pySoundLab Documentation

Release 6

Mirco Pezzoli, Federico Sala

January 25, 2017

Contents

CHAPTER 1

Introduction

pySoundLab is a Python package for acoustics studio audio measuraments. You can use pySoundLab to measure the impulse response of a room, its wideband energy and visualize the spectrum divided by portion of octave. It contains also a small application with graphic interface to perform these tasks without writing a single line of code! Simply type in your console **`pysoundlab`** to get the application running.

This package was developed by Mirco Pezzoli and Federico Sala as a project of "Sound Analysis, Synthesis and Processing" and "Computer Music" courses at Politecnico di Milano.

Being a Python module you can easily write your own script and add functionalities to **pySoundLab**. If you want to contribute at the project fork **pySoundLab** repository on Bitbucket.

CHAPTER 2

Requirements

Python:

The **pySoundLab** module is developed for Python 2. We advice version `2.7.12` or greater. By the way it should work with any Python version which support `CFFI` (see below). If you don't have Python you can install one of the distribution which already include `CFFI` and Numpy (and many other useful things) e.g. Anaconda or WinPython.

Note: No Python 3 support is provided.

CFFI:

The C Foreign Function Interface for Python is used to access the C-API of the PortAudio library from within Python. It supports CPython 2.6, 2.7, 3.x; and is distributed with PyPy. If it's not installed already, you should install it with your package manager (the package might be called python-cffi or similar), or you can get it with:

```
python -m pip install cffi --user
```

Libraries:

- numpy and scipy
- matplotlib
- sounddevice
- portaudio

sounddevice:

The crossplatform audio I/O is performed by `sounddevice` which is an interface for portaudio library. You must get portaudio before install `sounddevice`.

numpy and scipy:

Operations on data and I/O on disk are implemented with these scientific modules

matplotlib:

It is used to plot signals in time and frequency domain

All Python modules can be installed with `pip`.

portaudio:

The PortAudio library must be installed on your system (and CFFI must be able to find it). You should use your package manager to install it (the package might be called `libportaudio2` or similar). If you prefer, you can of course also download the sources and compile the library yourself. If you are using Mac OS X or Windows, the library will be installed automagically with pip (during `sounddevice` `pip` installation).

CHAPTER 3

Installation

Since it is not yet hosted on **PyPy** you have to manually download the wheel from our repository (or wherever it will be hosting). Once you've downloaded the wheel, you can use `pip` to install **pySoundLab** with one single command.

```
pip install path/to/your/wheel/pySoundLab-x.x.x-py2-none-any.whl
```

`pip` should install all the above mentioned dependencies automatically for you. If you experience errors in installing directly **pySoundLab**, we advice you to "manually" install the dependency first.

If you want to install it only for the current user you can add the flag `--user`.

To un-install use:

```
pip uninstall pysoundlab
```

CHAPTER 4

Usage

4.1 Application

Launch the application we made easily by typing this command in your bash:

pysoundlab

This will set up the graphical interface letting you to perform multichannel impulse response and to save your data.

With pysoundlab script application you will able to:

1. Choose the input and output audio devices
2. Choose an arbitrary number of input and output channels from the channels matrix
3. Specify the impulse response length
4. Specify the portion of octave analysis
5. Visualize the impulse response and its wideband and per-portion-of-octave energy.
6. Save the results.

Data is saved as following:

- Impulse response: .wav (wave file)
- Wideband energy: .csv (comma separated values)
- Per-band energy: .mat (matlab format)

4.2 Python Module

First import the module:

```
>>> import pysoundlab
```

Then you most likely load a test signal:

```
>>> rate, test = pysoundlab.wavutil.load_wave('path/your_test_signal.wav')
```

You can use the ones already in **pySoundalb**, they are in the package folder.

You can have knowledge about your audio devices with:

```
>>> pysoundlab.utils.devices()
... {'input': {u'Scarlett 2i2 USB - Core Audio': {'channels': 2, 'index': 2},
... u'Built-in Input - Core Audio': {'channels': 2, 'index': 0}},
... 'output': {u'Scarlett 2i2 USB - Core Audio': {'channels': 2, 'index': 2},
... u'Built-in Output - Core Audio': {'channels': 2, 'index': 1}}}
```

This are only few possibilities of **pySoundLab**, all the features are explained deeper in the *Documentation*.

CHAPTER 5

Documentation

In this paragraph you can find the documentation of **pySoundLab** submodules, functions and classes. Module's functions are divided in submodules accordingly to their main role. There are four submodule:

1. *measures*
2. utils
3. *wavutil*
4. *files*

Acoustics measurements are all implemented in *measures* while other modules are utilities for performing the measurements.

5.1 Measures

This module is the core of pySoundLab. It contains the main functions to compute the requested acoustics measurements:

- Impulse response
- Wideband energy
- Energy in portion of octaves (RTA)

You can use *impulse_response()* to compute the impulse response of a previously recorded signal. This signal must be the room response to our ad-hoc test signal located in input/. if you are interested in the procedure to obtain the result visit **pySoundLab** references.

The function *energy()* provides to you the energy of whatever kind of signal you give in its input.

The *filterBank* class is a fully customizable in portion of octave size central frequency implementation of a filberbank.

An utility to show the spectrum of a signal is brougth to you with function *plot_spectrum()*

Examples

Import the module to be able to use the functions:

```
>>> import pysoundlab
```

Computed the impulse response:

```
>>> ir = pysoundlab.impulse_response(recorded_data, test_signal)
```

measures.**inverse_filter**(*data*, *rate=96000*, *fi=10*, *ff=22000*)

This function computes an inverse filter.

It flips the input array and multiplies it by an exponential to give the signal an exponential decay. The expression of the exponential function depends on the signal duration, the start and the stop frequencies of the sine sweep signal. The inverse filter is used to extract the impulse response. If you want to know more of this impulse response computation method visit the **pySoundLab** references. In practice you will not use this method directly because it is called in *impulse_response()*.

Parameters

- **data** (*array-like*) – The signal
- **rate** (*int, optional*) – Signal sample rate
- **fi** (*int or float, optional*) – Initial frequency of the sweep (default 10Hz)
- **ff** (*int or float, optional*) – Final frequency of the sweep (default 22000Hz)

Returns the inverted signal

Return type np.array

measures.**impulse_response**(*data*, *test*, *length=None*, *rate=96000*, *fi=10*, *ff=22000*)

This function computes the impulse response.

`data` parameter is a recorded room response to our designed test signal given in input as `test` parameter. You can specify manually the length of the resulting impulse response with paramenter `length`. Impulse response is determined using a sinusoidal sweep as input signal and its inverse version computed with `inverse filter()`. For more informations read **pySoundLab** references.

Parameters

- **data** (*array-like*) – The recorded room response
- **test** (*array-like*) – The test signal used to record the response
- **length** (*int or float or None (optional)*) – Length in seconds of the impulse response (default None as original length)
- **rate** (*int or float (optional)*) – Sample rate (default 96KHz)
- **fi** (*int or float, optional*) – Initial frequency of the sweep (default 10Hz)
- **ff** (*int or float, optional*) – Final frequency of the sweep (default 22000Hz)

Returns The impulse response of the environment in which the software is run

Return type np.array

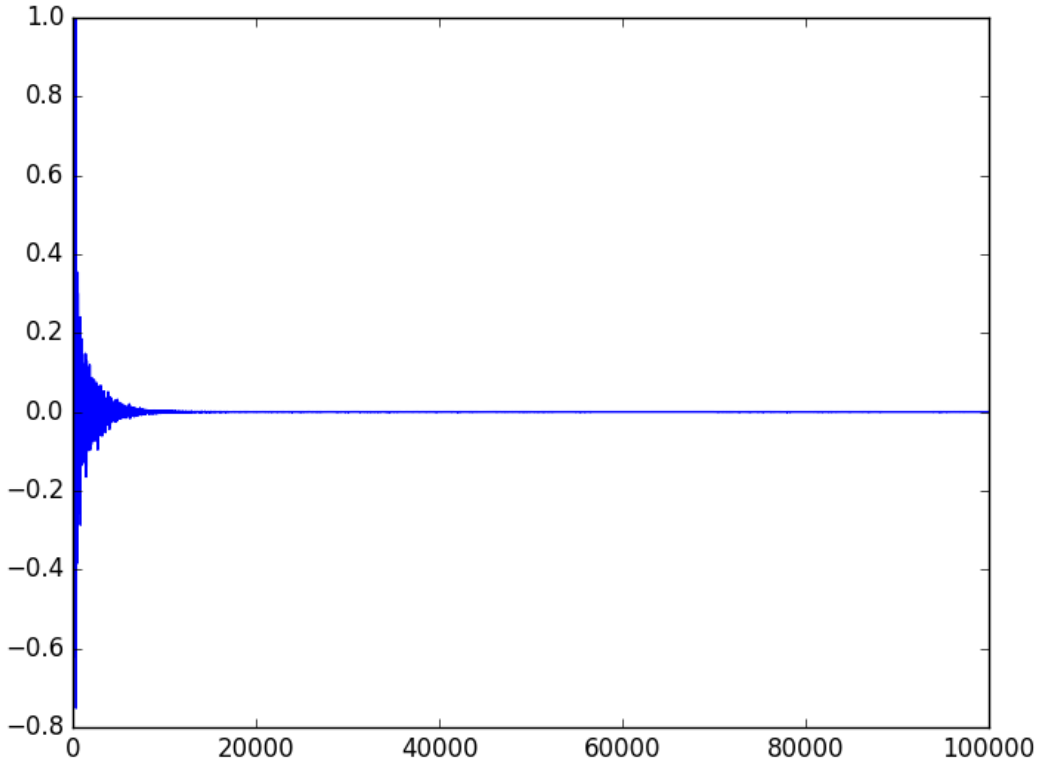
Examples

Computes the impulse response of a room

```
>>> import sounddevice as sd
>>> import pysoundlab
>>> wav = pysoundlab.wavutil
>>> rate, test = wav.load_wave('./input/test_signal.wav')
>>> recorded = sd.playrec(test, samplerate=rate, channels=1,
        dtype='float32')
>>> ir = pysoundlab.impulse_response(recorded, test, length=1.0)
```

Plot the impulse response

```
>>> import matplotlib.pyplot as plt
>>> plt.plot(ir)
... [<matplotlib.lines.Line2D object at 0x108ea1b10>]
>>> plt.show()
```



measures.**energy**(*data*, *rate*)

This function computes the wideband energy of a signal.

With this function you can compute the energy of each frequency content inside the signal spectrum.

Parameters

- **data** (*array-like*) – The signal
- **rate** (*int or float*) – Sampling rate

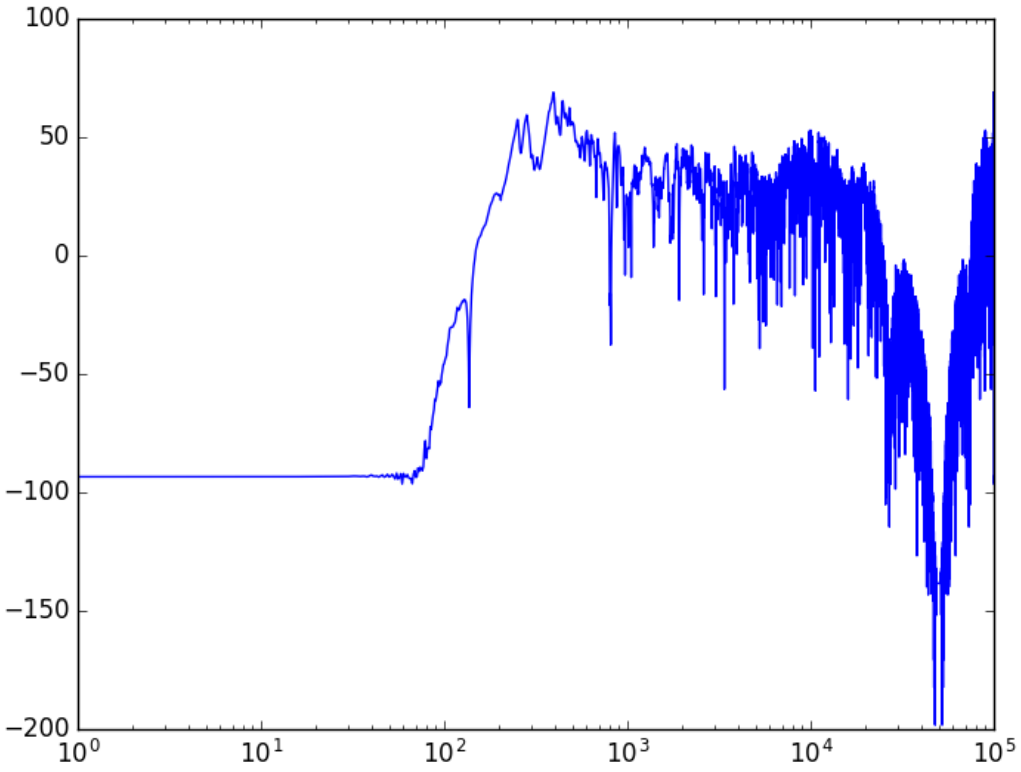
Returns Energy spectrum

Return type array-like

Examples

Computes energy and show results:

```
>>> w_energy = pysoundlab.energy(signal, rate)
>>> plt.plot(pysoundlab.decibel(w_energy))
... [<matplotlib.lines.Line2D object at 0x108ea1b10>]
>>> plt.show()
```



measures.**decibel**(*data*)

This function compute the signal in dB.

It is a simple utility function that transform signal values in its dB version. Really useful to plot the energy of a signal.

Parameters **data** (*array-like*) – The input signal

Returns The signal in dB

Return type array-like

Examples

```
>>> import matplotlib.pyplot as plt
>>> en = pysoundlab.energy(signal, rate)
>>> en_db = pysoundlab.decibel(en)
>>> plt.plot(en_db)
... [<matplotlib.lines.Line2D object at 0x108ea1b10>]
>>> plt.show()
```

measures.**plot_spectrum**(*data*, *rate*)

Utility function to plot the Spectrum.

It plots a Single-Sided Amplitude Spectrum of the signal `data`. It is useful to see the frequency contente of a signal. Values are automatically converted to dB.

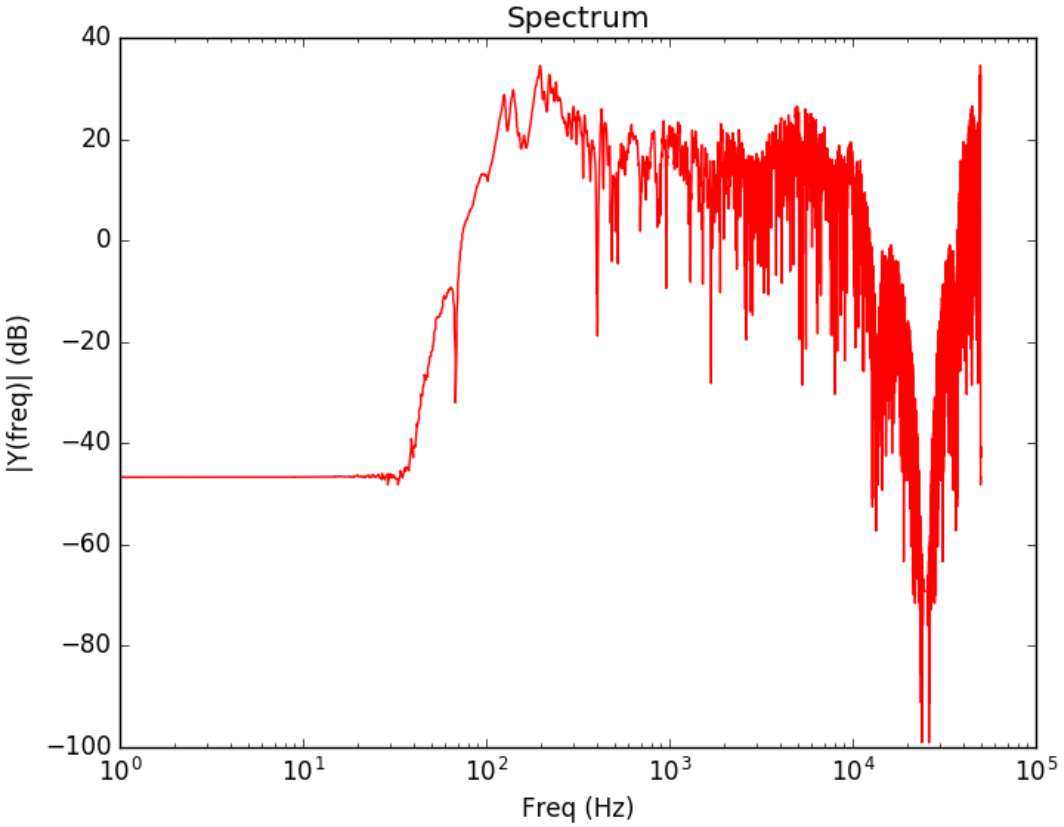
Parameters

- **data** (*array-like*) – The signal to visualize
- **rate** (*int*) – Sample rate

Examples

Plot the spectrum of a signal:

```
>>> pysoundlab.plot_spectrum(signal, rate)
```



class measures.**filterBank**(*rate*, *center=1000*, *portion=3*, *order=3*)

This is an implementation of a filterbank divided in portion of octave.

If you're going to use the same filterbank to filter multiple signals you have to initializate just one object of class `filterbank` and compute the filtered signals with the method *filter()*. You can choose the center frequency of octaves and the ratio of octave you want to visualize. With *plot()* you will see the energy in the portions of octave directly in a `matplotlib` bar graph.

freqs

dict

the filter frequencies

__init__(*rate*, *center=1000*, *portion=3*, *order=3*)

initializes and creates a filterbank

filter(*data*)
filters the signal

_bandpass_filter(*lowcut*, *highcut*, *rate*, *order=3*)
creates a bandpass

_butter_bandpass(*lowcut*, *highcut*, *rate*, *order=3*)
creates a Butterworth filter

bandpass_freq(*rate*, *center=1000*, *portion=3*)
computes filters frequencies

plot()
plots the results

Methods

__init__(*rate*, *center=1000*, *portion=3*, *order=3*)
This function initializes and creates the octave filter bank.

Parameters

- **rate** (*int*) – Sample rate
- **center** (*int, optional*) – The center frequency on which compute all the portions of octave. If 1000 (default), the center freuqencies follow the ISO standard.
- **portion** (*int, optional*) – The desired portion of octave. Default is 3, that it means to compute a filter bank at 1/3 of octave.
- **order** (*int, optional*) – The order of the Butterworth bandpass filter used to compute the variuos filters in the bank. Default value is 3.

__module__ = 'measures'

_filterBank__bandpass_filter(*lowcut*, *highcut*, *rate*, *order=3*)
Computes the filter coefficients according to scipy's sosfiltering.

Parameters

- **lowcut** (*int*) – Low cut frequency of the bandpass filter in Hz
- **highcut** (*int*) – High cut frequency of the bandpass filter in Hz
- **rate** (*int*) – Sample rate
- **order** (*int, optional*) – Order of the Butterworth bandpass filter. Default value is '3'.

Returns **sos** – Array of second-order filter coefficients.

Return type ndarray

See also:

scipy.signal.sosfilt scipy.signal.zpk2sos

_filterBank__butter_bandpass(*lowcut*, *highcut*, *rate*, *order=3*)
Creates a bandpass filter using a digital Butterworth filter.

It will return its zeros, poles and gain.

Parameters

- **lowcut** (*int*) – Low cut frequency of the bandpass filter in Hz

- **highcut** (*int*) – High cut frequency of the bandpass filter in Hz
- **rate** (*int*) – Sample rate
- **order** (*int, optional*) – Order of the Butterworth bandpass filter. Default value is 3.

Returns **z, p, k** – Zeros, poles, and system gain of the IIR filter transfer function.

Return type ndarray, ndarray, float

See also:

scipy.signal.butter

bandpass_freq(*rate*, *center=1000*, *portion=3*)

Computes the filter frequencies.

For each center frequency computed it will created the lower and the higher frequencies cut according to the specified portion. This information are stored as a dictionay in the attribute *freqs*. See the Returns section of the method for dictionary structure.

Parameters

- **rate** (*int*) – Sample rate
- **center** (*int, optional*) – The center frequency on which compute all the portions of octave. If 1000 (default), the center frequencies follow the ISO standard.
- **portion** (*int, optional*) – The desired portion of octave. Default is 3, that it means to compute a filter bank at 1/3 of octave. Otherwise, the center frequencies are computed following a different procedure.

Returns **freqs** – Center frequencies as key and (low_freq_cut, high_freq_cut) as values. The result is sorted on the key.

Return type dict

filter(*data*)

This function filters a signal.

The input is filetered and its energy divided per portion of octave is computed. The energy is computed according to the Parseval's theorem.

Parameters **data** (*array-like*) – The signal to filter

See also:

scipy.signal.sosfilt

plot()

Fucntion used to plot and show the

RTA in a bar way and on a log xscale. On the y-axis energy [dB] values per band are plotted.

5.2 Wavutil

This is a support module of **pySoundLab** package for save and open `.wav` files. The functions help to load/save wave files from/to disk and with samples contained in `numpy darray` used for calculation in *measures*.

Examples

```
>>> from pysoundlab import wavutil as wav
>>> import numpy as np
>>> data = np.random.rand(96000)
>>> wav.save_wave('/tmp/white_noise.wav', data)
```

wavutil.**save_wave**(*path*, *data*, *rate=96000*)

Function used to save a `numpy array` in a wave file on disk.

You have simply to provide the `path` and the `data` as a `numpy darray` to save the signal in a wavefile. The `numpy darray` can be 2D (see also section for more information).

Parameters

- **path** (*string*) – OS address of the output file
- **data** (*array-like*) – Array of data to be put in the file
- **rate** (*int, optional*) – Sample rate

Examples

Save a white noise of 1(s) at 96kHz

```
>>> wav.save_wave('/tmp/white_noise.wav', data)
```

See also:

scipy.io.wavfile.write

wavutil.**load_wave**(*path*)

Function used to load a wave file from disk.

You have simply to provide the `path` and the data will be loaded in a `numpy darray`. Whatever bitdepth is used samples will be cast to `numpy float32` The `numpy darray` can be 2D (see also section for more information).

Warning: Samples are cast to `numpy float32`

Parameters **path** (*string*) – OS address of the input file

Returns

- **rate** (*int*) – Sample rate
- **audio** (*array-like*) – Array containing the samples of the input file

Examples

```
>>> wav.load_wave('input/test_signal_44k.wav')
... (44100, array([ 0.        ,  0.00140385,  0.00283822, ..., -0.99960327,
...  0.99935913, -0.99908447], dtype=float32))
```

Typical use:

```
>>> rate, data = wav.load_wave('input/test_signal_44k.wav')
```

See also:

scipy.io.wavfile.read

5.3 Files

Utility functions to deal with files. The aim of this module is to bring to you some useful functions for directly loading files from directories.

The main function of the module is *find_files()* it search all files in a specified path.

Examples

```
>>> from pysoundlab import files as fi
>>> fi.find_files('input/')
... ['/your/path/to/input/test_signal_44k.wav',
... '/your/path/to/input/test_signal_48k.wav',
... '/your/path/to/input/test_signal_96k.wav']
```

files.**find_files**(*directory*, *ext=None*, *recurse=True*, *case_sensitive=False*, *limit=None*, *offset=0*)

Find files in a directory.

Specify the directory in which search files and the extension you want with other useful parameters. It will return all the paths of files with the specified extensions in a list. The number of files in the list is controlled with the parameters, `limit` and `offset`.

Parameters

- **directory** (*string*) – Directory of input files
- **ext** (*list, optional*) – Extensions to search (default None, that corresponds to .wav)
- **recurse** (*bool, optional*) – Recursive search (default True)
- **case_sensitive** (*bool, optional*) – Default False
- **limit** (*int, optional*) – Maximum number of files to load (default None, so all files)
- **offset** (*int, optional*) – Starting point of load (default 0)

Returns The list of found files' paths

Return type list

Examples

```
>>> fi.find_files('input/')
... ['/your/path/to/input/test_signal_44k.wav',
... '/your/path/to/input/test_signal_48k.wav',
... '/your/path/to/input/test_signal_96k.wav']
```

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

f

m

w

Symbols

B

D

E

F

I

L

M

P

S

W