ANR-14-CE24-0002-01

Projet DYCI2,
WP3 Dynamiques d'interaction improvisée,
SP3.1 Guidage de l'interaction improvisée

Rapport de livrable :

L3.1.2 Guidage de l'interaction improvisée II par flux d'entrée.

Livrable	Date	Contributeurs	Rédacteurs	Contenu
L3.1.2 Version 01	Mars 2017	A. Chemla (STMS), G. Assayag (STMS), Laurent Bonnasse-Gahot (ex-STMS)	A. Chemla	Maquette logicielle, Rapport

Résumé
Ce document présente la technologie de guidage d'improvisation interactive homme-machine par réaction à un flux d'entrée, issuedu post-doc de Laurent Bonnasse-Gahot et de l'implémentation d'Axel Chemla Romeu-Santos

Répertoire de développement logiciel collaboratif
https://forge.ircam.fr/p/somax

Adresse du livrable logiciel
DYCI2_WP3.1_L.3.1.2.zip
sur
https://forge.ircam.fr/p/Dyci2/

SoMax Documentation

Getting Started

Going further

Advanced

Welcome to the SoMax Documentation. SoMax is a musical improvisation software which aims to provide a stylistically-coherant improvisation while adapting itself to the musical context from real live musicians.

This document is made to provide any useful infomation regarding to SoMax, whether you are an absolute beginner or an accomplished developer.

If you have troubles installing SoMax, please go to the setup section.

If you are looking for basic information on how to start and play with SoMax, please go to the Getting Started section.

If you want more specific explanations of some features or other details, please go to the Going Further page.

If you are a programmer interested on deeper details, see the Advanced section.

Contents

Setup

This SoMax version is made for Max 7 32-bit (you can toggle the 32-bit version of Max by opening its properties), even if it should also work with Max 6.

Corpus construction corpus. To be able to use the corpus construction library, the following Python librairies have to be installed :

- `LibRosa`[1], used for the computation of audio descriptors
- `scikits.samplerate`, that fastens greatly the computation.

To do this, you can open the Terminal and using the `pip` system by typing this :

```
pip install librosa

pip install scikits.samplerate
```

[1] https://bmcfee.github.io/librosa/

Chapter 1

Getting Started

This section is made to guide a first contact you could have with SoMax.

1.1 Playing with SoMax

In this section, you will learn how to play with SoMax. This section assumes that SoMax is properly installed ; if a Max error is shown, please insure that all the steps described in the setup section have been properly followed.

Controllers and players. In SoMax, there are two main patches : a **conductor** and a **player**.

- The *conductor* is managing the inputs that the players receive, the outputs that the players send and several parameters regarding the interaction modes (that we will deepen in the section below).
- The *player* is a unit which generates improvisation. It receives information from the outside (from human or virtual players) and generate musical content.

Please note that there are two conductors : one that listens to MIDI streams called `conductor_midi.maxpat` and one that listens to audio streams called `conductor_audio.maxpat`. There is only one player, which can generate audio or MIDI regarding to the corpus chosen.

It is possible to use players separately ; in a first time it is more conveniant to use the conductors, which manage all the routings and give you quick accesses to the most important parameters. If you are using a MIDI device, please open

Figure 1.1: Audio conductor (to the left) and MIDI conductor (to the right)

the MIDI conductor ; if you want to use an audio stream, please open the audio conductor.

Configuring inputs. Before playing, let's check if your outputs and inputs are properly configured.

For MIDI players, please first select the MIDI input with the popup button below the *midi input* indication. Then, please check the MIDI channel of your device : depending on the MIDI channel your device will be considered by the players as a foreground channel, a background channel or a beat channel ; to have further information on this, please go to this section. For this tutorial, please insure that your MIDI is sending notes to the channel 1. If it works properly, you should see the notes played on the displayed keyboard.

For Audio players, please check in the Options>Audio Status, and activate the DSP status. If everything is OK, the input part should react to the audio stream.

Awaking the player. Before all, please press in the bottom of the conductor the *note against note* preset for `p_1` (it will be explained in the following tutorial). Now that the input is properly configured, please double-click on the first player, or the box `soma player_1`. A similar window than below should appear (cf fig. 1.4).

You juste have to select the corpus you want to play with in the popup menu, located at the top of the window. Now, play your instrument, and the system will play a note each time it receives an event from you (a note in MIDI, or an onset in audio).

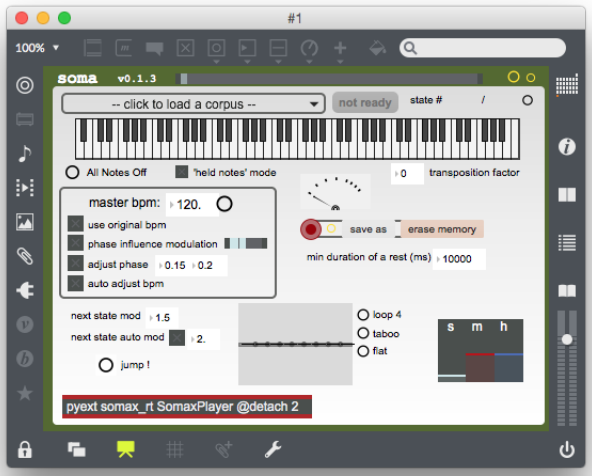


Figure 1.2: SoMax player patch

1.2 Testing interaction modes

Playing with interaction matrix. Now that we have managed to make the system play, now we will focus a bit on interaction modes. As SoMax is conceived in a rather free way, a lot of interaction modes are possible ; the conductor is offering a quick way to configue effective interactions with the interaction matrix. The interaction matrix is the grid displayed on the conductor as below :

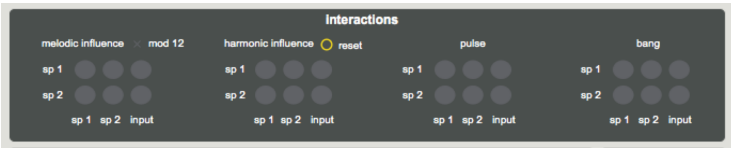


Figure 1.3: SoMax player patch

There are several matrices, defining how the system will be influenced by the input for the following musical dimensions : one for the melodic influence, one for the harmonic influence, one for the pulsation influence and one for "bang" (we will see this later).
This matrix has to be read as if the input entered the routing matrix from below, and were leaving from the right where a dot is lighted. If you look at the pictures below, you can see that the first one means that the melody of the input is influencing the first SoMax Player. The second matrix means that the input's pulsation is influencing the second SoMax Player.

Figure 1.4: Examples of influences matrix : melodic influence on the left, and pulse influence on the right.

Basically, here are quickly explained the meaning of the different parameters shown in the *Interactions* part:

- **melodic influence** : the melody detected from the input player is influencing the target player. When you are playing with MIDI, this listens to the *foreground channel* (see below).
- **harmonic influence** : the harmonies detected from the input player are influencing the target player. When you are playing with MIDI, this listens to the *background channel* (see below).
- **pulsation influence** : the pulsation detected from the input player is influencing the target player. When you are playing with MIDI, this listens to the *beat channel* (see below).
- **bang** : the onsets of the input player triggers the output of the target player. This typically gives a "note-to-note" effect on the interaction.

Foreground and background channels. When playing with MIDI, you can use separate MIDI channels to specify your interaction with SoMax. This can be done using the MIDI input matrix :

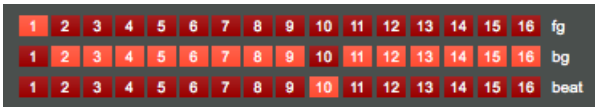


Figure 1.5: MIDI matrix on the MIDI conductor

here you can specify which MIDI channel you will send to the *foreground* channel, used for pulsation and melodic influences, and the ones you will send to the *background* channel, used for the harmonic influences. For example up here, only the first MIDI channel goes to the foreground channel, while every other channel is sent to the background channel. The beat channel is used for the pulse influence

Now play with these different interactions and make the best with each of these!

1.3 Modelling improvisation

Now that we have explored the different interactions of the conductor, we will now explore how to model the improvisation generated by each of the player.

The improvisation can be modeled with many parameters, reacting to its input or independently with internal rules.

Corpus. SoMax being a recomposition-based improvisation system, it needs a *memory* to perform its generation. This memory, that will be recomposed during the improvisation, can come from audio material or from MIDI material. You can select the memory you want to improvize with by selecting a specific *corpus* in the popup button in top-right corner of this window. You can also construct your own corpus ; more explanations in the section 2.3.2

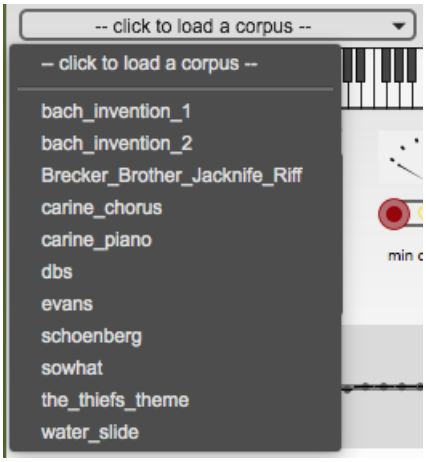


Figure 1.6: Corpus selection with the popup button

Influences. This is with the concept of *influence* that the reactivity is brought into SoMax. An *influence* is the active listening of a musical dimension that will guide the system towards a generation judged "*relevant*". Details can be found later (see section 2.3).

SoMax Player can listen to three musical dimensions : the *melodic* influence, the *harmonic* influence and the *self-influence*. The respective amount of influence of these three dimensions can be specified using the three sliders at the bottom of the window as below. The first one is the self-listening influence amount, the second one the melodic influence amount and the third one is the harmonic influence amount.

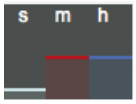


Figure 1.7: Influences' multislider of a SoMax Player

As the two first influences are quite clear and rely with the previous part, the *self-influence* needs a little more explanation. Basically, increasing self-influence

means gives higher priority to the memory's coherance with itself, whereas increasing other influences means gives higher priority to reaction.

Transposition & BPM. It is also possible to transpose the memory of the player. The recombination algorithm will thus find its solutions regarding to the transposition factor. This, for example, can be interesting if a given corpus belongs to a particular key, and that you want to play in another one.

It is also possible to give to the algorithm the BPM you want to play with, by adjusting the master BPM. Though, if you the player to keep the orginal tempo, you can check the *use original BPM* box. For further informations regarding to the rythmic adjustments, please refer to the rythmic adjustments part.

Taboos The *taboo* table is a high-level method to slightly conduct the generation of the algorithm. The taboo system weights the places of the memory that the system chose in a recent past.
This is basically done by example to penalize the freshly read states, avoiding the system to loop and forcing it to go somewhere else. To do that, you can click the preset *taboo*. In a different way, you can promote some of the read states to force the system to enter a loop. The detailed explanation of how taboos work can be read here.

1.4 Recording online

It is also possible to record online material to build a new corpus to play with. It can be MIDI or audio, depending of the input you give to him.

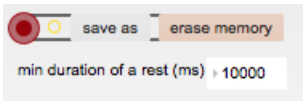


Figure 1.8: Recording in the `soma.maxpat`

To record on online material, please first click on `erase memory` (don't worry, it doesn't erase the original memory but just the one currently loaded in the player). Then, you just have to click the record button, and play. Click again to stop recording, then click the `save as` button to write what you have written in a file.
Please not that, if you are recording MIDI and using conductors, your input must to be in the pitch influence of the player.

Chapter 2

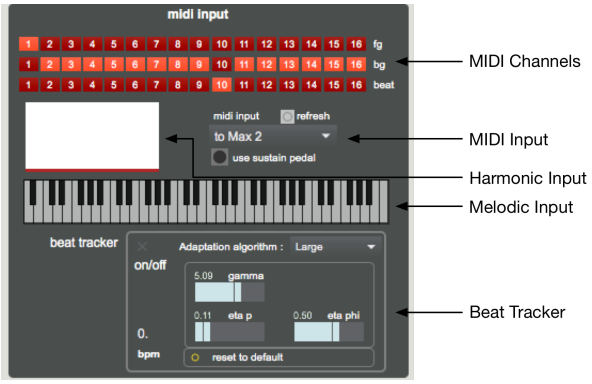
Going Further

This section is made to make you have a deeper understanding of SoMax by detailing all the features of the conductors and of the player, and by explaining how SoMax generates its improvization.

2.1 Overview of the conductors

This section will now exhaustively describe all the features displayed on the conductors. Notice that the only difference between the midi conductor and the audio conductor is the input part, so everything excepting this is relevant for both.

MIDI Input

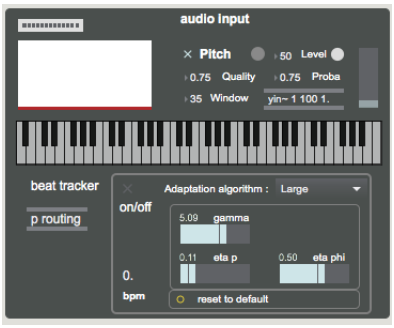


- **MIDI channels** : maps the channel of the MIDI input to the three channels used by SoMax players. The *foreground channel* (or *fg*) is used to influence

the melodic dimension of the players. The *background channel* (or *bg*) is used to influence the harmonic dimension of the players. Finally, the beat channel can be used to give pulsation to the players, that can help them to be synchronous (see the rhythmic adjustments part).

- **MIDI Input** : here is the popup menu to select the MIDI input you want to use.
- **Harmonic Input :** on this graph is shown the harmonic context computed with the data arriving by the background channel. This is a table containing the amount detected of the twelve notes of the tempered scale.
- **Melodic Input :** here is shown on a keyboard the melodic flow arriving by the foreground channel.
- **Beat Tracker :** this is a beat tracker which can be used to estimate the pulse of the MIDI Input. To activate it, please check the box at the top-left corner of the window. You can also press the space bar of the computer to give the players pulsation.

2.1.1 Audio Input



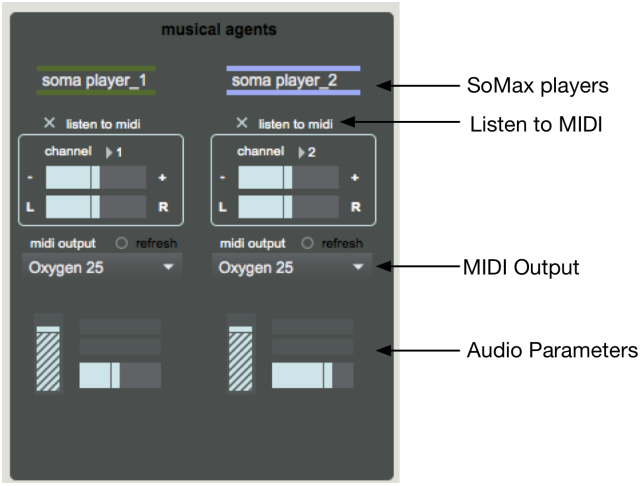
- **Input level** : shows the level of the audio input.
- **Pitch tracker** : here are the parameters of the pitch tracker, based on the Yin algorithm.
- **Harmonic Input** : here is shown the harmonic context detected on the audio input. This is a table containing the amount detected of the twelve notes of the tempered scale. This harmonic context is made with the chroma detection algorithm of `ircamdescriptors`.
- **Routing window** : double click the routing subpatch to select the audio input sent to the players, and the audio input used by the player in the case of recording.
- **Melodic Input** : here is shown on a keyboard the pitch computed by the pitch tracker.

- **Beat Tracker** : this is a beat tracker which can be used to estimate the pulse of the MIDI Input. To activate it, please check the box at the top-left corner of the window. You can also press the space bar of the computer to give the players pulsation.

The segmentation of the audio input flow is made with an onset detection algorithm based on the `bonk` Max external object. Please note that to configure the audio input device, you have to configure that into the Max preferences (Options>Audio Status).

Output

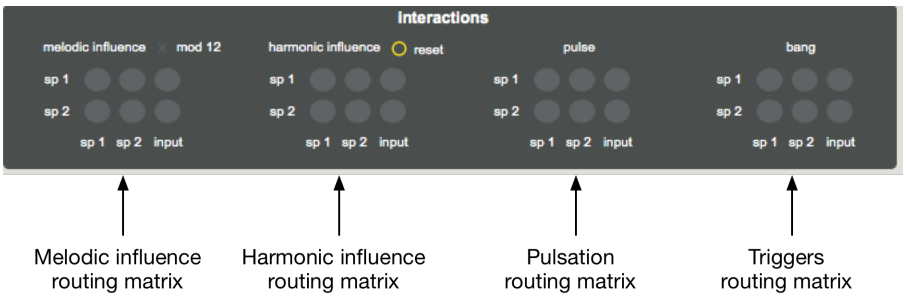
Please notice that the input sections are different between the audio conductor and the MIDI conductor, but that it is the same for the output.



- **Players** : accessing the players' parameters by double-clicking the boxes. The detailed overview of the player is available here.
- **Listen to MIDI :** as a player can produce MIDI streams and audio streams, you can able/disable the MIDI output of a player by checking this box.
- **MIDI Output** : with this popup menu, you can choose the MIDI output device. The refresh button updates the MIDI outputs available.
- **Audio Parametrs** : in the case where the player outputs audio streams, you can set the volume with the vertical slider and the pan with the horizontal slider.

Interactions matrix

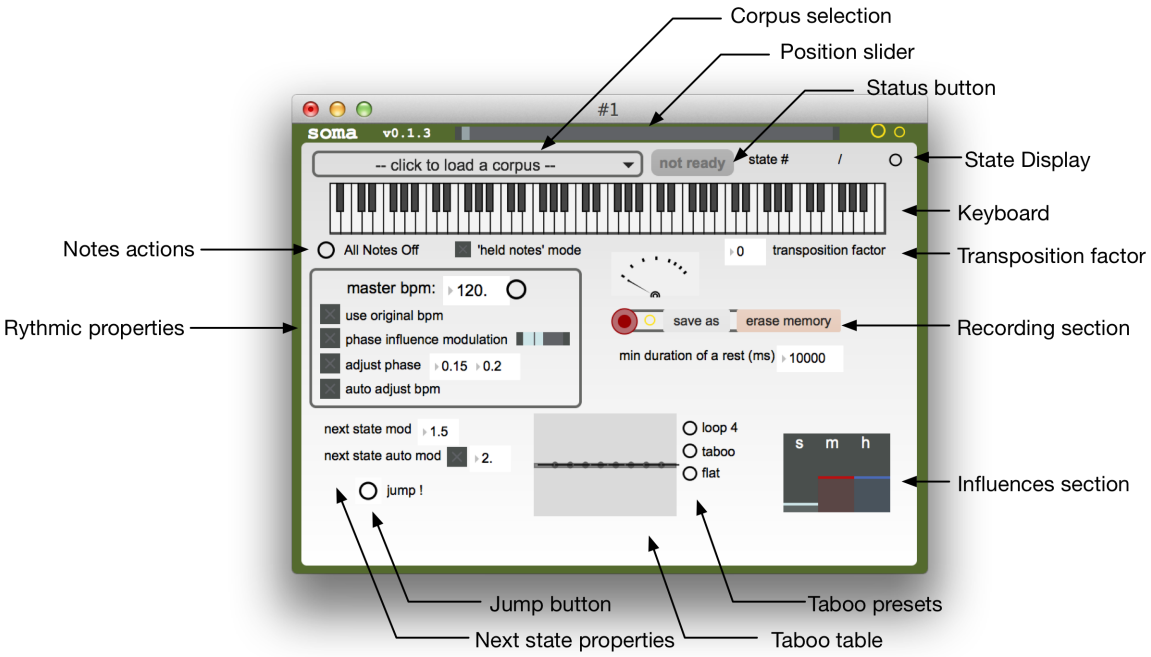
For all the matrices below, a matrix has to be read from bottom to left. This means that the input is arriving vertically, and the output is leaving horizontally. This is why input is available only as a column.



- **Melodic influence** : routes where the melodic information goes. Activating the *mod 12* checkbox means that the influence will be insensitive to the input's octave.
- **Harmonic influence** : routes where the harmonic information goes.
- **Pulse influence** : routes where the pulse information goes. Be careful, you can send input pulse information only if the beat tracker is activated.
- **Triggers** : the *bang* matrix is a special matrix which forces player's output in certain cases, as the note-against-note mode for instance.

There are below this part two routing presets. The first, called *note-against-note mode*, will force the player to answer any input note by another note. The second, called *beat mode*, will let the player improvize freely according to its parameters.

2.2 Overview of the player



- **Corpus Selection** : here you can select the memory the player will use.

- **Position Slider** : when the player improvizes, you can see the location in the memory in the played state. You can also force the player to go somewhere in the memory by adjusting manually the slider.

- **Play button** : tells the player to start or to stop improvizing. It can also show *not ready*, if no corpus is selected or if a problem occured.
- **State Display** : shows the position of the current state, and lights up when the system jumps in the memory.

- **Keyboard** : a keyboard-style display to show the current notes played by the player.

- **Transposition factor** : transpose the memory by the selected number of semi-tones. This can be useful if the corpus is in a given key and that the improvizer wish it to be in another one.
- **Recording session** : here you can record new corpus online. To do that, first erase the memory (otherwise it will be added to the previous one, which can be OK if you want to concatenate) and then click the record button.

When you are done, click again the record button and click `save as`, where a corpus name will be asked ; the corpus will be automatically added to the corpus list. Please note that if you are using conductors the input stream has to come into the player as pitch influence!

- **Influences section** : here you can select the respective amounts of self-influence, melodic influence or harmonic influence of the player. Reinforcing self-influence will encourage the player to promote its consistency with its memory, whereas reinforcing melodic or harmonic influence will encourage him matching the external context. More details in here.

- **Taboo table** : Here are shown the taboo weights that the player will use to ponderate the likeliness of the previous played states for a further selection. The points symbolize, from left to right, the weights applied to the eight previous selected states of the player. If the weight is negative, the corresponding state will be disavantaged, whereas if the weight is positive the state will be encouraged.
- **Taboo presets** : here are some presets for the taboo table. The *Loop 4* preset puts a high positive weight on the state played 8 steps before, creating some kind of loop in the improvization. The *taboo* preset corresponds to a classic taboo situation, where the closed past states are strongly inhibated to prevent the system falling in loops.

- **Jump button** : forces the system to jump in the memory. Can be used if the system is stuck in a region of the memory.

- **Next state properties** : SoMax, when selecting a state, automatically modulate the activity of the conjoint state of the memory to encourage continuity. The *next state mod* allows to set the amount of boost of the conjoint state. The *next state auto mod* forces the system to jump to avoid to much continuity, with a parameter which is a measure of the maximum tolerated continuity.
- **Rythmic properties** : here are shown all the features available regarding to the rythmic adjustment. You can set manually the BPM of the player, or tell him to use the original tempo information of its memory by clicking the *use original BPM* box.
  The *phase influence modulation* makes the system sensitive to the current phase in pulsation for its state selection (see here). That means that it will

favorize states in the memory close to the current rhythmic phase.
The *phase adjust* modulation is a feature of the SoMax system that is preserving the original timing of the memory as it rearranges it, while preserving the given pulsation feeling. If you want the player to align its pulsation to an external pulsation (via the conductor for instance), you can check the box *auto adjust bpm*, and define the range of possible BPMs.

- **Notes actions** : the *all notes off* button is a kind of panic button which cuts all the currently played notes. The "held notes" mode will prevent the player to play notes by itself, waiting for external signals to play one (as in the *note-against-note* mode in the conductor by instance).

2.3 Generation algorithm

To really understand what all the features described above really mean, we will now describe how SoMax generates its improvisation. For even further information, please read the Bonnasse-Gahot's report on SoMax[1].

SoMax is based on a musical memory, an existing musical sequence **segmented** into states and **labelled** according to their musical content. The improvisation is generated by recomposing it according to a non-linear path, assuring a stylistic consistency with the original sequence and a relevance regarding to the external context. The selection of this non-linear path is based on the key concept of *activity*, which is described below.

2.3.1 Activity and recombination method

Activity. The *activity* of a memory state can be understood as its score of relevance depending on the current musical context. To become reactive, the system is listening to the external context and takes observations from it. These observations will *activate* the states of the memory that show a certain amount of similarity and will softly guide the system towards the more relevant solutions.

In this version of SoMax, the memory is based on a n-gram tree, which can be seen as a dictionary of all the n-sized sub-sequences of the memory. This is to compare the n last observations with all the n-sized subsequences of the memory, introducing some past-depth in the reaction. Note that the values of the activity are continuous, so the total score can be seen as te total relevance of the subsequence..

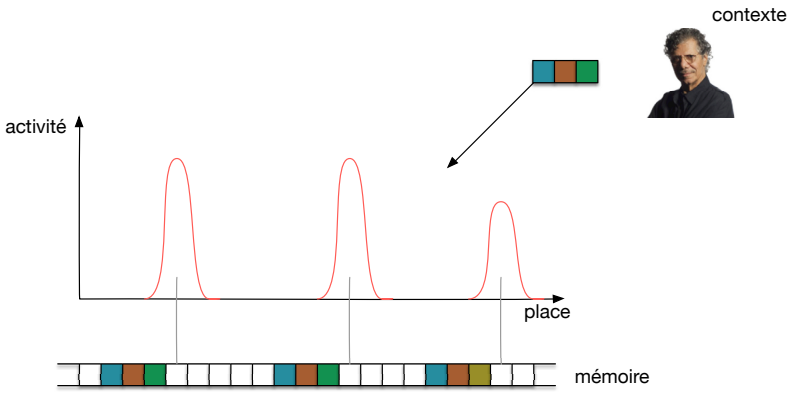


Figure 2.1: Activity of a memory stimulated by a 3-sized length observation history

The memory is a succession of musical segments, but this activity concept

[1] Laurent Bonnasse-Gahot, *An update on the SoMax project*, September 2014

lies on a continuous representation of the time. The activity is modelized as a Gaussian, created at the arriving on the observation and evolving synchornously through the memory. This Gaussian is also decaying in amplitude ; however if the Gaussian keeps matching the context as it evolves, it will be stimulated so that the activity is kept constant. If it stops matching the context, the Gaussian will so start to decay.

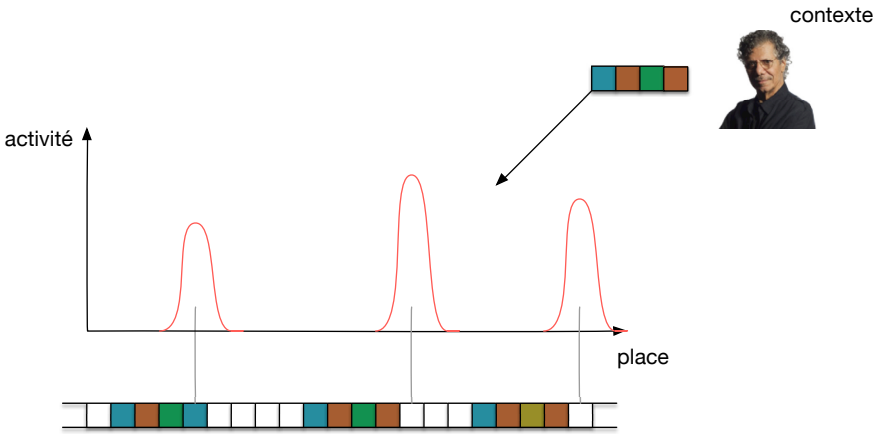


Figure 2.2: Activity profile after a hearing a new observation

This decaying Gaussian model allows the system to keep "formerly valid" solutions for a little moment, as the musical memory does not omit a short-term past immediatiely. Short-term valid solutions are so still eligible to be stimulated again by new observations. Basically, the activity of a memory state can be considered probabilistically as a *likelihood* of this place knowing the context ; for a more strict proof, please consider to the Bonnasse-Gahot report.

Recombination. SoMax bases itself on a short-term forecast of the context, which means that he uses the given context at the current time t to select the future state to play after the current one.
This can be understood as if it was sampling the context information and using each present slice of context to forecast the near future. By refreshing its activity profile, SoMax can choose the next event to be played by selecting the one which has the maximum activity. As SoMax chooses these events step-by-step, this routine is deeply reactive even if based on a short-term forecast.

2.3.2 Influences and dimensions

One of the advantage of the *activity* concept and its relation with the likelihood concept is that it gives the possibility to add several activity profiles regarding to different musical dimensions. This is what SoMax does, by cumulating three activity profiles : the *melodic* influence, the *harmonic* influence and the *self-listening*

(which is a little special).

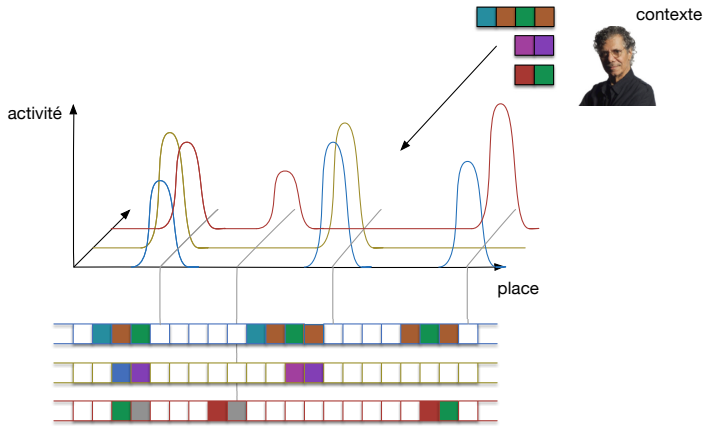


Figure 2.3: Different points of view on a musical memory.

For the melodic and harmonic influence, SoMax bases itself on the same musical memory but from different points of view, a melodic one and an harmonic one. Similarly, the observations of the external context are based on several dimensions, so that the places stimulated in the memory can be different regarding to the musical dimension.
Furthermore, the different points can be segmented in different ways ; for example, when you are using conductors, the melodic influence is separated by onsets and the harmonic influence is separated by pulse.

Then, given these two activity profiles, a weighted sum is done to get a global activity and then select a next state relevant regarding to the different dimensions. Basically, adjusting the weights is exactly what you do when you adjust the influence sliders in the player window. Weighting more the harmonic activity profile will so give to the harmonic dimension a bigger influence on the next state selection.

The *self-influence* point of view is a little special : it is not using the external context but itself to guide the improvisation, to keep consistency with the original musical memory. This is done by, at each generated state of the improvisation, listening to its own output to find states in the original memory that shares a common musical past and activating it in this specific point of view. This allows to assure some kind of stylistic consistency with the original memory, as there will always be a subsequence of the improvisation that can be found in the original memory.
This point of view is considered as its melodic and harmonic colleagues, so advantaging the self-listening will encourage the player to be consistent with the

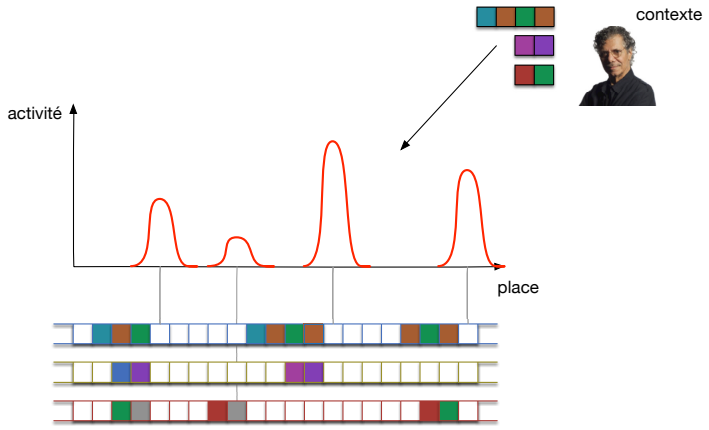


Figure 2.4: Summing the different weighted activites to get a global activity.

original memory rather than be listenting the external context.

2.3.3 Activity modulations

Once the global activity is obtained from the three points of view, it can be still modulated to guide the system and modelling the generated improvisation. From now two operations are implanted : the *taboo* system and the rythmic modulation.

Taboos. After refreshing its global activity profile, the system modulates the activity of the 8 previous states he played with the weights of the taboo table (see figure below).

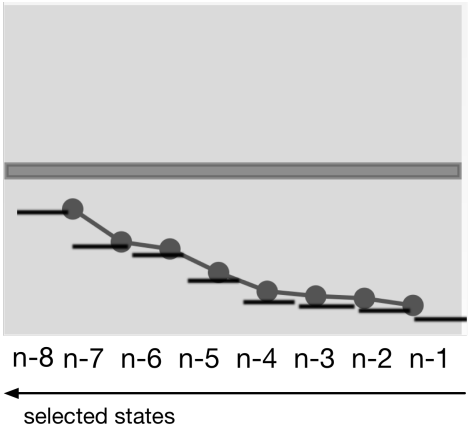


Figure 2.5: Classical taboo table

For example, the taboo presets whose table is displayed above strongly prohibites the recent few states played, to avoid the system looping or repeating the same state in a too short time. On the contrary, it is possible to set positive weights on the taboo table, to make the system loop as with the *loop 4* preset.

Rythmic modulation. It is also possible to modulate the global activity according to the rhythmic phase, and preferring the states of the memory at the same rhythmic phase as the current improvisation. This is done by modulating the global activity by a cosinus synchronous with the wanted improvisation time. There is also a selectivity parameter, which sharpens the cosine modulation function and so improve the selectivity.
You can able or disable the rhythmic modulation and its selectivity with the checkbox and the slider in the *Rythmic parameters* part in the player.

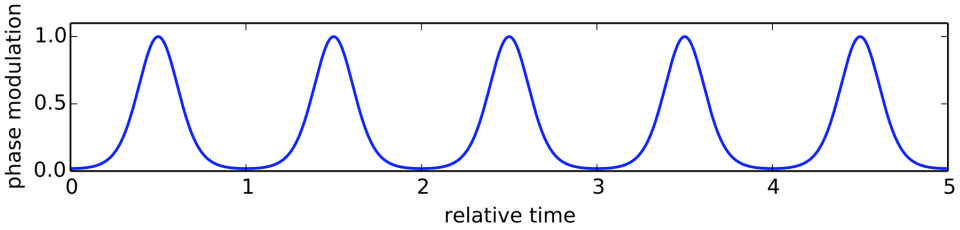


Figure 2.6: Phase modulation function

Jump. The last one is the jump command, which forces the system to move in another place of the memory. This is done by setting the activity of the cojoint state of the current one in the memory to 0, making it unlikely to be chosen.

2.3.4 Rythmic adjustments

Another important feature of SoMax is its capacity to align its improvisation to an external pulsation. Given a certain pulsation phase (constrained, from the musical memory or listened from the outside context), a jump of the system in another region of the memory can lead to rhythm phase breaks and a loss of the rhythmic feeling. The solution purposed here is a phase correction algorithm that adjust the original phase of the state aimed by the jump regarding to the actual one of the improvisation. This avoids the rhythm dislocation provoked by a phase disrupt and preserve the rhythmic feeling of the original memory.

You can able or disable this adjustment by checking the *adjust phase* box. The two numbers near the check box is the window where this adjustment is effective; if the difference between the unadjusted date and the adjusted date lies outside this temporal window, the phase adjustment mechanism will not apply.

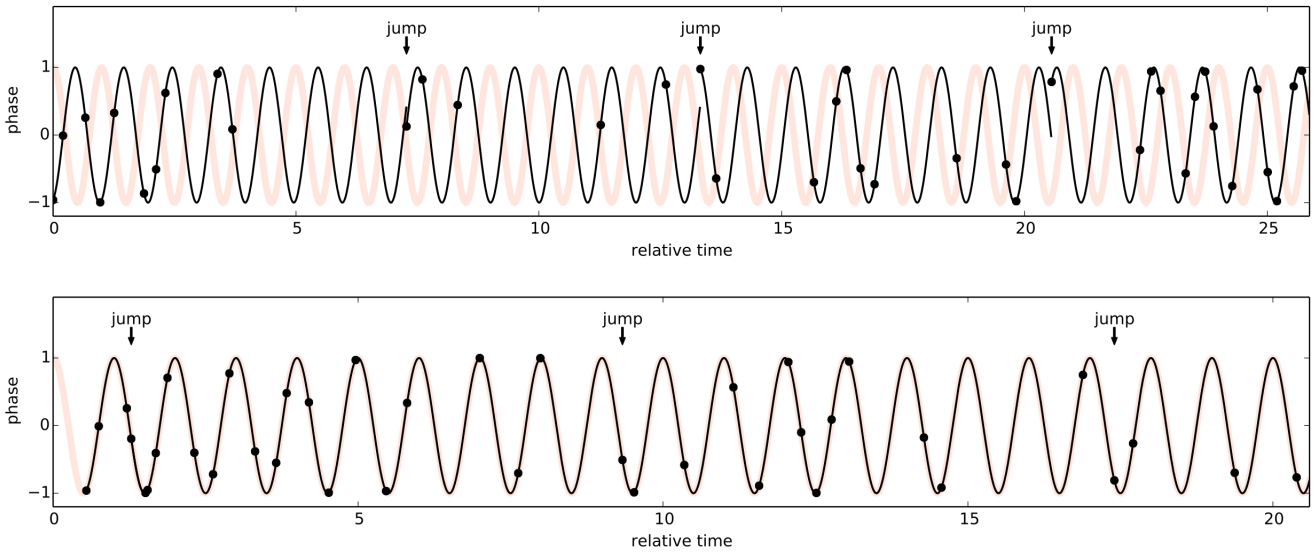


Figure 2.7: Phase adjustment mechanism

Chapter 3

Advanced

This part is made for the advanced users of SoMax, programmers who would like to use all the deepest possibilities of SoMax. SoMax is based on the dialog of a MaxMSP part managing all the realtime flow processing, and a Python core acting as a library that implements the high-level concepts of SoMax.
Once again, we will describe how it works with an ascending level of details ; first we will explain how to use the player as an object, secondly how the inside of the player works and finally the construction of the Python core of the player.

3.1 Modularity in SoMax

SoMax consists in three main patches, two conductors which have been described in a previous part and a player patch, called `soma.maxpat`.
It is important to notice that in fact the two conductors are high-level players that are just made to have an easier and quicker access to the possibilites of the player patch. It redirects audio/MIDI information and manages some of the parameters the player patch deals with ; however, if the possibilites of the conductor is too weak for you can use the player patch by itself.

`soma.maxpat` is an abstraction you can use as an unit improvisation object. Here are the different information it takes and the one it gives back (note that you have to initialize it with a name, to avoid namespace conflicts) :

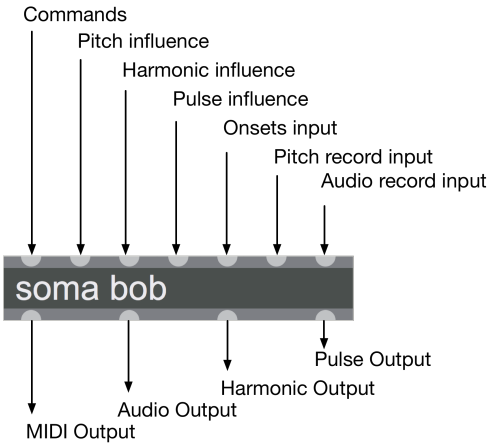


Figure 3.1: Inputs and outputs of the `somax` object

3.1.1 Inputs

Parameters. The first inlet is made to send miscanelleous commands to the SoMax object. Most of them are made to set diverse parameters of the player.

- *bang* : forces the player to output a state
- `start` : tells the player to start playing (assuming a corpus is first set)
- `stop` : tell the player to stop playing
- `load` [*symbol*] : tell the player to load the corpus *symbol*.

- `useOriginalBeatPhaseValue` [*boolean*] : activates the phase modulation and the phase adjustment features (see rhythmic modulation and phase adjustment).
- `phaseAdjustment` [*min*][*max*] : sets the phase adjustment window between *min* and *max* (see here).

- `bpm` [*value*] : sets the bpm to *value*.
- `useOriginalBpm` [*boolean*] : tells the player to use the original BPM of the memory.
- `autoBpm` [*boolean*] : activates the *auto adjust bpm* feature
- `autoBpmWindow` [*min*][*max*] : set the minimum and the maximum of the *auto adjust bpm* feature between *min* and *max*.

- `autoJump` [*boolean*] : activitates the *next state auto mod*, which forces the system to jump after a certain amount of continuity set by the parameter just below :
- `nextStateAutoMod` [*float*] is the parameter setting the maximum continuity of the improvisation regarding to the original memory, if the *next state auto mod* is activated.
- `listeningWeights` [*float*][*float*] [*float*] sets the respective weights of the *self-influence*, *melodic* influence, the *harmonic* influence. Note that the values, as long as they are positive, does not have a specific range as they are then normalized.
- `heldMode` [*boolean*] : makes the player waiting for *bang* messages to output its improvisation (notes or audio chunks).
- `color` [*float*][*float*][*float*][*float*] set the color of the player GUI (`RGBA` format).

Influences. The three next inputs are the influences of the player, respectively the *melodic* influence, the *harmonic* influence and the *pulse* influence. There is an additional *onset* input, whose role will also be explained.

- the *melodic influence* consists in MIDI formated messages ([*pitch channel velocity*]) each time a note of a melody occurs. For exemple, sending a message containing [*52 0 128*] will send a *E3* influence to the player.
- the *harmonic influence* consists in a *chroma table*, which is an array of twelve floats containing the "amount" from `0.` to `1.` of the chromatic notes, from C to B. In the case of MIDI, it is easy to analyse the present notes in a slice of time, and setting their respective value to 1. However, in the audio case, a chroma detection is necessary and so explains this harmonic message formatting.
  Please be careful! Even if the flow of harmonic information is countinuous, the information is segmented by the pulse of the player.

- the *pulse influence* is simply a succession of *bang* setting the pulsation of the player, that he will use as internal BPM and for the phase adjustments features.

In addition to these entries, on *onset* input is there. The onsets sent to this input are the onsets used for the phase adjustments features ; in the conductor these onsets are directly the onsets of the melodic channel (foreground channel in the case of MIDI, global flow in the case of audio). However, more specific applications could require something else, and so the onset input has been detached from the melodic input.

Record inputs. In the case where audio flows have to be recorded for a real-time corpus building, there are also two audio inputs : the first one, the *pitch record input*, will be used to extract the pitch, and the second one *audio record input* will be sliced and used as the audio material.
This distinction can be useful in a live context, where a precise but poor quality piezometric microphone can be used to track pitch, and another microphone of better quality is used to record the audio material. In the case where you just have one microphone, connect your input to both inlets.

3.1.2 Outputs

The `soma.maxpat` has also four outputs, two outputs being the musical outputs of the player and two being its information outputs.

- The first output of the player is a MIDI output, that can be directly linked to a `noteout` object in Max or be used as melodic influence for other `soma` players.
- The second output of the player is an Audio output, that can be directly linked to a `dac` object in Max or be used as audio material for other `soma` players.
- The two other outputs are made to be directly reinjected in other `soma` players, as harmonic influence and pulse influence.

With all these inputs and all these outputs, you can basically build a totally brand new `soma` players networks, with several outputs and inputs, and so fulfill your most secret desires.

3.2 Software architecture

We will now take a further look on how the `soma` player is built.

SoMax is an hybrid environment, built with a part in MaxMSP, which manages audio streams, MIDI streams, timing and interfacing, and a core library in Python implemanting the main concepts of SoMax.

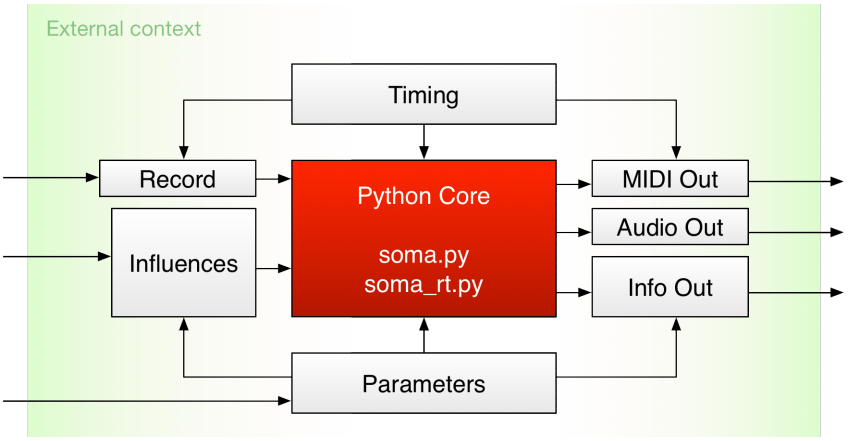


Figure 3.2: Inputs and outputs of the `somax` object

The code of the `soma.maxpat` is organized this way. We will now have a detailed look of all this part. In this code, there is very important abstraction called `pycore`, which in fact call Python core's methods.

Parameters : This section takes care of the parameters displayed to the user or available from the outside. It gathers all the information regarding the phase adjustments, the weights, the corpus, the auto jump and taboos, and sends its information to the Python Core.
Please note that the taboo presets are saved in specific tables, that you can feel free to adapt, or to add some others.

Influences : This section gathers all the influence signals from the outside context and format them to send it to the Python core. In addition to binding the pure information for the occuring date of the event, there is some specific formatting according to the point of view.
Regarding to the melodic influence, it eventually transposes it. With the harmonic influences, it also eventually transposes it, but also sends it periodically at every puslation of the player (in contrary to the melodic influence, which is rhythmically free).
Regarding to the self-influence, it first get the label of the played state, before

sending it to the Python core.

Regarding to the rythmic adjustments, the onsets detected from the outside are sent as external events to the Python core, and a BPM adjust command is also sent with some more information required by the Python core (see the `auto_adjust_bpm` function).

Record : The record part is quite direct : it

Time : All the player events are synchronized on a custom `transport`, identified with the ID of the patch. The BPM of this transport can be fixed, taken from the original corpus (which can change at every chosen state) or synchronized from an external pulse (which does not immediately change by first by a leaky integration, for a smoother change). The pulse of this transport is the one returned by the player to the external world.

Core : The python core receive all the information received and managed by the MaxMSP environment and returns all the information the player needs to generate music : states to play, musical contents, new BPMs... all the details regarding this Python core are given in the Python core part.

Process : As said in here, SoMax bases itself on a short-term forecast of the context, which means that he uses the given context at the current time t to select the future state to play after the current one.
This is based on a loop : when an event is selected to be played, it is then waiting for the right time to be played and, when the time has come, a new query is sent to the Python core and the played event is sent to the self-influence. The *process* part of the code contains this whole loop.

Midi out : In the case where the musical memory is some MIDI content, the Python core will output SoMax encoded information, that the part of this code will resitute as MIDI content ready to be exploited. This part also manages the panic message that is accessible from the user interface and the transposition part.

Audio out : In the vase where the musical memory is audio content, the Python core will output the states to play as intervals in the audio buffer. The audio restitution is based on two buffers : a reference buffer (`main_buff`) which contains the whole file, and a state buffer (`tmp_buff`) which is loaded with the state to play. The buffer is played by Ircam's `supervp`, to perform the realtime time manipulation (if the current BPM is different from the original one).

Output & Info : this part gives the user, on the interface, some feedback of the notes and states played (via the keyboard and the state position slider). It also returns the harmonic context of the state to the external world, which comes from the musical memory.

3.3 Python core

We will now enter the Python script which implements all the SoMax generation algorithm. The interface between MaxMSP and Python is made thanks to the `py` external by Thomas Grill [1].
This code is splitted in two scripts : the first script, `somax_rt.py` mainly links the MaxMSP to the real SoMax architecture contained in the second script, `soma.py`. As the major part of `somax_rt.py` just calls function of the main script, we will first describe the `soma.py` script.

3.3.1 Overview of the code

The `soma.py` script contains the main Python object, `Player`, which has the leading role in the generation of the improvisation. The `Player` class contains the musical memory and its differents points of view, each embodied in a general abstract class called `StreamView`. It also carries the `new_event` function, which carries the step-by-step generation of the events, and the phase adjustments manipulations.

A point of view on the memory is stored in an abstract object called `StreamView`. This object is responsible for the correct data extraction from the corpus file according to its affected musical dimension, its n-gram representation and its activity profile. It relies on two other subobjects : the `ActivitySpace` object which takes care of the activity profile and its updates over time, and the `KappaSpace` which containts all the information relative to the memory content and its n-gram representation.

For example, the influences from the outside context are directly stimulating the `ActivitySpace` of its corresponding `StreamView`, which asks the `KappaSpace` the stimulated places and updates its activity profile.
When online recording is done, some new states are added to the `StreamView`'s data and the n-gram of the `KappaSpace` is updated.
When a new state is queried at a given date, the `Player` ask all its `StreamViews` to give it back their activity profiles, makes the weighted sum of it all, apply modulations and jump and finally returns the appropriate state and the appropriate time.

The theoric SoMax concepts are based on a continuous model, which can't be implemanted on a computer. The activity profile is so modelized as arrays of (*date*, *value*) pairs. When the activity profile is updated, the dates are translated and the values are exponentially decaying, until the value reaches an extinction

[1] http://grrrr.org/research/software/py/

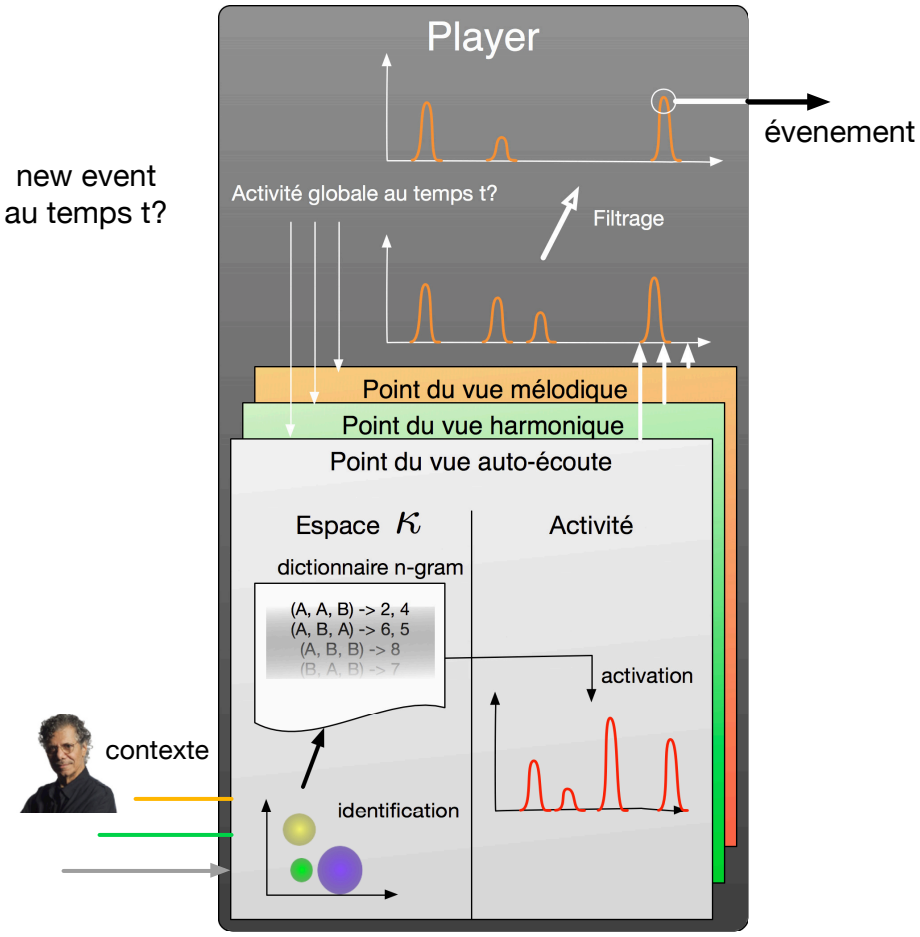


Figure 3.3: Inputs and outputs of the `somax` object

threshold, and the corresponding pair is deleted.

If we now come back to the `somax_rt.py` script which relies the MaxMSP environment and the `Player` class, we can now explain that it uniquely consists in an object called `SomaxPlayer` which possesses a `Player` instance, and takes care of properly translating the outside queries to the `Player` object.
We can though notice the `parse_new_slice`, which adds to the `Player` memory new material in the case of real-time recording, and the `update_buffer` function which takes care of updating the buffers for the MaxMSP environment.

3.3.2 Detailed documentation

Here is now the exact documentation of the `soma` library. This gathers classes defined in both `soma.py` and `soma_rt.py`.

SomaxPlayer

Attributes:

- **bob** : the instance of `Player` used to generate the improvisation
- s_ngram_size : size of the self-listenting n-gram used by the player
- m_ngram_size : size of themelodic n-gram used by the player
- h_ngram_size : size of the harmonic n-gram used by the player
- crossfade : length of the crossfade used to concatenate the states in buffer
- sr_str : suffix of the messages sent to the Max environment (you normally don't have to move that)
- shorten_rests

Methods:

Note that below, the *srid* argument is the ID of the `soma.maxpat` instance, made to label the messages in the environment (in the case of several players).

- **new_state** *float*: date *float*:srid
  returns the new event obtainted by calling the `new_state` function of the `Player`.
- **load** *str*:name
  tells the `Player` to load the corpus called *name*.
- **self_influence** *float*:date *int*:pitch
  makes the self-influence `StreamView` of the `Player` register the *pitch* event occured at improvisation time *date*.
- **pitch_influence** *float*:date *int*:pitch
  makes the melodic `StreamView` of the `Player` register the *pitch* event occured at improvisation time *date*.
- **harmonic_influence** *float*:date 12×*float*:chromas makes the harmonic `StreamView` of the `Player` register the *chromas* event occured at improvisation time *date*.
- **start** *float*:date *int*:event *float*:srid resets all the `StreamView` of the `Player` and the internal history, and finally asks the `Player` a new state.

- reset
  makes a new `Player` instance.
- jump
  ask the `Player` to jump.
- record_external_event *float*:date
  add the date of a recent external event to the `Player` registry. We don't care about the nature of the event, as this information is only used for rhyhtmic adjustments.
- set_tau_mem_decay *float*:tau
  sets the tau_mem_decay attribute of all `ActivityPattern` see `ActivityPattern` documentation.
- set_activation_threshold *float*:activation_threshold
  sets the activation_threshold attribute of all `ActivityPattern`, see `ActivityPattern` documentation.
- set_weights 3×*float*:weights
  sets respectively the *self-influence* weight, the *melodic* weight and the *harmonic* weight of the player's `StreamView`.
- set_phase_ref *float*:phase_ref

- set_gamma *float*:gamma sets the gamma attribute of the `Player`. See `Player` documentation.
- set_adjust_phase *boolean*:adjust_phase
  enables the phase adjustment of the `Player`. See `Player` documentation.
- set_adjust_phase_w 2×*float*:phase_w
  sets the phase adjustment window of the `Player`. See `Player` documentation.
- set_taboo_params 8×*float*:phase_w
  sets the taboo coefficients of the `Player`.
- adjust_bpm *float*:current_bpm *float*:min_bpm *float*:max_bpm *float*:date
  tells the `Player` to re-adjust BPM regarding to the recent events from the *current_bpm* and within [*min_bpm*, *max_bpm*]. It then outputs the BPM, to be given to the global transport.
- set_w_length *float*:w_length
  sets the BPM adjustment window of the `Player`. See `Player` documentation.
- set_next_state *float*:next_state
  sets the next state modulation amount of the `Player`. See `Player` documentation.

- set_auto_jump_mode *boolean*:auto_jump_mode
  enables the auto-jump mode of the `Player`. See `Player` documentation.
- set_tau_next_state *float*:tau_next_state
  sets the maximum continuity tolerance for the auto-jump mode of the `Player`. See `Player` documentation.
- set_verbose *boolean*:verbose enables the verbose mode the `Player`, which will output a lot of informations during the process in the Max window.
- get_label *int*:index *float*:srid outputs the label of the *index*th state of the memory.
- get_notes *int*:index *float*:srid
  outputs the MIDI musical content of the *index*th state of the memory. Outputs something only if the memory is MIDI.
- get_state_ticks_duration *int*:event *float*:srid
  outputs the duration of the *index*th state of the memory in ticks.
- get_bpm *int*:event *float*:srid
  outputs the bpm of the *index*th state of the memory.
- get_hctxt *int*:event *float*:srid
  outputs the harmonic context of the *index*th state of the memory as a 12-chroma array.
- parse_new_slice *SoMax entry*:slice
  adds the new slice to the `Player memory`. For additional information of the SoMax entry formatting, please go to the corpus construction section.
- save_as *string*:name tells the `Player` to save its actual memory in a file named *name*.
- set_crossfade *float*:crossfade sets the crossfade duration in ms, used to concatenate the conjoint states.
- update_buffer *string*:main_name *string*:temp_name *int*:event *float*:tb_pos *float*:t_length
  internal routines made to update the temporary buffer from the main buffer containing the whole file.

Player

Attributes:

- **s_l** : Self-influence `StreamView` object
- **m_l** : Melodic influence `StreamView` object

- **h_l** : Harmonic influence `StreamView` object
- s_w : Self-influence activity weight
- m_w: Melodic influence activity weight
- h_w: Harmonic influence activity weight
- prepare_to_jump : is the system about to jump at the next step?
- phase_ref : `Player`'s phase reference, used for phase adjustments calculations
- gamma : selectivity of the phase modulation feature.
- adjust_phase : is the phase adjustment mode enable
- phase_adjustment_w : phase adjustment window
- w_length : bpm adjustment window
- taboo_length : length of the taboo table
- b_step : number of states per taboo value
- **taboo_params** : value of taboo table
- next_state : conjoint state modulation during the new event selection
- auto_jump_mode : is auto jump mode enabled
- tau_next_state : maximum continuity for the improvisation generation if the auto jump is enabled
- last_jump : last event before the previous jump

Methods:

- load_mem *string*:corpus
  tells the `StreamView` objects of the `Player` to load the *corpus* memory. The files can be different regarding to the `StreamView` ; go to the corpus construction section.
- save_mem *string*:name
  saves the `StreamView`'s memory in new files named *name*.
- new_event *string*:date
  asks the system the most relevant state of the memory around the date. Outputs a pair *(state, date)* ; note that the date can be different if the phase adjustment is enabled.
- jump
  sets the activity of the present conjoint state in the memory very low, to force the system to jump.

- adjust_bpm *float*:current_bpm *float*:min_bpm *float*:max_bpm *float*:date asks the best bpm estimation regarding to the past recorded external events.

StreamView

Init :

Attributes:

- **k_self_listening**: the `KappaSpace` where the memory knowledge is
- **activity** : the `ActivitySpace` where the activity profile is
- **mm_data** : the raw content of the memory
- **kappa_event_history** : history of the events received by the context's influence
- compute_pre_rep : the function used to select the `StreamView`'s relevant data from the raw memory. Depending to the `StreamView`'s nature, this function is one of the `compute_pre_rep` methods described below
- event_list : list of the dates of the memory states
- rep_list : list of the labels of the memory states
- event_history : history of the states played during improvisation
- external_events : history of the external_events recorded and used for BPM adjustments. Only used by the self-listening `StreamView`.
- id : an arbitrary number identifying the stream view

Methods:

- load_mem *string*:name
  load the file named *name* and stores the raw data into its memory.
- init_stream_view:
  takes the raw data, takes the relevant data from it and ask the `KappaSpace` to build the n-gram.
- update_memory *SoMaxEntry*:new_slice
  updates its memory by adding the *new_slice*, and updates its `KappaSpace`.
- compute_pre_rep_pitch *int*:n
  takes the pitch information from the nth event of the raw content of memory
- compute_pre_rep_chroma *int*:n
  takes the chroma information from the nth event of the raw content of memory
- update_player_activity *float*: new_date
  asks the `ActivityPattern` of the `StreamView` to updates its activity profile at the date *new_date*.

- record_external_event *float*: date
  Inserts the *date* of an external event into the external events history. Only used by the self-influence `StreamView`.

KappaSpace

Attributes:

- **ke_map** : the dictionary containing every occurrences dates of the subsequences found in the memory. This dictionary is the representation of the n-gram.
- ngram_size : size of the n-gram
- kappa_activation :
- node_specificity :
- kappa_activity_threshold

Methods:

- kappa_activation_pitch *int*: pitch
- kappa_activation_som_chr: 12×*float*: chromas
- map_kappa_to_events *list*: event_list *list*: rep_list

Activity Space

Attributes:

- date : current date of the `ActivitySpace`
- zeta : list of the dates of activity stimulation
- value : value of activity peaks
- tau_mem_decay : decay time of the peaks
- t_width :
- extinction_threshold: threshold below which an activity stimulation disappears.
- available : is `ActivitySpace` available for update

Methods:

- update_activity *float*:new_date
  update the activity profile to the new_date, which means updating the *zeta* list and the *value* list
- clean_up:
  deletes the values in activity profile whose activity is below the extinction threshold.
- insert *list*:new_zeta_list *list* new_value_list
  inserts the new events (*zeta*, *value*) splitted up in two lists to the activity profile.
- get_activity
  returns the current activity profile

3.4 SoMax corpus file format.

SoMax is supposed to work with `JSON`[2] files supposed to have a specific structure. Originally built with Matlab, a whole corpus construction with Python has been conceived to allow the easy construction and specification of these corpus.
This algorithm is also made to be compatible with further developments of SoMax and more generally for the OMax family, and is so much more flexible that the current SoMax version ; though a good variety of corpus can be built with this new library.

SoMax understands `JSON` file, which is in fact a dictionary, with a very specific structure described below :

- `name` : name of the corpus
- `size` : number of states of the corpus
- `type` : not used
- `typeID` : `MIDI` or `Audio` (also not used)
- `data` : main dictionary of the memory states
  - `beat` : tempo-relative information as an array [*beat*, `tempo`, 0, 0]
  - `extras` : free information used for the harmonic background, most of the time chroma vectors (but where also supposed to contain Mel Frequence Cepstral Coefficients)

[2] JavaScript Object Notation

- `notes` : in the case of a MIDI files, the corresponding notes of the event. A note is itself a dictionary containing :
  - `note` : An array [*pitch velocity channel*]
  - `time` : An arrat [*relative time duration*]
- `seg` : additional index, in the case of a corpus containing several files
- `slice` : the entry containing the label of the state, as a vector [*label*, 0]. In traditional uses, 0 to 127 represents a single note, 128 to 139 represents a chord and 140 represents silence
- `time` : time properties of the state, as an array [*time*, `duration`] in milliseconds]

Please note that the `data` entry must contain an all empty state as state 0, permitting the SoMax Player to initialize.

3.5 Corpus construction library

Associated with the Somax package is a corpus construction library programmed with Python.

This library provides high-level methods to construct standard corpus from audio files or a MIDI files, which can be interactive thus allowing the user to quickly parametrize the different processes. The corpus construction is also object-oriented, permitting easy modifications and extensions of the building routines and their dynamic use in the Python routine engine.

3.5.1 Basic construction

The corpus construction routine can be launched by just executing the `build.py` Python file (in Terminal):

```
python build.py <file path>
```

where you can easily put the file path by dragging and dropping the file into the terminal window.

The path can lead to a single file, or to a folder ; in the case of the file, the program will scan its folder to automatically find the additionnal files (named *corpusname_suffix*).
In the case of the folder, the program will concatenate all the found files to compute the final corpus file. If additional files are provided, please be careful that all the main files have their corresponding extension files. If it doesn't,

the program will ask you whether to delete the extension, or to compute the corresponding extension algorithm on the main file.

In every case, the files outputed by the algorithm will be placed in the corpus folder of SoMax.

3.5.2 Additional options

Several options have available in addition to this simple corpus construction.

Verbose mode. You can activate the verbose mode of the construction algorithm by simply adding `-v` before the corpus path :

```
python build.py -v <file path>
```

This allows to get back some information about the construction routine.

Output mode Similarly, you can select the output location of the corpus file by adding the `-o` option :

```
python build.py -o <file path>
```

After the calculation, the program will ask you the location where you want to output the corpus file.

Interactive mode Last but not least, you can activate the interaction mode by adding the *-i* extension. The interactive mode allows some kind of intermediate level between the high-level model method and the object-oriented dynamic approach, where the program will ask you step by step if you want to change anything during the average routine. We will detail this mode by giving the complete description of library below.

3.5.3 Overview of the library

The corpus construction library is mainly based on two main type of objects :

- a big object, called `CorpusBuilder`, in charge of gathering all the relevant files of the corpus, assigning them a given process and then executing them
- and a set of objects called *operations*, childs of an abstract object called `MetaOp`, in charge of implemanting a given treatment to a set of file and outputing the corresponding corpus file.

The CorpusBuilder. The Corpus Builder object, which is initialized with a corpus path, optionally a corpus name, and several key arguments for different parameters, has two main tasks.
When initialized, it will scan the given corpus path (file or folder) to build a dictionnary, called `ops`, which lists for every extension type a corresponding operation and ordered list of the file to be analyzed. The `ops` dictionary has the extension of the found files as key (`h`, `m` for example) and a tuple `(operation, file paths)` as value. For example :
`{'': (OpSomaxStandard, 'corpus.mid'), 'h':(OpSomaxHarmonic, 'corpus_h.mid' 'm':(OpSomaxMelodic, 'corpus_m.mid')}`

These default operations are contained in the `callback_dic` attribute, which contain the default operations applied to a given extension. The `build_corpus` method then takes in charge to apply the corresponding operation to the given files, and then computing the corpus.

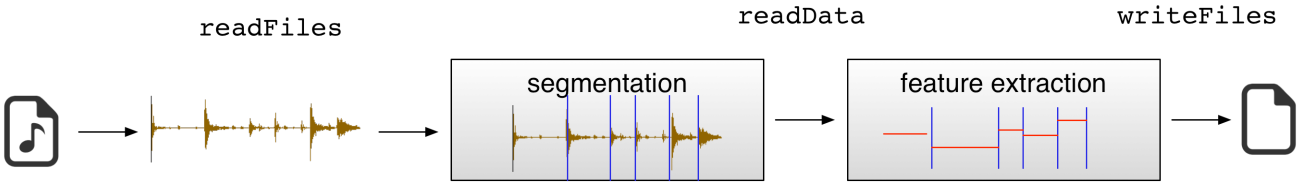


Figure 3.4: Standard corpus construction routine, witch `SegmenationOp` functions

The operations. The operations are the units in charge of generating the corpus file. All the operations units are based on an strong heritage nature, to permit easy modifications of a part of the process by overloading methods or attributes. The main abstract object, `MetaOp`, is a very simple object initialized with just a corpus name and containing a single method, named `process`. This method just takes the output location of the resulting files. This very simple object permits, for example, to generate content from scratch without needing an input file.
This abstract object also has a `setParameter` function, which is used for example in the interactive mode to set parameters from outside the operation, insuring a correct format a giving access to only relevant attributes to the user.

For more conventional operations, based on the segmentation and feature extraction of an input file, there is a higher level abstraction called `SegmentationOp`. Contrary to `MetaOp`, its initialization needs a list of input files and the generic `process` function has to pass three subfunctions :

- `readFiles`, which extracts the raw data from the input files,
- `readData`, which takes the raw data and extracts from it the informations needed in the corpus file

- `writeFiles`, which outputs the corresponding files at a given location

Furthermore, this abstraction has a `admitted_extensions` list of extension which defines all the files that the operation can read.

Existing operations. The casual corpus construction routines of SoMax have been implemented using these objects, as childs of the `SegmentationOp` object. These routines can read MIDI and audio files, and outputs files formatted in the Somax `JSON` structure.

The main operation, `OpSomaxStandard`, basically segments the file into states (in the case of several files, it concatenates them), extracts features from these states by labelling them by their pitch features and adds additional information as timing and chroma context (for more information see SoMax corpus file format section). Note that, in the case of MIDI files, the channels used for labelling are the *foreground channels*, while the channels used for chroma context are the *background channels*, that you can configure (see below).
The segmentation mode is here quite important. Several modes are available : the default *onsets* mode segments the files with onset detection, whereas the *beats* mode segments the files with beat detection. The beat mode is more appropriate in the case of pulsed musical context, and the onset is more in the case of free context. An other one, called *free* mode, basically segments the file at regular intervals.

Two other operations implement the additional standard operations for the harmonic dimension and the melodic dimension : `OpSomaxHarmonic` and `OpSomaxMelodic`. They both inherit `OpSomaxStandard`, as they globally are based on this class with just a few difference. *OpSomaxHarmonic*, to be more adequate on the detection of harmonic context, segments by default by beats, and does not label its information as the `*_h.json` files are used by SoMax only to get the `extra` entries. It is even simpler for the `OpSomaxMelodic`, whose only difference with `OpSomaxStandard` is that the *mod12* mode, which labels the states by relative pitch. Both are simple example of how to quickly use the operations' inheritance model to customize the corpus construction.

There is the list of the parameters of the three operations described above that you can configure, for example via the interactive mode.

- `fgChannels` : in the case of a MIDI segmentation, choose the foreground channels used for the labelling. Has to be an array of intergers from 1 to 16, as `[1 2 3]`.
- `bgChannels` : in the case of a MIDI segmentation, choose the background channels used for the computation of the chroma context. Has to be an array

of intergers from 1 to 16, as `[1 2 3]`.

- `mod12` : in the case of a MIDI segmentation, tells if the labelling of the information is octave-sensitive (from 0 to 140) or relative (from 0 to 12). Has to be a boolean. For more information on the Somax standard labelling system, see here.
- `segtype` : in the case of an audio segmentation, sets the segmentation mode among `onsets`, `beats` and `free`.
- `usebeats` : in the case of an audio segmentation, tells the operation to use or not a beat detection algorithm to detect beats and tempo of the file for `beat` entry of the file. If not, sets the tempo to 120 and places a beat every 500 ms. Has to be a boolean.
- `freeInterval` : in the case of an audio segmentation, sets the time interval of the *free* segmentation mode. Has to be a float, and in seconds.

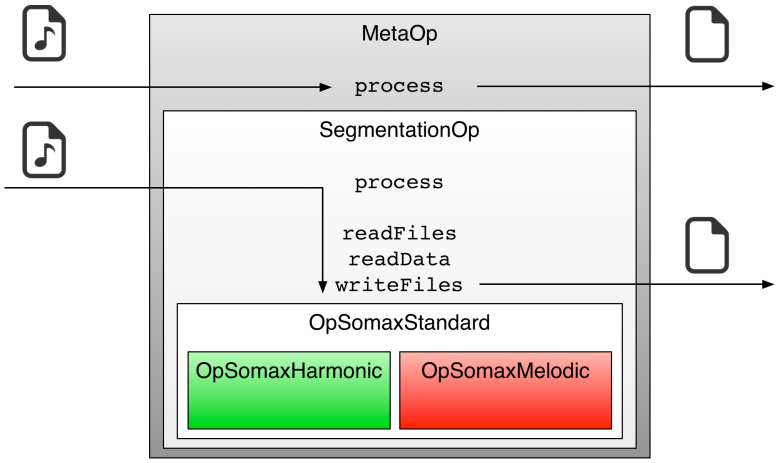


Figure 3.5: Inheritance schema for currently implanted operations