CLEESE v0.1

A Python toolbox for random and deterministic sound transformations

October 2017

Contents

1 Introduction

CLEESE (Combinatorial Expressive Speech Engine) is a Python toolbox for performing random or deterministic pitch, timescale, filtering and gain transformations on an input sound. It is originally aimed at generating many random variations of a single speech utterance, to be used as stimuli in the scope of listening tests for reverse correlation experiments. The modifications can be both static or time-varying. Besides its original purpose, the toolbox can also be used for producing individual, user-determined modifications.

CLEESE operates by generating a set of random breakpoint functions (BPFs) in the appropriate format for each treatment, which are then passed to the included spectral processing engine (based on a Phase Vocoder) with the corresponding parameters. Alternatively, the BPFs can be externally created by the user, and so it can also be used as a Phase Vocoder-based effects unit.

CLEESE was developed by Juan José Burred in collaboration with the CREAM (Cracking the Emotional Code of Music) project headed by Jean-Julien Aucouturier at IRCAM/CNRS. The toolbox has been tested on both Python 2 and 3. It requires Numpy and Scipy.

2 Operation modes

CLEESE can be used in several different modes, depending on how the main processing function is called. Examples of several typical usage scenarios are included in the example script `run_cleese.py`.

2.1 Batch generation

In batch mode, CLEESE generates many random modifications from a single input sound file, called the base sound. It can be launched as follows:

```
import cleese

inputFile  = 'path_to_input_sound.wav'
configFile = 'path_to_config_file.py'

cleese.process(soundData=inputFile, configFile=configFile)
```

Two parameters have to be set by the user:

- `inputFile`: the path to the base sound, which has to be a mono sound in WAV format.
- `configFile`: the path to the configuration script

All the generation parameters for all treatments are set up in the configuration script that has to be edited or created by the user. An example of configuration script with parameters for all treatments is included with the toolbox: `cleeseConfig_all.py`. Configuration parameters will be detailed in Sect. 3.

For each run in batch mode, the toolbox generates the following folder structure, where `<outPath>` is specified in the parameter file:

- `<outPath>/<currentExperimentFolder>`: main folder for the current generation experiment. The name `<currentExperimentFolder>` is automatically created from the current date and time. This folder contains:
  - `<baseSound>.wav`: a copy of the base sound used for the current experiment
  - `*.py`: a copy of the configuration script used for the current experiment
  - One subfolder for each one of the performed treatments, which can be either `pitch`, `eq`, `stretch` or `gain`, or a combination (chaining) of them. Each of them contains, for each generated stimulus:
    - `<baseSound>.xxxx.<treatment>.wav`: the generated stimulus, where `xxxx` is a running number (e.g.: `cage.0001.stretch.wav`)
    - `<baseSound>.xxxx.<treatment>BPF.txt`: the generated BPF, in ASCII format, for the generated stimulus (e.g.: `cage.0001.stretchBPF.txt`)

2.2 Passing a given BPF

When passing the `BPF` argument to `cleese.process`, it is possible to impose a given BPF with a certain treatment to an input file. In this way, the toolbox can be used as a traditional effects unit.

```
import cleese

inputFile  = 'path_to_input_sound.wav'
configFile = 'path_to_config_file.py'

import numpy as np
givenBPF = np.array([[0.,0.],[3.,500.]])
cleese.process(soundData=inputFile, configFile=configFile, BPF=givenBPF)
```

The `BPF` argument can be either:

- A Numpy array containing the BPF, in the format specified in Sect. 4
- A scalar, in which case the treatment performed is static

In this usage scenario, only one file is output, stored at the `<outPath>` folder, as specified in the configuration file.

2.3 Array input and output

Instead of providing a file name for the input sound, it is possible to pass a Numpy array containing the input waveform. In this case, the main function will provide as output both the modified sound and the generated BPF as Numpy

arrays. No files or folder structures are created as output:

```
import cleese

inputFile  = 'path_to_input_sound.wav'
configFile = 'path_to_config_file.py'

waveIn,sr = cleese.wavRead(inputFile)
waveOut,BPFout = cleese.process(soundData=waveIn, configFile=configFile, sr=sr)
```

Note that the sampling rate `sr` has to be passed as well! Like when passing a BPF, only a single sound is generated.

3 Configuration script

All the generation parameters are set in the configuration script. Please refer to the included configuration script `cleeseConfig_all.py` for an example.

3.1 Main parameters

The main parameters are set as follows:

```
# main parameters
main_pars = {
    'outPath': '/path_to_output_folder/',      # output root folder
    'numFiles': 10,      # number of output files to generate (for random modifications)
    'chain': True,       # apply transformation in series (True) or parallel (False)
    'transf': ['stretch','pitch','eq','gain']  # modifications to apply
}

# global analysis parameters
ana_pars = {
    'anaWinLen':   0.04,     # analysis window length in sec
    'oversampling':   8,     # number of hops per analysis window
}
```

Chaining of transformations

If `'chain'` is set to `True`, the transformations specified in the `'transf'` list will be applied as a chain, in the order implied by the list. For instance, the list `['stretch','pitch','eq','gain']` will produce the output folders `stretch`, `stretch_pitch`, `stretch_pitch_eq` and `stretch_pitch_eq_gain`, each one containing an additional step in the process chain.

If `'chain'` is set to `False`, the transformations will be in parallel (all starting from the original sound file), producing the output folders `stretch`, `pitch`, `eq` and `gain`.

3.2 Common parameters

The following parameters are shared by all treatments, but can take different values for each of them. Treatment-specific parameters will be covered in Sect. 5.

In the following, <`treatment`> has to be replaced by one of the strings in `['stretch','pitch','eq','gain']`:

```
# common treatment parameters
<treatment>_pars = {
    'winLen': 0.11,     # BPF window in seconds. If 0 : static transformation
    'numWin': 6,        # number of BPF windows. If 0 : static transformation
    'winUnit': 'n',     # 's': force winLen in seconds,
                        # 'n': force number of windows (of equal length)
    'std': 300,         # standard deviation for each BPF point of the random modification
    'trunc': 1,         # truncate distribution values (factor of std)
    'BPFtype': 'ramp', # type of breakpoint function:
                        #      'ramp': linear interpolation between breakpoints
                        #      'square': square BPF, with specified transition times at edges
    'trTime': 0.02      # in sec: transition time for square BPF
}
```

- `winLen`: Length in seconds of the treatment window (i.e., the window used to generate the timestamps in the BPFs - see Sect. 4). It should be longer than `anaWinLen`. This is only used if `'winUnit': 's'` (see below).
  - Static treatment: if `'winLen': 0`, the treatment is static (flat BPF).
- `numWin`: Total number of treatment windows. This is only used if `'winUnit': 'n'` (see below).
  - Static treatment: if `'numWin': 0`, the treatment is static (flat BPF).
- `winUnit`: Whether to enforce window length in seconds (`'s'`) or integer number of windows (`'n'`).
- `std`: Standard deviation of a Gaussian distribution from which the random values at each timestamp of the BPFs will be sampled. The unit of the std is specific to each treatment:
  - For `pitch`: cents
  - For `eq` and `gain`: amplitude dBs
  - For `stretch`: stretching factor (>1: expansion, <1: compression)
- `trunc`: Factor of the std above which distribution samples are not allowed. If a sample is higher than `std*trunc`, a new random value is sampled at that point.
- `BPFtype`: Type of BPF. Can be either `ramp` or `square` (see Sect. 4).
- `trTime`: For BPFs of type `square`, length in seconds of the transition phases.

4 BPFs

In CLEESE, sound transformations can be time-varying: the amount of modification (e.g. the pitch shifting or time stretching factors) can dynamically change over the duration of the input sound file. The breakpoint functions (BPFs) determine how these modifications vary over time. For the `pitch`, `stretch` and `gain` treatments, BPFs are one-dimensional (temporal). For the `eq` treatment, BPFs are two-dimensional (spectro-temporal).

As has been seen, BPFs can be either randomly generated by CLEESE or provided by the user.

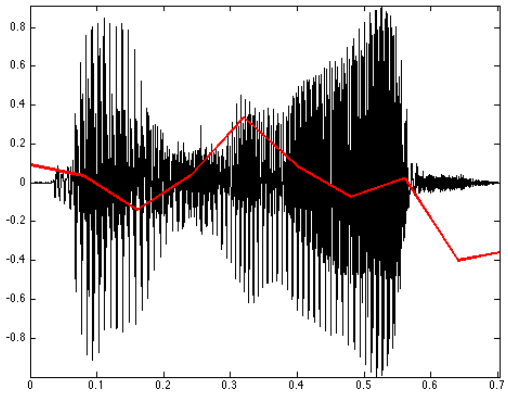
4.1 Temporal BPFs

For the `pitch`, `stretch` and `gain` treatments, BPFs are temporal: they are two-column matrices with rows of the form:

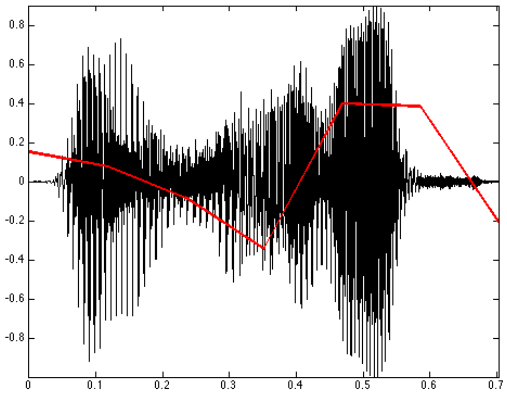
```
time value
```

`time` is in seconds, and `value` is in the same units than the `std` parameter. CLEESE can randomly generate one-dimensional BPFs of two types:

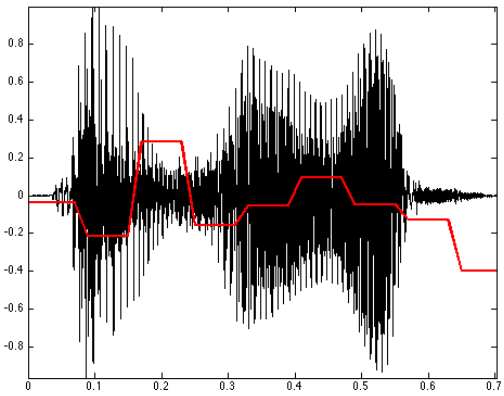
- **Ramps** (`'BPFtype': 'ramp'`): the BPF is interpreted as a linearly interpolated function. The result is that the corresponding sound parameter is changed gradually (linearly) between timestamps. Examples are shown on Fig. 1 for a treatment window defined in terms of seconds (`'winUnit': 's'`), and on Fig. 2 for a treatment window defined in terms of window number (`'winUnit': 'n'`). Note that in the first case, the length of the last window depends on the length of the input sound. In the second case, all windows have the same length.



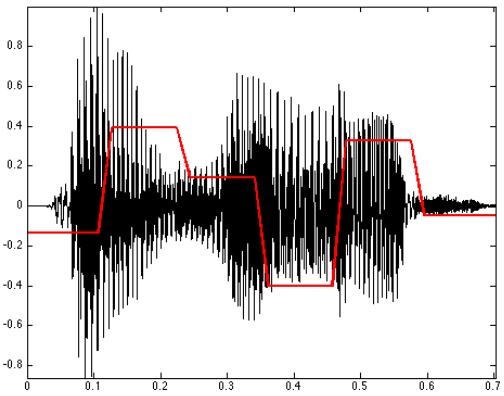
(a) Ramp BPF with window specified in seconds



(b) Ramp BPF with window specified in number



(c) Square BPF with window specified in seconds



(d) Square BPF with window specified in number

Figure 1: Examples of randomly generated temporal BPFs.

- **Square** (`'BPFtype': 'square'`): the BPF is a square wave with sloped transitions, whose length is controlled by `trTime`. Examples are shown on Fig. 3 for a treatment window defined in terms of seconds (`'winUnit': 's'`), and on Fig. 4 for a treatment window defined in terms of window number (`'winUnit': 'n'`).

4.2 Spectro-temporal BPFs

The `eq` treatment performs time-varying filtering over a number of determined frequency bands. It thus expects a spectro-temporal (two-dimensional) BPF whose rows are defined as follows:

```
time numberOfBands freq1 value1 freq2 value2 freq3 value3 ...
```

The temporal basis can again be generated as `ramp` or `square`. In contrast, in the frequency axis, points are always interpolated linearly. Thus, a spectro-temporal BPF can be interpreted as a time-varying piecewise-linear spectral envelope.

5 Treatments

5.1 Time stretching (`stretch`)

This treatment stretches or compresses locally the sound file according to the current stretching factor (oscillating around 1) at the current timestamp. This is the only treatment that changes the duration of the output compared to

the base sound. The used algorithm is a phase vocoder with phase locking based on frame-wise peak picking.

5.2 Pitch shifting (pitch)

The BPF is used to transpose up and down the pitch of the sound. The used algorithm is a phase vocoder with phase locking based on frame-wise peak picking, followed by resampling on a window-by-window basis.

5.3 Time-varying equalization (eq)

This treatment divides the spectrum into a set of frequency bands, and applies random amplitudes to the bands. The definition of band edges is constant, the amplitudes can be time-varying. The corresponding BPF is thus two-dimensional and follows the format described in Sect. 4.2.

There are two possible ways to define the band division:

- **Linear** division into a given number of bands between 0 Hz and Nyquist.
- Division according to a **mel** scale into a given number of bands. Note that it it possible to specify any number of filters (less or more than the traditional 40 filters for mel cepstra).

These settings are defined by the following treatment-specific parameters:

```
eq_pars = {
    'scale': 'mel',    # mel, linear
    'numBands': 10
}
```

5.4 Time-varying gain (gain)

For gain or level randomization, the BPF is interpolated and interpreted as an amplitude modulator. Note that the corresponding standard deviation is specified in base-10 logarithm. If the resulting output exceeds the maximum float amplitude of `1.0`, the whole output signal is normalized.