AudioGuide

By Ben Hackbarth, Norbert Schnell, Philippe Esling, Diemo Schwarz.

Contents

1 Installation

Here is a list of the resources that AudioGuide requires on your computer. If you run OSX 10.6 or higher[1].

ircamdescriptors AudioGuide uses IRCAM's descriptor analysis binary (Please note, I am not permitted to distribute this binary). When you run Audioguide the first time, a nice lady will ask you to enter the path of the descriptor binary.

pysdif First download the sdif library and configure, make and install. Then download my patched version of the pysdif module. 'cd' into the directory when unzipped and run: 'python2.7 setup.py install'. You're now setup to use pysdif in python2.7.

numpy Most python2.7 installations come with numpy, a numerical computation module. Upgrading to the latest python2.7 should get you there. If you don't have it, you can download the source code or a binary installer here.

csound Needed only if you would like Audioguide to automatically render concatenations (which you probably do). Download an installer from here.

2 Quick Start

Using Audioguide comes down to interacting with two python scripts in the audioGuide folder. One script does segmentation of corpus soundfiles. The other performs concatenation based on the variables found in a options file. While you do not need to know how to write Python code in order to use AudioGuide, it is not a bad idea to know some Python basics since the variables in the options file for concatenation is written is Python's syntax.

The reason that segmentation and concatenation are separated is that I find is useful to fine-tune the segmentation of corpus sounds *before* using them in a concatenation. Sound segmentation is a difficult technical problem and should remain conceptually and aesthetically open-ended.

[1]python2.7 comes with numpy and this AudioGuide distribution comes with a precompiled pysdif. So you will only need the IRCAmn binary and csound.

I have yet to find an algorithm that does not require adjustments based on the nature of the sound in question and the intended result.

2.1 Segmenting Corpus Soundfiles

Note: If you only want to use folders of sounds that have been pre-segmented into individual files, you can skip[2] the *Segmenting Corpus Soundfiles* section and proceed to the concatenation section.

The script you use to segment your corpus files with a script called 'segmentSf.py'. The output of segmentSf.py is a textfile which denotes the start and stop times of autonomous sound segments in a continuous audiofile. Once you find a segmentation that you're happy with, you don't need to keep running this script. Whats more, you do not *need* to use 'segmentSf.py' if you do not want to – instead you could:

1. use whole soundfiles as segments
2. create segmentation files by hand
3. create segmentation files with other software as long the textfile is written in the same format as AudioGuide's.

To segment a corpus file, 'cd' into the audioGuide-1.02 folder and run the following command:

```
$ > ./segmentSf.py examples/lachenmann.aiff
```

Audioguide will think for a second, and then output the following data detailing the segmentation of this audiofile:

```
———————————————————— Segment Soundfile ————————————————————
   Creating segmentation label file for lachenmann.aiff...
              Evaluating Target lachenmann.aiff from 0.00 sec -> 64.65 sec
 100% |=======================================================================|
   Done.

   Threshold dB: -40.000000      rise: 1.300000 Offset dB: -54.619647

   AudioGuide found 132 segments
```

As a result of running this python script, Audioguide wrote a textfile called examples/lachenmann.aiff.txt. In it are 132 segments obtained using a triggering threshold of -40 dB, a rise ratio of 1.3 and a offset dB value of -54 (you can read more about how to alter these values

[2]But make sure tell AudioGuide not to search for segmentation textfiles by setting the corpus attribute wholeFIle=True. See the Manipulating How Directories Are Read subsection of the CORPUS options section for more info.

and their effect in the subsequent section).

2.2 Concatenating

Next you call the concatenate script concatenateSf.py with an AudioGuide options file as the first (and only) argument.

To run one of the examples in the examples directory, run the following command inside the audioGuide-1.02 directory:

```
./concatenateSf.py examples/01-simpleSelection.py
```

..which will use the options contained in ‘examples/01-simpleSelection.py’ to parameterize the concatenative algorithm. In this file you specify the target sound, the corpus sounds, and lots of other options that parameterize the concatenative process. The ‘concatenateSf.py’ script will perform the following operations in order:

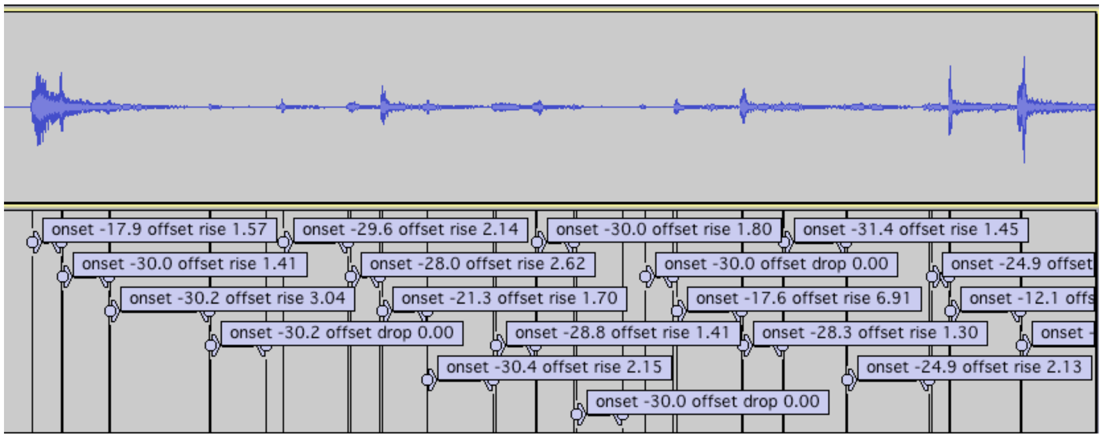
1. Run an ircamdescriptor analysis of the soundfile in the 4.1 TARGET variable[3].
2. Segment the target sound according to your options file. An Audacity-style labels file is created in a file called ‘tgtLabels.txt’ in the output directory.
3. Run an ircamdescriptor analysis of the soundfiles in the CORPUS variable (only the first time each of these files are used).
4. (If you’ve specified them) Remove corpus segment according to descriptor limitations (Nothing above a certain pitch, nothing below a certain dynamic, etc.).
5. Normalize target and corpus descriptor data according to you options file.
6. Go through each target segment one by one. Select corpus segment(s) to match each target segment according to the descriptors and search passes in the SEARCH variable of your options file. Control over the layering and superimposition of corpus sounds is specified in the SUPERIMPOSE variable.
7. Write selected segments to a sound score called ‘csoundScore.txt’ in the output directory.
8. If you have csound, ‘csoundScore.txt’ is rendered with audioguide/scripts/csoundRender.orc orchestra to create an audiofile called ‘output/output.aiff’.

[3]An SDIF analysis is only done once – subsequent usages of a soundfile simply read SDIF data from disk. Analysis files are stored in a directory called ‘data_json/’ in the audioguide folder. This directory can become quite large since these SDIF files are quite large in size. Removing this folder will cause all SDIF files to be recomputed.

9. If you have csound, automatic playback of ‘output.aiff’ at the command line. Works on OS X (afplay) and Linux (aplay).

3 Corpus Segmentation

Audioguide uses a labeling format identical to that of the soundfile editor Audacity. So, to examine your segments, open lachenmann.aiff in Audcaity, then import labels and select ‘examples/lachenmann.aiff.txt’.



onset -17.9 offset rise 1.57
onset -29.6 offset rise 2.14
onset -30.0 offset rise 1.80
onset -31.4 offset rise 1.45
onset -30.0 offset rise 1.41
onset -28.0 offset rise 2.62
onset -30.0 offset drop 0.00
onset -24.9 offset
onset -30.2 offset rise 3.04
onset -21.3 offset rise 1.70
onset -17.6 offset rise 6.91
onset -12.1 offs
onset -30.2 offset drop 0.00
onset -28.8 offset rise 1.41
onset -28.3 offset rise 1.30
onset
onset -30.4 offset rise 2.15
onset -24.9 offset rise 2.13
onset -30.0 offset drop 0.00

These segmentation textfiles document one segment per line. here is an example line: ‘1.50 2.13 onset -12.9 offset rise 1.33’.

Everything after startSec and endSec is not required by AudioGuide, but give you information about the segmentation logic. These fields correspond to: startSec endSec onset thresholdValue offset offsetMethod methodValue.

.. and indicate ..

3.1 Onset Detection

thresholdValue: the value of the threshold which triggered this segment’s onset. it is in negative dB where -100 dB is very soft and -6 dB is very loud. you can change the threshold level that triggers onsets by passing the segmentSf.py script a ‘-t’ flag:

```
./segmentSf.py −t −30 soundfilename.wav
```

...sets the threshold to -30. it will produce less onsets then -40 (the default).

3.2 Offset Detection

AudioGuide's soundfile segmentation uses two methods for creating an offset (an offset means to end a currently active sound segment). The first method is simple – if the amplitude of a segment drops below a certain dB threshold, called 'drop' and can be changed from its default value using the '-d' flag.

```
./segmentSf.py -d 4 soundfilename.wav
```

... which changes the drop dB value to 4 dB above the minimum amplitude found in the entire soundfile.

The second method is more complicated: if the amplitude of the sound is louder that the previous value and the ratio of the current value over the previous value is above the 'rise ratio'. This is a very useful construct: imagine that you are in a current sound segment, but the soundfile suddenly gets much louder, and you'd like to end the current segment so that you may start a new one which reflects this change. You can override the default rise ratio (1.1) using the '-r' flag:

```
./segmentSf.py -r 4 soundfilename.wav
```

...a value of 1.1 will be quite sensitive to changes in amplitude. A value of 4 will be less sensitive - a larger crescendo is needed to turn off a currently active segment.

Note that to segment a whole directory of soundfiles, you may use wildcard characters in the bash shell, as in:

```
./segmentSf.py mydir/*.aiff # create a segmentation file for each aiff file located in
  mydir/
```

4 The Concatenate Options File

The options file used by the concatenate script if a python file that defines a bunch of variables. Most variables are changed with simple assignments using the '=' symbol. For instance, to change the path of the csound output sound file, write the following in your options.py file:

```
CSOUND_RENDER_FILEPATH = '/path/to/the/file/i/want.aiff' # sets the path of the csound
  output aiff file
LOG_FILEPATH = None # None tells AudioGuide to not make a log.txt file.
```

However, there are five objects that are written into the options file as well – 'tsf', 'csf', 'spass', 'd', 'si'. These objects take required parameters and also take keyword arguments. The following sections describe the object-style variables.

4.1 The TARGET Variable

The TARGET variable is written as a ‘tsf’ object which requires the path to a soundfile and also takes the following optional keyword arguments:

```
tsf('path', start=0, end=file-length, thresh=-20, rise=1.3, scaleDb=0, minSegLen=0.05,
  maxSegLen=4)
```

start The time in seconds to start reading the target soundfile.

end The time in seconds to stop reading the target soundfile.

scaleDb Applies an amplitude change to the target sound. by default, it is 0, yielding no change. -6 = twice as soft. The target’s amplitude will usually affect concatenation: the louder the target, the more corpus sounds can be composted to approximate it energy.

thresh Set the threshold for segmentation: a value in negative dB. The lower the value the soft the target’s amplitude can be in order to trigger a selection from the corpus. So, -12 yields fewer corpus selections that -24.

rise

minSegLen The minimum duration in seconds of a target segment.

maxSegLen The maximum duration in seconds of a target segment.

```
TARGET = tsf('cage.aiff') # uses the whole soundfile at its given amplitude
TARGET = tsf('cage.aiff', start=5, end=7, scaleDb=6) # only use seconds 5-7 of cage.
 aiff at double the amplitude.
```

4.2 The CORPUS Variable

The CORPUS variable is defined as a list of ‘csf’ objects which require a path to a soundfile OR a directory. File paths and/or directory paths may be full paths or relative paths to the location of the options file you’re using. A ‘csf’ object takes the following optional keyword arguments:

```
csf('path', start=0, end=file-length, includeTimes=[], excludeTimes=[], limit=[],
 wholeFile=False, recursive=True, includeStr=None, excludeStr=None, onsetLen=0.01,
 offsetLen=30, transMethod=None, transQuantize=0, allowRepetition=True,
 restrictRepetition=0.1, restrictOverlaps=None, restrictInTime=None, scaleDb=0,
 superimposeRule=None, segmentationFile=None, segmentationExtension='.txt')
```

The simplest way to include a soundfile in your corpus is to use its path as the first argument of the 'csf' object:

```
CORPUS = [csf("lachenmann.aiff")] # will search for a segmentation file called
 lachenmann.aiff.txt and add all of its segments to the corpus
```

The simplest way to include a directory of soundfiles in your corpus is to use its path as the first argument of the 'csf' object:

```
CORPUS = [csf("lachenmann.aiff"), csf('piano')] # will use segments from lachenmann.
 aiff as well as all sounds in the directory called piano
```

However, as you can see above, each 'csf' object has *a lot* of optional arguments to give you better control over what segments are used, how directories are read and how segments are treated during concatenation.

Note: Each of these keyword arguments only apply to the csf object within which they are written. If you'd like to specify these parameters for the entire corpus, see section 4.2.5: Specifying CORPUS entry attributes globally.

4.2.1 Manipulating which segments are added to the corpus

start Any segments which start before this time will be ignored.

end Any segments which start after this time will be ignored.

```
csf('lachenmann.aiff', start=20) # only use segments who start later than 20s.
csf('lachenmann.aiff', start=20, end=50) # only use segments who start between 20-50s.
```

includeTimes A list of two-number lists which specify regions of segments to include from this file's list of segment times. See example below.

excludeTimes Same as includeTimes but excludes segments in the identified regions.

```
csf('lachenmann.aiff', includeTimes=[(1, 4), (10, 12)]) # only use segments falling
 between 1-4 seconds and 10-12 seconds.
csf('lachenmann.aiff', excludeTimes=[(30, 55)]) # use all segments except those
 falling between 30-55s.
```

limit A list of equation-like strings where segmented descriptor names are used to include/exclude segments from this file / directory.

```
csf('lachenmann.aiff', limit['centroid-seg >= 1000']) # segments whose centroid-seg is
 equal to or above 1000.
```

```
csf('lachenmann.aiff', limit['centroid-seg < 50%']) # only use 50% of segments with
 the lowest centroid-seg.

csf('lachenmann.aiff', limit['power-seg < 50%', 'power-seg > 10%']) # only use
 segments whose power-seg falls between 10%-50% of the total range of power-seg's in
 this file/directory.
```

4.2.2 Manipulating How Directories Are Read

The following keyword arguments are useful when dealing with directories of files.

wholeFile (False): if True audioguide will use this soundfile as one single segment. If False, audioguide will search for a segmnetation file made with segmentSf.py.

recursive (True): if True audioguide will include sounds in all subfolders of a given directory.

```
csf('sliced/my-directory', wholeFile=True) # will not search for a segmentation txt
 file, but use whole soundfiles as single segments.

csf('/Users/ben/gravillons', recursive=False) # will only use soundfiles in the named
 folder, ignoring its subdirectories.
```

includeStr (None): A string which is matched against the filename (not full path) of each soundfile in a given directory. If part of the soundfile name matches this string, it is included. If not it is excluded. This is case sensitive. See example below.

excludeStr (None): Opposite of includeStr.

```
# includeStr/excludeStr have lots of uses. One to highlight here: working with sample
  databases which are normalized. Rather than having each corpus segment be at 0dbs,
 we apply a scaleDb value based on the presence of a `dynamic' written into the
 filename.
csf('Vienna-harpNotes/', includeStr=['_f_', '_ff_'], scaleDb=-6),
csf('Vienna-harpNotes/', includeStr='_mf_', scaleDb=-18),
csf('Vienna-harpNotes/', includeStr='_p_', scaleDb=-30),
# this will use all sounds from this folder which match one of the three dynamics.
```

4.2.3 Manipulating How Segments Will Be Concatenated

scaleDb applies an amplitude change to each segment of this collection. by default, it is 0, yielding no change. -6 = twice as soft. Note that amplitude scaling affects both the concatenative algorithm and the csound rendering.

onsetLen if onsetLen is a float or integer, it is the fade-in time in seconds. If it is a string formed as '10%', it is interpreted as a percent of each segment's duration. So, onsetLen=0.1 yields a 100 ms. attack envelope while onsetLen='50%' yields a fade in over 50% of the segment's duration.

offsetLen Same as onsetLen, but for the envelope fade out.

```
csf('lachenmann.aiff', onsetLen=0.1, offsetLen='50%') # will use a segmentation file
 called marmotTent.txt, not the default lachenmann.aiff.txt.
```

transMethod A string indicating how to transpose segments chosen from this corpus entry.

transQuantize Quantization interval for transposition of corpus sounds. 1 will quantize to semitones, 0.5 to quarter tones, 2 to whole tones, etc.

```
csf('piano/', transMethod='f0') # transpose corpus segments to match the target's f0.
csf('piano/', transMethod='f0-chroma', transQuantize=0.5) # transpose corpus segments
 to match the target's f0 mod 12. Then quantize each resulting pitch to the newest
 quarter of tone.
```

allowRepetition If False, any of the segments from this corpus entry may only be picked one time. If True there is no restriction.

restrictRepetition A delay time in seconds where, once chosen, a segment from this corpus entry is invalid to be picked again. The default is 0.1, which the same corpus segment from being selected in quick succession.

```
csf('piano/', allowRepetition=False) # each individual segment found in this directory
  of files may only be deleted one time during concatenation.
csf('piano/', restrictRepetition=2.5) # Each segment is invalid to be picked if it has
  already been selected in the last 2.5 seconds.
```

restrictOverlaps An integer specifying how many overlapping samples from this collection may be chosen by the concatenative algorithm at any given moment. So, restrictOverlaps=2 only permits 2 overlapping voices at a time.

restrictInTime a time in seconds specifying how often a sample from this entry may be selected. – for example restrictInTime=0.5 would permit segments from this collection to be select a maximum of once every 0.5 seconds.

superimposeRule This one is a little crazy. Basically, you can specify when this corpus's segments can be chosen based on the number of simultanously selected samples. You do this by writing a little equation as a 2-item list. superimposeRule=('==', 0) says that this set of corpus segments may only be chosen is this is the first selection for this target segment (sim selection '0'). superimposeRule=('¿', 2) say this corpus's segments are only valid to by picked if there are already more than 2 selections for this target segment. I know, right?

4.2.4 Miscellaneous

segmentationFile Manually specify the segmentation text file. By default, AudioGuide automatically looks for a file with the same name as the soundfile plus the extension '.txt'. You may specify a string, or a list of strings to include multiply segmentation files which all use the same soundfile.

segmentationExtension Manually specify the segmentation text file extension.

```
csf('lachenmann.aiff', segmentationFile='marmotTent.txt') # will use a segmentation
 file called marmotTent.txt, not the default lachenmann.aiff.txt.

csf('lachenmann.aiff', segmentationExtension='-gran.txt') # will use a segmentation
 file called lachenmann.aiff-gran.txt, not the default lachenmann.aiff.txt.
```

4.2.5 Specifying CORPUS entry attributes globally

Corpus entry attributes may be specified globally using the variable CORPUS_GLOBAL_ATTRIBUTES. Note that they are specified in dictionary format rather than keyword format.

```
CORPUS = [csf('lachenmann.aiff', scaleDb=-6), csf('piano/', scaleDb=-6, wholeFile=True
 )]

# is equivalent to

CORPUS = [csf('lachenmann.aiff'), csf('piano/', wholeFile=True)]
CORPUS_GLOBAL_ATTRIBUTES = {'scaleDb': -6}
```

4.3 The SEARCH Variable

the SEARCH variable specifies how Audioguide pick corpus segments to match target segments. The idea here is make a *very* flexible searching structure where the user can create multiple search passes on different criteria.

4.4 SEARCH and spass objects

The SEARCH variable is written as a list of 'spass' objects.

```
spass(result_type, descriptor1... descriptorN, percent=None, minratio=None, maxratio=
 None)
```

Here is the most simple case:

```
SEARCH = [spass('closest', d('centroid'))] # will search all corpus segments and
 select the one with the `closest' centroid to the target segment.
```

Note that the first argument is the type of search performed – in this case, selecting the closest sample. Following the arguments are a list of descriptor objects which specify which descriptors to use. Finally there are some keyword parameters that we will touch on later.

```
SEARCH = [spass('closest', d('centroid'), d('effDur-seg'))] # will search all corpus
 segments and select the one with the `closest' centroid and effective duration
 compared to the target segment.
```

Ok, great. As you can probably imagine, the first argument, 'closest', tells AudioGuide to pick the closest sound. But, there are also other possibilities:

```
SEARCH = [spass('farthest', d('centroid'))] # return the worst matching segment.

SEARCH = [spass('closest_percent', d('centroid'), percent=20)] # return the top 20
 percent best matches.

SEARCH = [spass('farthest_percent', d('centroid'), percent=20)] # return the worst 20
 percent of matches.
```

If you use 'closest_percent' or 'farthest_percent' as the one and only spass object in the SEARCH variable, AudioGuide will select a corpus segment randomly among the final candidates. However, you can also chain spass objects together, essentially constructing a hierarchical search algorithm. So, for example, take the following SEARCH variable with two separate phases:

```
SEARCH = [
spass('closest_percent', d('effDur-seg'), percent=20), # take the best 20% of matches
 from the corpus
spass('closest', d('mfccs')), # now find the best matching segment from the 20 percent
  that remains.
]
```

I use the above example a lot when using AudioGuide. It first matches effDur-seg, the effective duration of the target measured againts't the effective duration of each corpus segment. It retains the 20% closest matches, and throws away the worst 80%. Then, with the remaining 20%, the timbre of the sounds are matched according to mfccs.

```
SEARCH = [spass('ratio_limit', d('centroid-seg'), minratio=0.9, maxratio=1.1)] #
 reduce the number of samples in the corpus such
```

Remember, the order of the spass objects in the SEARCH variable is very important – it is essentially the order of operations.

4.4.1 The D object

Use the ‘d’ object for specifying a descriptor in the SEARCH variable.

```
d('descriptor name', weight=1, norm=2, normmethod='stddev', distance='euclidean',
 energy=False)
```

weight How to weight this descriptor in relation to other descriptors.

```
SEARCH= [spass('closest', d('centroid', weight=1), d('noisiness', weight=0.5))]
 # centroid is twice as important as noisiness.
```

norm A value of 2 normalizes the target and corpus data separately. A value of 1 normalizes the target and corpus data together. 2 will yield a better rendering of the target’s temporal contour. 1 will remain more faithful to concrete descriptor values. I recommend using 2 by default, only using 1 when dealing with very ‘descriptive’ descriptors like duration or pitch.

```
SEARCH= [spass('closest', d('centroid'), d('effDur-seg', norm=1))]
```

normmethod How to normalize data – either ’stddev’ or ’minmax’. minmax is more precise, stddev is more forgiving of ‘outliers.’

distance Only valid for time-varying descriptors. How to arithmetically evaluate distance between continuously valued array. ’euclidean’ does a simple least squares search. Other methods include ’pearson’, ’buttuck’ and ’logjammin’.

```
SEARCH= [spass('closest', d('centroid', distance='pearson'))] # uses a pearson
 correlation formula for determining distance between the continuously valued
 centroid of target and corpus segments.
```

energy Only valid for time-varying descriptors. Weight distance calculations with the corpus segments energy values. So, softer frames will not penalize distance.

5 Examples

5.1 Example 1

```
TARGET = tsf('cage.aiff', thresh=-28)

CORPUS = [
csf('lachenmann.aiff'),
]
```

```
SEARCH = [
spass('closest_percent', d('effDur-seg', norm=1), percent=20),
spass('closest', d('mfccs'))
]

TARGET_ONSET_DESCRIPTORS = {'power': 1}

#  This example is called singleSelection for a reason -- here we set the superimpose
 object to only allow one corpus segment to be selected for each target segment (
 maxSegment=1).  Since the first spass in SEARCH is using the descriptor effDur-seg,
 we can except to have somewhat similar durations for the selected corpus segments.
 However, note that this might not be true, in particular if you use a corpus will
 wildly different segment durations that your target.  If you don't care about
 duration, you can remove the first spass object from SEARCH.  If you want durations
 to be rendered to match the target more precisely, see below.
SUPERIMPOSE = si(maxSegment=1)


# If you uncomment one of the following lines, csound will stretch selected corpus
 samples to match the duration of the target segments (after they're picked).  This
 will not affect which segments are selected, only their playback duration in csound
 rendering.  By default, CSOUND_STRETCH_CORPUS_DURATIONS_TO_MATCH_TARGET=0, which does
  not perform and temporal manipulation.
#CSOUND_STRETCH_CORPUS_DURATIONS_TO_MATCH_TARGET = 1 # 1 = phase vocoder (will not
 change pitch)
#CSOUND_STRETCH_CORPUS_DURATIONS_TO_MATCH_TARGET = 2 # 2 = tape-head transposition (
 will change pitch, but not "phasy")
```

6 The SUPERIMPOSE variable

Use the 'si' object for specifying how corpus segments may be superimposed during concatenation.

```
SUPERIMPOSE = si(minSegment=None, maxSegment=None, minOnset=None, maxOnset=8,
 minOverlap=None, maxOverlap=None, overlapAmpThresh=-70., searchOrder='power',
 calcMethod='mixture', peakAlign=False)
```

minSegment The minimum number of corpus segments that must be chosen to match a target segment.

maxSegment The maximum number of corpus segments that must be chosen to match a target segment.

minOnset The minimum number of corpus segments that must be chosen to begin at any single moment in time.

maxOnset The maximum number of corpus segments that must be chosen to begin at any single moment in time.

minOverlap The minimum number of overlapping corpus segments at any single moment in time. Note that an 'overlap' is determined according to an amplitude threshold – see overlapAmpThresh.

maxOverlap The maximum number of overlapping corpus segments at any single moment in time. Note that an 'overlap' is determined according to an amplitude threshold – see overlapAmpThresh.

overlapAmpThresh

searchOrder ('power' or 'time') The default is 'time', which indicated to match corpus segments to target segments in the temporal order of the target (i.e., first searched segment is the first segment in time). 'power' indicates to first sort the target segments from loudest to softest, then search for corpus matches.

calcMethod A string which denotes how to calculate overlapping corpus sounds. None does nothing – each corpus selection is unaware of previous selections. 'subtract' subtracts the energy of a selected corpus sound from the target's amplitude so that future selections might be later in time and softer. 'mixture' subtracts the amplitude and then attempts to mix the descriptors of simultaneous sounds together. Note that some descriptors are not algorithmically mixable, such as f0, zeroCross, and peak descriptors.

7 Other Options

7.1 Printing/Interaction

ALERT_ON_ERROR (type=bool, default=False) will attempt to play the system alert sound when exiting with an error. Pretty irritating, but then again you probably deserve it.

VERBOSITY (type=int, default=2) affects the amount of information audioguide prints to the terminal. A value of 0 yields nothing. A value of 1 prints a minimal amount of information. A value of 2 (the default) prints refreshing progress bars to indicate the progress of the algorithms.

PRINT_SELECTION_HISTO (type=bool, default=False) if True will print robust information about corpus selection after concatenation. If false (the default) will add this information to the log file, if used.

PRINT_SIM_SELECTION_HISTO (type=bool, default=False) if True will print robust information about corpus overlapping selection after concatenation. If false (the default) will add this information to the log file, if used.

7.2 Concatenation

RANDOM_SEED (type=int/None, default=None) sets the pseudo-random seed for random unit selection. By default a value of None will use the system's timestamp. Setting an integer will create repeatable random results.

OUTPUT_GAIN_DB (type=int/None, default=None) adds a uniform gain in dB to all selected corpus units. Affects the subtractive envelope calculations and descriptor mixtures as well as csound rendering.

OUTPUT_TIME_STRETCH (type=float, default=1.) stretch the temporality of selected units. A value of 2 will stretch all events offsets by a factor of 2.

OUTPUT_TIME_ADD (type=float, default=0.) offset the start time of selected events by a value in seconds.