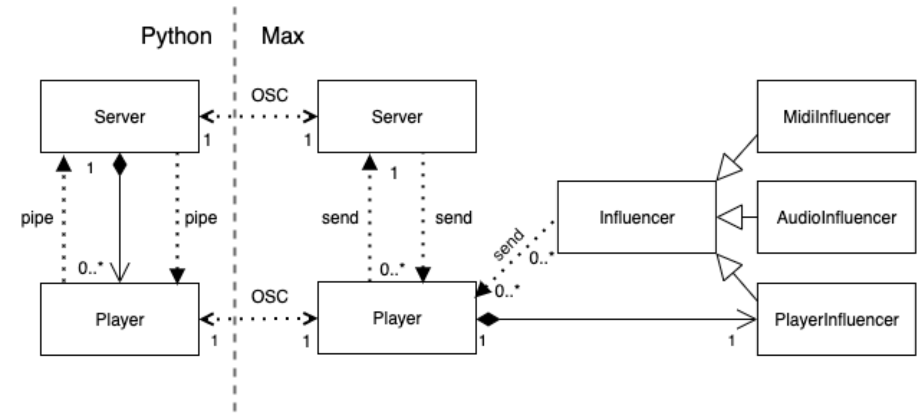

Figure 3: Interaction model for the wireless architecture. Dotted arrow lines denote some sort of "wireless" communication between objects while filled arrow lines denote their traditional UML relations (composition, generalization) and corresponding cardinality.

The `Player` object in Max was originally designed as a thin client around the `Player` class in the back-end, but has with this update been given two additional roles: routing and re-influencing. The routing module of the player (whose user interface is depicted in figure 4) lists all available influencers and players and receives influences based on what the user selects. The player also contains a `PlayerInfluencer`, which is a simple variation on the `MidiInfluencer` that also accepts certain metadata from the back-end to optimize performance and reduce latency, and give the user more detailed control when routing influences between players.[9] When initialized, the `Player` is assigned a unique (user-controlled) name as well as two unique OSC ports - one for receiving one for sending messages to the corresponding `Player` class. An instantiation message is sent to the `Server` object, which creates and owns the `Player` in the back-end, but once communication is established, the `Player` object and `Player` class communicate influences, MIDI output and parameter changes directly over said OSC addresses.

Finally, the `Server` object, which directly corresponds to the `Server` class (in Python) and the root of the entire system, is also initialized with unique receive and send OSC ports as well as a unique name. In addition, it also contains the beat tracker module [5], corpus builder (which is a simple user interface for the process of corpus building as described in section 2.2 of [7]), as well as a number of utilities for managing and recording the MIDI output. The role of the `Server` class will be described in detail in section 3.2.

[9]More details on advanced routing with the `PlayerInfluencer` can be found in [3].

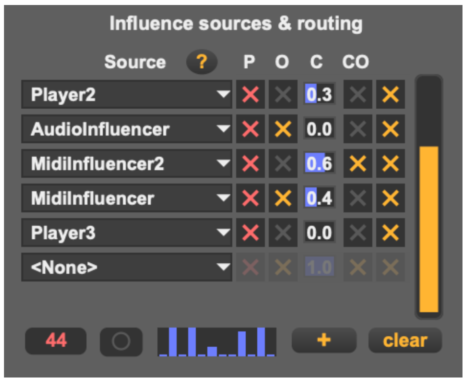

Figure 4: The routing module of each player. Here, all influencers and other players are listed, and the user can select whether to listen to pitch influences (P), onsets (O) and/or chroma influences (C) from that particular source. As chroma influences are continuous, it's also possible to mix chromas from different sources. The point of segmentation is determined by the chroma onset (CO).

3.1.1 Additional Updates to the User Interface

In addition to the changes related to the wireless architecture, the user interface of each individual component has been redesigned from scratch. A Model-View-Controller design pattern has been applied to ensure that no inconsistencies exist between the Python back-end (model) and the Max user interface (view and controller). Each object has been remodelled so that multiple views/controllers may exist for the same data - one compact view, listing only the most vital parameters that are necessary to give the performer an overview while interacting with the system, and one full view where all parameters are available. This update also includes detailed (user-oriented) documentation of each object as well as a tutorial.

3.2 The Parallelized Python Architecture

Due to the changes presented in the previous section, the architecture of the Python back-end, originally implemented in a concurrent (but single core) manner, had to be updated to a parallel multicore solution. As shown in section 1.4, a generate-influence cycle for a single player can in some cases take 30 milliseconds or more. In the case of a single player, this means 30 milliseconds of latency for the player, which of course is not ideal, but it will not break the perception of pulse assuming that the latency stays fairly constant. This is however not the case for multiple players operating on a single core. E.g., if all three players receives their influence/onset messages simultaneously, the delay between sending the message and the output of the last player can be up to 90 milliseconds, while if influence/onset messages overlap perfectly, the delay for each player is only 30 milliseconds. This creates a latency interval for each player varying between 30-90 ms under even load (in this particular case, increasing linearly with each added player), effectively breaking all perception of pulse in the generated content.

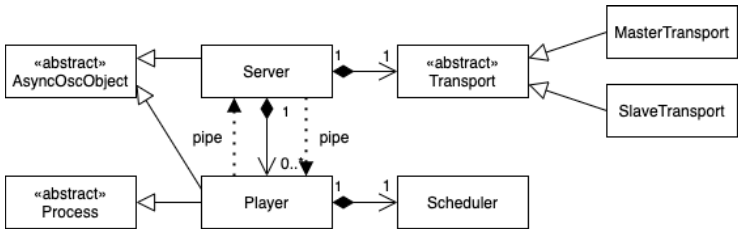

Figure 5: Simplified class diagram over the new parallelized architecture.

In the new architecture, shown in figure 5, which combines the concurrent strategies of the previous implementation with parallelization, each `Player` runs in its own `Process` [4], thereby not impacting each other performance-wise as long as there are fewer players running than free cores on the system. The `Server` is running two coroutines using the `asyncio` module [1], where one coroutine continuously receives messages from the Max `Server` object over OSC and the other coroutine continuously updates the time of the `Transport` class and forwards this to each `Player`. Do note that there are two different `Transport`s available in figure 5 - the `MasterTransport` and the `SlaveTransport`. In the former case, the time (measured in ticks $t \in \mathbb{R}_+$ as in previous sections) is updated with each callback so that at update $i = 0, 1, \ldots,$

$$t_i = t_{i-1} + \Delta T \cdot \frac{\zeta_{i-1}}{60}, \tag{44}$$

where $\Delta T \in \mathbb{R}_+$ denotes the time (in seconds) elapsed since the last callback and ζ_{i-1} denotes the tempo of the scheduler at the previous time step. In the case of the `SlaveScheduler`, the idea is that the time is updated from an external source, for example the master clock of Max or the current time from Ableton Live, i.e.

$$t_i = t_i^{(\text{ext})}. \tag{45}$$

These messages are sent over OSC through the `Server` object (in Max) and the second coroutine of the `Server` class is therefore unused in this case. The time t_i is at each update i sent to all `Player`s of the `Server` using a pipeline.

The `Player` class is also running two coroutines; one coroutine continuously receiving messages from the Max (parameter updates, influences and onsets) and one coroutine continuously receiving message from the pipeline connected to the `Server`. As in the earlier architecture, these messages are queued through the `Scheduler`, which was described in section 3.4 of [7], with the only exception that each `Player` now has its own `Scheduler`, instead of a shared one for all `Players`. Also, if the user has set a specific `Player` as the ≪Tempo Master≫, i.e. the source from which the `Transport` should receive its tempo, this data is returned to the `Server` through another pipeline and updated accordingly, e.g. for a tempo $\zeta_w^{(\mathcal{C})}$ returned by the `Player` after update t_i, the tempo of the `Transport` is updated so that

$$\zeta_i = \zeta_w^{(\mathcal{C})}. \tag{46}$$

References

[1] asyncio — Asynchronous I/O. `https://docs.python.org/3/library/asyncio.html`. Accessed: 2021-03-30.

[2] Cycling74 - Max. `https://cycling74.com/products/max/`. Accessed: 2021-03-24.

[3] A Gentle Introduction to Somax. `https://github.com/DYCI2/Somax2/blob/master/Introduction%20Somax.pdf`. Accessed: 2021-03-29.

[4] multiprocessing - Process-based parallelism. `https://docs.python.org/3/library/multiprocessing.html`. Accessed: 2021-03-30.

[5] 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.

[6] Joakim Borg. Somax 2: A Real-time Framework for Human-Machine Improvisation. *Internal Report - Ircam*, 2019.

[7] Joakim Borg. Dynamic Classification Models for Human-Machine Improvisation and Composition. Master's thesis, Aalborg University, 2020.