# pambox Documentation

***Release***

**Author**

November 30, 2016

# Contents

**1 Installing** **3**

**2 Structure of the toolbox** **5**

**3 Conventions** **7**

**4 Contents** **9**

4.1 Audio . . . . . 9
4.2 Inner ear processing . . . . . 10
4.3 Middle ear processes . . . . . 11
4.4 Outer ear processes . . . . . 11
4.5 Central auditory processing . . . . . 11
4.6 Speech Intelligibility Models and Experiments . . . . . 16
4.7 Signal Distortion and Processing . . . . . 29
4.8 Utilities . . . . . 32

**5 Indices and tables** **41**

**Bibliography** **43**

**Python Module Index** **45**

pambox is a Python package to facilitate the development of auditory models, with a focus on speech intelligibility prediction models.

The Grand Idea is for *pambox* to be a repository of published auditory models, as well as a simple and powerful tool for developing auditory models. Components should be reusable and easy to modify. *pambox* uses a standard interface for all speech intelligibility prediction models in the package, which should simplify comparisons across models.

In case Python is not your thing and you prefer Matlab, the Auditory Modeling Toolbox is an excellent alternative.

CHAPTER 1

# Installing

Right now, *pambox* is only available through Github. It should be available via *pip* soon. To install *pambox* from source:

```
git clone https://github.com/achabotl/pambox.git
cd pambox
python setup.py install
```

You'll also need all the requirements in requirements.txt. If you use conda, you can simply run the following to install all the dependencies:

```
conda install --file requirements.txt
```

CHAPTER 2

# Structure of the toolbox

The structure of the toolbox is inspired by the auditory system. The classes and functions are split between "peripheral" and "central" parts. The "peripheral" part is directly accessible through an *`inner`*, a *`middle`*, and an *`outer`* module. The *`central`* part is more general and contains the modules and functions for central processes, without much extra separation for now.

The *`speech`* module contains speech intelligibility models and various functions and classes to facilitate speech intelligibility prediction experiments.

The *`utils`* module contains functions for manipulating signals, such as setting levels, or padding signals, that are not directly auditory processes.

The *`distort`* module contains distortions and processes that can be applied to signals. Most of them are used in speech intelligibility experiments.

The *`audio`* module is a thin wrapper around pyaudio that simplifies the playback of numpy arrays, which is often useful for debugging.

# Conventions

In the spirit of Python, pambox has a few conventions about "the way to do things".

- Single channels signals always have the shape `(N, )`, where *N* is the length of the signal.
- Multi-channels signals always have the shape `(M, N)`, where *M* is the number of channels and *N* is the signals' length. This greatly simplifies looping over channels.
- Filterbanks are classes with names ending in `Filterbank` and must take at least the sampling frequency and the center frequencies as arguments, for example: `GeneralFilterbank(fs, center_frequencies=(100, 200), *args, **kgwars)`. `center_frequencies` can have a default value. Filtering is done via a `filter` method that only takes the signal to filter and return multi-channel signals, for example: `GeneralFilterbank(fs=44100).filter(x)` returns a `(M, N)` array, where `M` can be 1.
- Speech intelligibility models are classes with a `predict` method. See *Speech Intelligibility Models* for more details.

CHAPTER 4

# Contents

## 4.1 Audio

The *audio* module provides a single function, *play()*. By default, the output is scaled to prevent clipping and the sampling frequency is 44.1 KHz. Here a simple example where we play some white noise created with NumPy:

```
from pambox import audio
import numpy as np
audio.play(np.random.randn(10000))
```

To play back the signal without normalization, simply set `normalize` to `False`. Be careful here! It might get loud!):

```
audio.play(np.random.randn(10000), normalize=False)
```

### 4.1.1 API

*audio* provides a simple wrapper around *pyaudio* to simplify sound playback.

`pambox.audio.`**`play`**(*x*, *fs=44100*, *normalize=True*)

Plays sound.

**Parameters** **x** : array_like,

Signal to be played. The shape should be nChannels x Length.

**fs** : int (optional)

Sampling frequency. The default is 44100 Hz.

**normalize** : bool (optional)

Normalize the signal such that the maximum (absolute value) is 1 to prevent clipping. The default is True.

#### Examples

To playback a numpy array:

```
>>> from pambox import audio
>>> import numpy as np
>>> audio.play(np.random.randn(10000))
```

## 4.2 Inner ear processing

This module groups properties and processes of the inner ear, namely peripheral filtering and envelope extraction.

### 4.2.1 Filterbanks

All filterbanks provide a `filter()` method that takes only the input signal. The filterbank's parameters must be defined when creating the filterbank. For example, here we create a Gammatone filterbank for a sampling frequency of 44.1 kHz and a sequence of octave-spaced center frequencies:

```
>>> import numpy as np
>>> from pambox.inner import GammatoneFilterbank
>>> g = GammatoneFilterbank(44100, [250, 500, 1000, 2000, 4000])
>>> x = np.random.randn(2 * 44100)
>>> y = g.filter(x)
>>> y.shape
(5, 88200)
```

- `GammatoneFilterbank` is a gammatone filterbank which uses Malcom Slaney's implementation.
- `RectangularFilterbank` performs bandpass filtering of a signal using rectangular filters.

### 4.2.2 Envelope extraction

- *`hilbert_envelope()`* extracts the Hilbert envelope of a signal.
- *`lowpass_env_filtering()`* low-pass filters a signal using a Butterworth filter.

### 4.2.3 Other functions

- *`erb_bandwidth()`* gives the ERB bandwidth for a given center frequencies.

### 4.2.4 API

*`pambox.inner`* regroups processes of the inner ear.

`pambox.inner.`**`erb_bandwidth`**`(`*fc*`)`

Bandwitdh of an Equivalent Rectangular Bandwidth (ERB).

**Parameters** **fc** : ndarray

Center frequency, or center frequencies, of the filter.

**Returns** ndarray or float

Equivalent rectangular bandwidth of the filter(s).

`pambox.inner.`**`hilbert_envelope`**`(`*signal*`)`

Calculates the Hilbert envelope of a signal.

**Parameters** **signal** : array_like

Signal on which to calculate the hilbert envelope. The calculation is done along the last axis (i.e. `axis=-1`).

**Returns** ndarray

`pambox.inner.lowpass_env_filtering(x, cutoff=150.0, n=1, fs=22050)`

Low-pass filters a signal using a Butterworth filter.

**Parameters** **x** : ndarray

**cutoff** : float, optional

Cut-off frequency of the low-pass filter, in Hz. The default is 150 Hz.

**n** : int, optional

Order of the low-pass filter. The default is 1.

**fs** : float, optional

Sampling frequency of the signal to filter. The default is 22050 Hz.

**Returns** ndarray

Low-pass filtered signal.

## 4.3 Middle ear processes

There's nothing here right now.

### 4.3.1 API

`pambox.periph.middle` regroups processes of the middle ear.

## 4.4 Outer ear processes

There's nothing here right now, but there should be some things to access HRTF databases pretty soon.

### 4.4.1 API

`pambox.periph.outer` regroups processes of the outer ear.

## 4.5 Central auditory processing

The `central` module regroups what is *not* considered to be part of the outer, middle, or inner ear. It's a rather broad concept.

It contains:

- An Equalization–Cancellation (EC) stage, in `EC`.
- An implementation of the EPSM modulation filterbank in `EPSMModulationFilterbank`.
- An Ideal Observer, `IdealObs`, as used in the `Sepsm` model.

### 4.5.1 API

`central` contains processes performed by the 'central' auditory system.

## Classes

- *EC* – An Equalization–Cancellation stage, as used by *[wan2014]*.
- *EPSMModulationFilterbank* – EPSM modulation filterbank, as used by *[jorgensen2011]*.
- *IdealObs* – An IdealObserver, as used by *[jorgensen2011]*.

**class** pambox.central.**EC**(*fs*, *win_len=None*, *overlap=0.5*, *sigma_e=0.25*, *sigma_d=0.000105*, *padding_windows=10*, *fast_cancel=True*)

Equalization-Cancellation process used by the STEC model *[wan2014]*.

The *equalize* method finds the optimal gains and delays that minimizes the energy of the cancelled signal.

The *cancel* method uses the gains and delays found by the *equalize* method to "cancel" the two signals.

The *jitter* method applies amplitude and time jitters to the input as a form of internal noise.

### References

*[wan2014]*

### Examples

```
>>> ec = EC()
>>> alphas, taus = ec.equalize(left, right, cf)
>>> y = ec.cancel(left, right, alphas, taus)
```

**static apply_jitter**(*x*, *epsilons*, *deltas*, *out=None*)

Apply jitter to a signal

**Parameters** **x** : ndarray

Input signal.

**epsilons** : ndarray of floats

Amplitude jitter coefficients.

**deltas** : ndarray of ints

Time jitters, they have to be integers because they will be used as indices.

**out** : array or None

Array where to write the output. If None, which is the default, the function returns a new array.

**Returns** **out** : ndarray

Jittered signal.

**cancel**(*left*, *right*, *alpha*, *tau*)

Cancel left and right signal using gains and delays.

**Parameters** **left, right** : array_like

Signals for which to find the optimal parameters. They can be 1D or 2D. If they are 2D, the signals are cancelled along the last dimension.

**alpha** : ndarray

Optimal amplitude cancellation gains.

**tau** : ndarray

Optimal cancellation delays.

**Returns** **y** : ndarray

**create_jitter**(*x*)

Create amplitude and time jitter for a signal.

**Parameters** **x** : ndarray

Input signal.

**Returns** **alphas** : ndarray of floats

Amplitude jitters.

**deltas** : ndarray of ints

Jitter indices.

**equalize**(*left*, *right*, *cf*)

Finds the optimal gains and delays that minimize the energy of the cancelled signals.

**Parameters** **left, right** : ndarrays

Signals for which to find the optimal parameters. They can be 1D or 2D. If they are 2D, the signals are cancelled along the last dimension.

**cf** : float or list of floats

Center frequency of the channel at which the equalization takes place. If the inputs are multi-channel, then cf must be a list of center frequencies.

**Returns** **alphas** : ndarray

Optimal gains. The shape depends on the input signals and on the *win_len* and `overlap` attributes.

**taus** : ndarrays

Optimal delays in seconds. The shape depends on the input signals and on the `win_len` and *overlap* attributes.

**jitter**(*x*, *out=None*)

Applies amplitude and time jitter to a signal.

**Parameters** **x** : array_like

Input signal, will be casted to 'float'. It can be one or 2 dimensional.

**out** : None or array_like

Define where to write the jitter signal. Defaults to *None*, i.e. creates a new array. Can be used to jitter an array "in place".

**Returns** **out** : ndarray

Jittered signal.

### Notes

The amplitude jitters are taken from a normal Gaussian distribution with a mean of zero and a standard distribution of `sigma_e`. The time jitters are taken from a normal Gaussian distribution with mean zero and standard distribution `sigma_d` in seconds. The default jitter values come from *[durlach1963]*.

### References

*[durlach1963]*

**class** `pambox.central.`**`EPSMModulationFilterbank`**(*fs*, *modf*, *q=1.0*, *low_pass_order=3.0*)

Implementation of the EPSM modulation filterbank.

**Parameters** **fs** : int

Sampling frequency of the signal.

**modf** : array_like

List of the center frequencies of the modulation filterbank.

**q** : float

Q-factor of the modulation filters. Defaults to 1.

**low_pass_order** : float

Order of the low-pass filter. Defaults to 3.

### Notes

The envelope power spectrum model (EPSM) filterbank was defined in *[ewert2000]* and the implementation was validated against the Matlab implementation of *[jorgensen2011]*.

### References

*[ewert2000]*, *[jorgensen2011]*

### Methods

| | |
|---|---|
| filter(signal) | Filters the signal using the modulation filterbank. |

**`filter`**(*signal*)

**Parameters** **signal** : ndarray

Temporal envelope of a signal

**Returns**

**——-**

**tuple of ndarray**

Integrated power spectrum at the output of each filter Filtered time signals.

**class** `pambox.central.`**`IdealObs`**(*k=<Mock name='mock.sqrt()' id='140421556783120'>*, *q=0.5*, *sigma_s=0.6*, *m=8000.0*)

Statistical ideal observer.

Converts input values (usually SNRenv) to a percentage.

**Parameters** **k** : float, optional

(Default value = sqrt(1.2))

**q** : float, optional

(Default value = 0.5)

**sigma_s** : float, optional

(Default value = 0.6)

**m** : int, optional

Number of words in the vocabulary. (Default value = 8000)

### Notes

Implemented as described in *[jorgensen2011]*.

### Examples

Converting values to percent correct using the default parameters of the ideal observer:

```
>>> from pambox import central
>>> obs = central.IdealObs()
>>> obs.transform((0, 1, 2, 3))
```

**fit_obs** (*values*, *pcdata*, *sigma_s=None*, *m=None*, *tries=10*)

Finds the parameters of the ideal observer.

Finds the paramaters `k`, `q`, and `sigma_s`, that minimize the least-square error between a data set and transformed SNRenv.

By default the `m` parameter is fixed and the property `m` is used. It can also be defined as an optional parameter.

It is also possible to fix the *sigma_s* parameter by passing it as an optional argument. Otherwise, it is optimized with *k* and *q*.

**Parameters** **values** : ndarray

The linear SNRenv values that are to be converted to percent correct.

**pcdata** : ndarray

The data, in percentage between 0 and 1, of correctly understood tokens. Must be the same shape as *values*.

**sigma_s** : float, optional

(Default value = None)

**m** : float, optional

(Default value = None)

**tries** : int, optional

How many attempts to fit the observer if the start values do not converge. The default is 10 times.

**Returns** self

**get_params** ()

Returns the parameters of the ideal observer as dict.

**Parameters** **None**

**Returns** **params** : dict

Dictionary of internal parameters of the ideal observer.

**transform**(*values*)

Converts inputs values to a percent correct.

**Parameters** **values** : array_like

Linear values to transform.

**Returns** **pc** : ndarray

Array of intelligibility percentage values, of the same shape as *values*.

## 4.6 Speech Intelligibility Models and Experiments

### 4.6.1 Introduction

The `speech` module groups together speech intelligibility models and other tools to facilitate the creation of speech intelligibility prediction "experiments".

### 4.6.2 Speech Intelligibility Models

Speech intelligibility models are classes that take at least a `fs` argument. All predictions are done via a `predict` method with the signature: `predict(clean=None, mix=None, noise=None)`. This signature allows models to require only a subset of the inputs. For example, blind models might only require the mixture of processed speech and noise: `predict(mix=noisy_speech)`; or just the clean signal and the noise: `predict(clean=speech, noise=noise)`.

The reference level is that a signal with an RMS value of 1 corresponds to 0 dB SPL.

Here is a small example, assuming that we have access to two signals, `mix` which is a mixture of the clean speech and the noise, and `noise`, which is the noise alone.

```
>>> from pambox.speech import Sepsm
>>> s = Sepsm(fs=22050)
>>> res = s.predict(mix=mix, noise=noise)
```

For models that do not take time signals as inputs, such as the `Sii`, two other types of interfaces are defined:

- `predict_spec` if the model takes frequency spectra as its inputs. Once again, the spectra of the clean speech, of the mixture, and of the noise should be provided:

```
>>> from pambox.speech import Sii
>>> s = Sii(fs=22050)
>>> res = s.predict_spec(clean=clean_spec, noise=noise_spec)
```

- `predict_ir` if the models takes impulse responses as its inputs. The function then takes two inputs, the impulse response to the target, and the concatenated impulse responses to the maskers:

```
>>> from pambox.speech import IrModel
>>> s = IrModel(fs=22050)
>>> res = s.predict_ir(clean_ir, noise_irs)
```

Intelligibility models return a dictionary with **at least** the following key:

- p (for "predictions"): which is a dictionary with the outputs of the model. They keys are the names of the outputs. This allows models to have multiple return values. For example, the *MrSepsm* returns two prediction values:

```
>>> s = MrSepsm(fs=22050)
>>> res = s.predict(clean, mix, noise)
>>> res['p']
{'lt_snr_env': 10.5, 'snr_env': 20.5}
```

It might seem a bit over-complicated, but it allows for an easier storing of the results of an experiment.

Additionally, the models can add any other keys to the results dictionary. For example, a model can return some of its internal attributes, its internal representation, etc.

### 4.6.3 Speech Materials

The *Material* class simplifies the access to the speech files when doing speech intelligibility prediction experiments.

When creating the class, you have to define:

- where the sentences can be found
- their sampling frequency
- their reference level, in dB SPL (the reference is that a signal with an RMS value of 1 corresponds to 0 dB SPL),
- as well as the path to a file where the corresponding speech-shaped noise for this particular material can be found.

For example, to create a speech material object for IEEE sentences stored in the *../stimuli/ieee* folder:

```
>>> sm = SpeechMaterial(
...     fs=25000,
...     path_to_sentences='../stimuli/ieee',
...     path_to_ssn='ieee_ssn.wav',
...     ref_level=74
...     name='IEEE'
...     )
```

Each speech file can be loaded using its name:

```
>>> x = sm.load_file(sm.files[0])
```

Or files can be loaded as an iterator:

```
>>> all_files = sm.load_files()
>>> for x in all_files:
...     # do some processing on `x`
...     pass
```

By default, the list of files is simply all the files found in the path_to_sentences. To overwrite this behavior, simply replace the *files_list()* function:

```
>>> def new_files_list():
...     return ['file1.wav', 'file2.wav']
>>> sm.files_list = new_files_list
```

It is common that individual sentences of a speech material are not adjusted to the exact same level. This is typically done to compensate for differences in intelligibility between sentences. In order to keep the inter-sentence level difference, it is recommended to use the *set_level()* method of the speech material. The code below sets the level of the first sentence to 65 dB SPL, with the reference that a signal with an RMS value of 1 has a level of 0 dB SPL.

```
>>> x = sm.load_file(sm.files[0])
>>> adjusted_x = sm.set_level(x, 65)
```

Accessing the speech-shaped noise corresponding the speech material is done using the `ssn()` function:

```
>>> ieee_ssn = sm.ssn()
```

By default, this will return the entirety of the SSN. However, it is often required to select a section of noise that is the same length as a target speech signal, therefore, you can get a random portion of the SSN of the same length as the signal *x* using:

```
>>> ssn_section = sm.ssn(x)
```

If you are given a speech material but you don't know it's average level, you can use the help function `average_level()` to find the average leve, in dB, of all the sentences in the speech material:

```
>>> average_level = sm.average_level()
```

### 4.6.4 Speech Intelligibility Experiment

Performing speech intelligibility experiments usually involves a tedious process of looping through all conditions to study, such as different SNRs, processing conditions, and sentences. The `Experiment` class simplifies and automates the process of going through all the experimental conditions. It also gathers all the results in a way that is simple to manipulate, transform, and plot.

#### Basic Example

An experiment requires at least: a model, a speech material, and a list of SNRs.

```
>>> from pambox.speech import Experiment, Sepsm, Material
>>> models = Sepsm()
>>> material = Material()
>>> snrs = np.arange(-9,-5, 3)
>>> exp = Experiment(models, material, snrs, write=False)
>>> df = exp.run(2)
>>> df
 Distortion params   Model    Output  SNR  Sentence number      Value
0                None   Sepsm   snr_env   -9                0   1.432468
1                None   Sepsm   snr_env   -6                0   5.165170
2                None   Sepsm   snr_env   -9                1   6.308387
3                None   Sepsm   snr_env   -6                1  10.314227
```

Additionally, you can assign a type of processing, such as reverberation, spectral subtraction, or any arbitrary type of processing. To keep things simply, let's apply a compression to the mixture and to the noise. Your distortion function *must return* the clean speech, the mixture, and the noise alone.

```
>>> def compress(clean, noise, power):
...     mixture = (clean + noise) ** (1 / power)
...     noise = noise ** (1 / power)
...     return clean, mixture, noise
...
>>> powers = range(1, 4)
>>> exp = Experiment(models, material, snrs, mix_signals, powers)
>>> df = exp.run(2)
>>> df
```

If the distortion parameters are stored in a list of dictionaries, they will be saved in separate columns in the output dataframe. Otherwise, they will be saved as tuples in the "Distortion params" column.

### 4.6.5 API

The `pambox.speech` module gather speech intelligibility models, a framework to run intelligibility experiments, as well as a wrapper around speech materials.

**class** `pambox.speech.`**`Sepsm`**(*fs=22050*, *cf=(63*, *80*, *100*, *125*, *160*, *200*, *250*, *315*, *400*, *500*, *630*, *800*, *1000*, *1250*, *1600*, *2000*, *2500*, *3150*, *4000*, *5000*, *6300*, *8000)*, *modf=(1.0*, *2.0*, *4.0*, *8.0*, *16.0*, *32.0*, *64.0)*, *downsamp_factor=10*, *noise_floor=0.01*, *snr_env_limit=0.001*, *name='sEPSM'*)

Implement the sEPSM intelligibility model [1].

**Parameters** **fs** : int

(Default value = 22050)

**cf** : array_like

(Default value = _default_center_cf)

**modf** : array_like

(Default value = _default_modf)

**downsamp_factor** : int

(Default value = 10)

**noise_floor** : float

(Default value = 0.01)

**snr_env_limit** : float

(Default value = 0.001)

#### Notes

Modifed on 2014-10-07 by Alexandre Chabot-Leclerc: Remove the unnecessary factor to compensate for filter bandwidth when computing the bands above threshold. The diffuse hearing threshold are already adjusted for filter bandwidth.

#### References

*[R2]*

**`plot_bands_above_thres`**(*res*)

Plot bands that were above threshold as a bar chart.

**Parameters** **res** : dict

A *dict* as output by the sEPSM model. The dictionary must have a *bands_above_threshold_idx* key.

**Returns** None

**`plot_exc_ptns`**(*res*, *db=True*, *attr='exc_ptns'*, *vmin=None*, *vmax=None*)

Plot the excitation patterns from a prediction.

**Parameters** **res** : dict

Results from an sEPSM prediction. The dictionay should have a "exc_ptns" key. Otherwise, the key to use can be defined using the *attr* parameter.

**db** : bool, optional, (Default value = *True*)

Plot as dB if *True*, otherwise plots linear values.

**attr** : string, optional, (Default value = 'exc_ptns')

Dictionary key to use for plotting the excitation patters

**vmin, vmax** : float, optional, (Default = None)

Minimum and maximum value to normalize the color scale.

**`plot_filtered_envs`**(*envs*, *fs*, *axes=None*)

Plot the filtered envelopes.

**Parameters** **envs** : ndarray

List of envelope signals.

**fs** : int

Sampling frequency.

**axes** : axes, (Default value = None)

Matplotlib axes where to place the plot. Defaults to creating a new figure is *None*.

**`plot_snr_env_matrix`**(*res*, *ax=None*, *vmin=None*, *vmax=None*)

**Parameters** **res** : dict

Output of the `predict()` function.

**ax** : ax object

Matplotlib axis where the data should be plotted. A new axis will be created if the value is *None*. (Default value = None)

**vmin** : float, optional

Minimum value of the heatmanp. The minimum value will be infered from the data if *None*. (Default value = None)

**vmax** : float, optional

Maxiumum value of the heatmanp. The maximum value will be infered from the data if *None*. (Default value = None)

**`predict`**(*clean=None*, *mix=None*, *noise=None*)

Predicts intelligibility.

The sEPSM requires at least the mixture and the noise alone to make a prediction. The clean signal will also be processed if it is available, but it is not used to make the prediction.

**Parameters** **clean** : ndarray (optional)

Clean speech signal, optional.

**mix** : ndarray

Mixture of the processed speech and noise.

**noise** : ndarrays

Processed noise signal alone.

**Returns** **res** : dict

Dictionary of the model predictions. The keys are as follow: - 'p': is a dictionary with the model predictions. In this case it contains a 'snr_env' key. - 'snr_env_matrix': 2D matrix of the SNRenv as a function audio frequency and modulation frequency. - 'exc_ptns': Modulation power at the output of the modulation filterbank for the intput signals. It is a (N_SIG, N_CHAN, N_MODF) array. - 'band_above_thres_idx': Array of the indexes of the bands that were above hearing threshold.

**class** `pambox.speech.`**`MrSepsm`**(*fs=22050*, *cf=(63*, *80*, *100*, *125*, *160*, *200*, *250*, *315*, *400*, *500*, *630*, *800*, *1000*, *1250*, *1600*, *2000*, *2500*, *3150*, *4000*, *5000*, *6300*, *8000)*, *modf=(1.0*, *2.0*, *4.0*, *8.0*, *16.0*, *32.0*, *64.0*, *128.0*, *256.0)*, *downsamp_factor=10*, *noise_floor=0.001*, *snr_env_limit=0.001*, *snr_env_ceil=None*, *min_win=None*, *name='MrSepsm'*, *output_time_signals=False*)

Multi-resolution envelope power spectrum model (mr-sEPSM).

**Parameters** **fs** : int, optional, (Default value = 22050)

Sampling frequency.

**cf** : array_like, optional

Center frequency of the cochlear filters.

**modf** : array_like, optional (Default value = _default_modf)

Center frequency of modulation filters.

**downsamp_factor** : int, optional, (Default value = 10)

Envelope downsampling factor. Simply used to make calculattion faster.

**noise_floor** : float, optional, (Default value = 0.001)

Value of the internal noise floor of the model. The default is -30 dB.

**snr_env_limit** : float, optional, (Default value = 0.001)

Lower limit of the SNRenv values. Default is -30 dB.

**snr_env_ceil** : float, optional, (Default value = None)

Upper limit of the SNRenv. No limit is applied if *None*.

**min_win** : float, optional, (Default value = None)

Minimal duration of the multi-resolution windows, in ms.

**name** : string, optional, (Default value = 'MrSepsm')

Name of the model.

**output_time_signals** : bool, optional

Output the time signals signals in the results dictionary. Adds the keys 'chan_sigs', 'chan_envs', and 'filtered_envs'.

## References

*[jorgensen2013multi]*

**`plot_mr_exc_ptns`** (*ptns*, *dur=None*, *db=True*, *vmin=None*, *vmax=None*, *fig_subplt=None*, *attr='exc_ptns'*, *add_cbar=True*, *add_ylabel=True*, *title=None*)

Plots multi-resolution representation of envelope powers.

**Parameters** **ptns** : dict

Predictions from the model. Must have a *mr_snr_env_matrix* key.

**dur** : bool

Display dB values of the modulation power or SNRenv values. (Default: True.)

**vmax** : float

Maximum value of the colormap. If *None*, the data's maxium value is used. (Default: None)

**db** : bool

Plot the values in dB. (Default value = True)

**vmin** : float

Minimum value of the heatmap. The value will be infered from the data if *None*. (Default value = None)

**fig_subplt** : tuple of (fig, axes)

Matplotlib *figure* and *axes* objects where the data should be plotted. If *None* is provided, a new figures with the necessary axes will be created. (Default value = None)

**attr** : string

Key to query in the *ptns* dictionary. (Default value = 'exc_ptns')

**add_cbar** : bool

Add a colorbar to the figure. (Default value = True)

**add_ylabel** : bool

Add a y-label to the axis. (Default value = True)

**title** : bool

Add a title to the axis. (Default value = None)

**`predict`** (*clean=None*, *mix=None*, *noise=None*)

Predicts intelligibility using the mr-sEPSM.

The mr-sEPSM requires at least the mix and the noise alone to make a prediction. The clean signal will also be processed if it is available, but it is not used to make the prediction.

**Parameters** **clean** : ndarray (optional)

Clean speech signal, optional.

**mix** : ndarray

Mixture of the processed speech and noise.

**noise** : ndarrays

**Returns** dict

Dictionary with the predictions by the model.

**class** `pambox.speech.`**`BsEPSM`**(*fs=22050*, *name='BinauralMrSepsm'*, *cf=(63*, *80*, *100*, *125*, *160*, *200*, *250*, *315*, *400*, *500*, *630*, *800*, *1000*, *1250*, *1600*, *2000*, *2500*, *3150*, *4000*, *5000*, *6300*, *8000)*, *modf=(1.0*, *2.0*, *4.0*, *8.0*, *16.0*, *32.0*, *64.0*, *128.0*, *256.0)*, *downsamp_factor=10*, *noise_floor=0.001*, *snr_env_limit=0.001*, *sigma_e=0.25*, *sigma_d=0.000105*, *fast_cancel=True*, *debug=False*, *win_len=0.02*, *ec_padding_windows=10*)

Binaural implementation of the sEPSM model.

Implementation used in *[chabot-leclerc2016]*.

### References

*[chabot-leclerc2016]*

**`plot_tau_hist`**(*res*, *cfs=None*, *bins=None*, *return_ax=False*)

Plot histogram of tau values.

**Parameters** **res** : dict

Results from the *predict* function.

**cfs** : list

Index of center frequencies to plot.

**bins** : int

Number of bins in the histogram. If *None*, uses bins between -700 us and 700 us. Default is *None*.

**return_ax** : bool, optional

If True, returns the figure Axes. Default is False.

**`predict`**(*clean=None*, *mixture=None*, *noise=None*)

Predict intelligibility.

**Parameters** **clean, mixture, noise** : ndarray

Binaural input signals.

**Returns** **res** : dict

Model predictions and internal values. Model predictions are stored as a dictionary under the key *'p'*.

**class** `pambox.speech.`**`SII`**(*T=0.0*, *I=0*)

Speech intelligibility index model, *[ansi1997sii]*.

**Parameters** **T** : float or array_like, optional, (Default is 0)

Hearing threshold. 18 values in dB HL or a single value.

**I** : int, optional, (Default is 0, normal speech)

Band importance function selector. See Notes section below.

### Notes

Arguments for 'I':

A scalar having a value of either 0, 1, 2, 3, 4, 5, or 6. The Band-importance functions associated with each scalar are:

- 0: Average speech as specified in Table 3 (DEFAULT)
- **1: various nonsense syllable tests where most English phonemes occur** equally often (as specified in Table B.2)
- 2: CID-22 (as specified in Table B.2)
- 3: NU6 (as specified in Table B.2)
- 4: Diagnostic Rhyme test (as specified in Table B.2)
- **5: short passages of easy reading material (as specified in** Table B.2)
- 6: SPIN (as specified in Table B.2)

### References

*[ansi1997sii]*

**`predict_spec`**(*clean=None*, *mix=None*, *noise=-50*)

Predicts intelligibility based on the spectra levels of the speech and the noise.

**Parameters clean: array_like or float**

Speech level in dB SPL. Equivalent to “E” in *[ansi1997sii]*.

**mix** : ignored

This argument is present only to conform to the API.

**noise: array_like, optional, (Default is -50 dB SPL)**

Noise level in dB SPL. Equivalent to N in *[ansi1997sii]*.

**Returns** res: dict

Dictionary with prediction values under res[’p’][’sii’].

**class** `pambox.speech.`**`Material`**(*fs=22050*, *path_to_sentences=’../stimuli/clue/sentencesWAV22’*, *path_to_maskers=None*, *path_to_ssn=’../stimuli/clue/SSN_CLUE22.wav’*, *ref_level=74*, *name=’CLUE’*, *force_mono=False*)

Load and manipulate speech materials for intelligibility experiments

**`average_level`**()

Calculate the average level across all sentences.

The levels are calculated according to the toolbox’s reference level.

**Returns mean** : float

Mean level across all sentences, in dB SPL.

**std** : float

Standard deviation of the levels across all sentences.

**See also:**

`utils.dbspl`

**`create_filtered_ssn`**(*files=None*, *duration=5*)

Create speech-shaped noise based on the average long-term spectrum of the speech material.

**Parameters files** : list, optional

List of files to concatenate. Each file should be an *ndarray*. If *files* is None, all the files from the speech material will be used. They are loaded with the method *load_files()*.

**duration** : float, optional

Duration of the noise, in seconds. The default is 5 seconds.

**Returns ssn** : ndarray

**create_ssn**(*files=None*, *repetitions=200*)

Creates a speech-shaped noise from the sentences.

Creates a speech-shaped noise by randomly adding together sentences from the speech material. The output noise is 75% the length of all concatenated sentences.

**Parameters files** : list, optional

List of files to concatenate. Each file should be an *ndarray*. If *files* is None, all the files from the speech material will be used. They are loaded with the method *load_files()*.

**repetitions** : int

Number of times to superimpose the randomized sentences. The default is 120 times.

**Returns ssn** : ndarray

### Notes

Before each addition, the random stream of sentences is jittered to prevent perfect alignment of all sentences. The maximum jitter is equal to 25% of the length of the concatenated sentences.

**files_list**()

Return a list of all the .wav files in the *path_to_sentences* directory.

**Returns files** : list

List of all files.

**load_file**(*filename*)

Read a speech file by name.

**Parameters filename** : string

Name of the file to read. The file just be in the directory defined by *root_path* and *path_to_sentences*.

**Returns** ndarray

Wav file read from disk, as floating point array.

**load_files**(*n=None*)

Read files from disk, starting from the first one.

**Parameters n** : int, optional

Number of files to read. Default (*None*) is to read all files.

**Returns** generator

Generator where each item is an *ndarray* of the file loaded.

**static pick_section**(*signal*, *section=None*)

Pick section of signal

**Parameters section** : int or ndarray, optional

If an integer is given, returns section of length *n* Alternatively, if *section* is an ndarray the signal returned will be of the same length as the *section* signal. If *x* is *None*, the full signal is returned.

**Returns**

**——-**

**ndarray**

Speech-shaped noise signal.

**`set_level`**(*x*, *level*)

Set level of a sentence, in dB.

**Parameters** **x** : ndarray

sentence

**level** : float

Level, in dB, at which the sentences are recorded. The reference is that and RMS of 1 corresponds to 0 dB SPL.

**Returns** array_like

Adjusted sentences with a *level* db SPL with the reference that a signal with an RMS of 1 corresponds to 0 db SPL.

**`ssn`**(*x=None*)

Returns the speech-shaped noise appropriate for the speech material.

**Parameters** **x** : int or ndarray, optional

If an integer is given, returns a speech-shaped noise of length *n* Alternatively, if a sentenced is given, the speech-shaped noise returned will be of the same length as the input signal. If *x* is *None*, the full SSN signal is returned.

**Returns**

**——-**

**ndarray**

Speech-shaped noise signal.

**class** `pambox.speech.`**`Experiment`**(*models*, *material*, *snrs*, *distortion=None*, *dist_params=(None, )*, *fixed_level=65*, *fixed_target=True*, *name=None*, *write=True*, *output_path='./output/'*, *timestamp_format='%Y%m%d-%H%M%S'*, *adjust_levels_bef_proc=False*)

Performs a speech intelligibility experiment.

The masker is truncated to the length of the target, or is padded with zeros if it is shorter than the target.

**Parameters** **models** : single model, or list

List of intelligibility models.

**materials** : object

object that implements a *next* interface that returns the next pair of target and maskers.

**snrs** : array_like

List of SNRs.

**name** : str,

name of experiment, will be appended to the date when writing to file.

**write** : bool, optional

Write the result to file, as CSV, the default is True.

**output_path** : string, optional.

Path where the results will be written if *write* is True. The default is './output'.

**timestamp_format** : str, optional

Datetime timestamp format for the CSV file name. The default is of the form YYYYMMDD-HHMMSS.

**`adjust_levels`** (*target*, *masker*, *snr*)
Adjusts level of target and maskers.

Uses the *self.fixed_level* as the reference level for the target and masker. If *self.fixed_target* is True, the masker level is varied to set the required SNR, otherwise the target level is changed.

**Parameters**

- **`target`** – ndarray Target signal.
- **`masker`** – ndarray Masker signal.
- **`snr`** – float SNR at which to set the target and masker.

**Returns** tuple Level adjusted *target* and *masker*.

**`append_results`** (*df*, *res*, *model*, *snr*, *i_target*, *params*, ***kwargs*)
Appends results to a DataFrame

**Parameters** **df** : dataframe

DataFrame where the new results will be appended.

**res** : dict

Output dictionary from an intelligibility model.

**model: object**

Intelligibility model. Will use it's *name* attribute, if available, to add the source model to the DataFrame. Otherwise, the __*class*__.__*name*__ attribute will be used.

**snr** : float

SNR at which the simulation was performed.

**i_target** : int

Number of the target sentence

**params** : object

Parameters that were passed to the distortion process.

**Returns** **df** : dataframe

DataFrame with new entry appended.

**classmethod `pred_to_pc`** (*df*, *fc*, *col='Value'*, *models=None*, *out_name='Intelligibility'*)
Converts the data in a given column to percent correct.

**Parameters** **df** : Dataframe

Dataframe where the intelligibility predictions are stored.

**fc** : function

The function used to convert the model outputs to intelligibility. The function must take a float as input and returns a float.

**col** : string

Name of the column to convert to intelligibility. The default is "Value".

**models** : string, list or dict

This argument can either be a string, with the name of the model for which the output value will be transformed to intelligibility, or a list of model names. The argument can also be a dictionary where the keys are model names and the values are "output names", i.e. the name of the value output by the model. This is useful if a model has multiple prediction values. The default is *None*, all the rows will be converted with the same function.

**out_name** : str

Name of the output column (default: 'Intelligibility')

**Returns df** : dataframe

Dataframe with the new column column with intelligibility values.

**static `prediction`** (*model*, *target*, *mix*, *masker*)

Predicts intelligibility for a target and masker pair. The target and masker are simply added together to create the mixture.

**Parameters model :**

**target :**

**masker :**

**Returns**

**return**

**`preprocessing`** (*target*, *masker*, *snr*, *params*)

Applies preprocessing to the target and masker before setting the levels. In this case, the masker is padded with zeros if it is longer than the target, or it is truncated to be the same length as the target.

**Parameters**

- **`target`** –
- **`masker`** –

**Returns**

**`run`** (*n=None*, *seed=0*, *parallel=False*, *profile=None*, *output_filename=None*)

Run the experiment.

**Parameters n** : int

Number of sentences to process.

**seed** : int

Seed for the random number generator. Default is 0.

**parallel** : bool

If False, the experiment is ran locally, using a for-loop. If True, we use IPython.parallel to run the experiment in parallel. We try to connect to the current profile.

**output_filename** : string

Name of the output file where the results will be saved. If the filename contains directories, they will be created in self.output_path. If it is *None*, the default is to use the current date and time. The default is *None*.

**Returns df** : pd.Dataframe

Pandas dataframe with the experimental results.

**classmethod** **`srts_from_df`**(*df*, *col='Intelligibility'*, *srt_at=50*, *model_srts=None*)

Get dataframe with SRTs

**Parameters df** : Data Frame

DataFrame resulting from an experiment. It must have an "Intelligibility" column.

**col** : string (optional)

Name of the column to use for the SRT calculation. The default value is the 'Intelligibility' column.

**srt_at** : float, tuple (optional)

Value corresponding to the SRT. The default is 50 (%).

**model_srts** : dict

Overrides default `srt_at` for particular models. The dictionary must be a tuple of the model name and model output: ('Model', 'Output')

**Returns**

**——-**

**out** : Data frame

Data frame, with an SRT column.

## 4.7 Signal Distortion and Processing

The `distort` module groups together various distortions and types of processing that can be applied to signals.

- `mix_noise()` adds together two signals at a given SNR.
- `noise_from_signal()` creates a noise with the same spectrum as the input signal. Optionally, it can also keep the signal's envelope.
- `overlap_and_add()` reconstructs a signal using the overlap and add method.
- `phase_jitter()` applies phase jitter to a signal.
- `spec_sub()` applies spectral subtraction to a signal.

### 4.7.1 API

`pambox.distort` regroups various types of distortions and processings that can be applied to signals.

**class** `pambox.distort.`**`WestermannCrm`**(*fs=40000*)

Applies HRTF and BRIR for a given target and masker distance.

**Parameters fs** : int

Samping frequenc of the process. (Default value = 40000)

### References

*[R1]*

### Attributes

| | |
|---|---|
| brir | (dict) Binaural room impulse responses for each distance. |
| delays | (dict) Delay until the first peak in the BRIR for each distance. |
| dist | (ndarray) List of the valid distances (0.5, 2, 5, and 10 meters). |

**apply** (*x*, *m*, *tdist*, *mdist*, *align=True*)

Applies the "Westermann" distortion to a target and masker.

target and masker are not co-located, the masker is equalized before applying the BRIR, so that both the target and masker will have the same average spectrum after the BRIR filtering.

By default, the delay introduced by the BRIR is compensated for, such that the maxiumum of the BRIR happen simulatenously.

**Parameters** **x** : ndarray

Mono clean speech signal of length *N*.

**m** : ndarray

Mono masker signal of length *N*.

**tdist** : float

Target distance, in meters.

**mdist** : float

Masker distance, in meters.

**align** : bool

Compensate for the delay in the BRIRs with distance (default is *True*).

**Returns** **mix** : (2, N) ndarray

Mixture processesed by the BRIRs.

**noise** : (2, N)

Noise alone processed by the BRIRs.

pambox.distort.**mix_noise** (*clean*, *noise*, *sent_level*, *snr=None*)

Mix a signal signal noise at a given signal-to-noise ratio.

**Parameters** **clean** : ndarray

Clean signal.

**noise** : ndarray

Noise signal.

**sent_level** : float

Sentence level, in dB SPL.

**snr :**

Signal-to-noise ratio at which to mix the signals, in dB. If snr is *None*, no noise is mixed with the signal (Default value = None)

**Returns** tuple of ndarrays

Returns the clean signal, the mixture, and the noise.

`pambox.distort.`**`noise_from_signal`**(*x*, *fs=40000*, *keep_env=False*)

Create a noise with same spectrum as the input signal.

**Parameters** **x** : array_like

Input signal.

**fs** : int

Sampling frequency of the input signal. (Default value = 40000)

**keep_env** : bool

Apply the envelope of the original signal to the noise. (Default value = False)

**Returns** ndarray

Noise signal.

`pambox.distort.`**`overlap_and_add`**(*powers*, *phases*, *len_window*, *shift_size*)

Reconstruct a signal with the overlap and add method.

**Parameters** **powers** : ndarray

Magnitude of the power spectrum of the signal to reconstruct.

**phases** : ndarray

Phase of the signal to reconstruct.

**len_window** : int

Frame length, in samples.

**shift_size** : int

Shift length. For non overlapping signals, in would equal *len_window*. For 50% overlapping signals, it would be *len_window/2*.

**Returns** ndarray

Reconstructed time-domain signal.

`pambox.distort.`**`phase_jitter`**(*x*, *a*)

Apply phase jitter to a signal.

The expression of phase jitter is:

$$y(t) = s(t) * cos(\Phi(t)),$$

where $\Phi(t)$ is a random process uniformly distributed over $[0, 2\pi\alpha]$. The effect of the jitter when $\alpha$ is 0.5 or 1 is to completely destroy the carrier signal, effectively yielding modulated white noise.

**Parameters** **x** : ndarray

Signal

**a** : float

Phase jitter parameter, typically between 0 and 1, but it can be anything.

**Returns** ndarray

Processed signal of the same dimension as the input signal.

`pambox.distort.`**`reverb`**(*x*, *rt*)

Applies reverberation to a signal.

**Parameters** **x** : ndarray

Input signal.

**rt** : float

Reverberation time

**Returns** ndarray

Processed signal.

`pambox.distort.`**`spec_sub`**(*x*, *noise*, *factor*, *w=512.0*, *padz=512.0*, *shift_p=0.5*)

Apply spectral subtraction to a signal.

The defaul values of the parameters are typical for a sampling frequency of 44100 Hz. Note that (W+padz) is the final frame window and hence the fft length (it is normally chose as a power of 2).

**Parameters** **x** : ndarray

Input signal

**noise :**

Input noise signal

**factor** : float

Noise subtraction factor, must be larger than 0.

**w** : int

Frame length, in samples. (Default value = 1024 / 2.)

**padz** : int

Zero padding (pad with padz/2 from the left and the right) (Default value = 1024 / 2.)

**shift_p** : float

Shift percentage (overlap) between each window, in fraction of the window size (Default value = 0.5)

**Returns** **clean_estimate** : ndarray

Estimate of the clean signal.

**noise_estimate** : ndarray

Estimate of the noisy signal.

## 4.8 Utilities

The *`utils`* groups together function that are not auditory processes but that are, nonetheless, useful or essential to manipulate signals.

## 4.8.1 Signal levels

*pambox* defines a reference level for digital signals. The convention is that a signal with a root-mean-square (RMS) value of 1 corresponds to a level of 0 dB SPL. In other words:

$$L[dBSPL] = 20 * \log_{10} \frac{P}{Ref},$$

where $Ref$ is 1.

The functions `setdbspl()`, `dbspl()`, and `rms()` help in doing this conversion.

## 4.8.2 Adding signals and adjusting their lengths

Adding together signals loaded from disks is often problematic because they tend to have different lengths. The functions `add_signals()` and `make_same_length()` simplify this. The former simply adds two signals and pads the shortest one with zeros if necessary. The latter force two signals to be of the same lengths by either zero-padding the shortest (default) or by cutting the second signal to the length of the first, for example:

```
>>> a = [1, 1]
>>> b = [2, 2, 2]
>>> make_same_length(a, b)
[1, 1, 0], [2, 2, 2]
>>> make_same_length(a, b, extend_first=False)
[1, 1], [2, 2]
```

This can be useful when using models operating in the envelope domain, as padding with zeros increase the energy at low modulation frequencies.

The `int2srt()` function finds the speech reception threshold (SRT) for a given intelligibility curve. It is actually a more general linear interpolation function, but the most common use case in this toolbox is to find SRTs.

The function `psy_fn()` calculates a psychometric function based on a mean (that would be the SRT @ 50%) and a standard deviation. This function can be useful when trying to fit a psychometric function to a series of data points.

## 4.8.3 FFT Filtering and general speedups

FIR filtering is rather slow when using long impulse responses. The function `fftfilt()` makes such filtering faster by executing the filtering using the overlap-and-add method in the frequency domain rather than as a convolution. It is largely inspired from the Matlab implementation and was adapted from a suggested addition to Scipy. It might be removed from the toolbox if *fftfilt* becomes a part of Scipy.

The function `next_pow_2()` is a convenient way to obtain the next power of two for a given integer. It's mostly useful when picking an FFT length.

## 4.8.4 API

`pambox.utils.`**`add_signals`**`(`*a*, *b*`)`

Adds two vectors of different lengths by zero padding the shortest one.

**Parameters** **a,b** : ndarray

Arrays to make of the same length.

**Returns** ndarray

Sum of the signal, of the same length as the longest of the two inputs.

`pambox.utils.`**`dbspl`**(*x*, *ac=False*, *offset=0.0*, *axis=-1*)

Computes RMS value of signal in dB.

By default, a signal with an RMS value of 1 will have a level of 0 dB SPL.

**Parameters** **x** : array_like

Signal for which to caculate the sound-pressure level.

**ac** : bool

Consider only the AC component of the signal, i.e. the mean is removed (Default value = False)

**offset** : float

Reference to convert between RMS and dB SPL. (Default value = 0.0)

**axis** : int

Axis on which to compute the SPL value (Default value = -1, last axis)

**Returns** ndarray

Sound-pressure levels.

**See also:**

*`setdbspl`*, *`rms`*

### References

*[R3]*

`pambox.utils.`**`fftfilt`**(*b*, *x*, *n=None*)

FIR filtering using the FFT and the overlap-add method.

Filters the data in *x* using the FIR coefficients in *b*. If *x* is a matrix, the rows are filtered. If *b* is a matrix, each filter is applied to *x*. If both *b* and *x* are matrices with the same number of rows, each row of *x* is filtered with the respective row of *b*.

**Parameters** **b** : array_like

Coefficients of the FIR filter.

**x** : array_like

Signal to filter.

**n** : int, optional.

Length of the FFT. If *n* is not provided, a value of *n* will be chosen by *fftfilt*. See Notes for details.

**Returns** **y** : ndarray

Filtered signal.

### Notes

Filter the signal *x* with the FIR filter described by the coefficients in *b* using the overlap-add method. If the FFT length *n* is not specified, it and the overlap-add block length are selected so as to minimize the computational cost of the filtering operation.

If *x* is longer than *b*, then *n* and *L* will be chosen as to minimize the product of the number of blocks and the number of flops per FFT.

If a value of *n* is provided, the FFT length will be the next power of 2 after *n* and each block of data will be of length *N_fft - N_b + 1*. If *n* is smaller than the length of *b*, the FFT length will be the length of *b*.

### Examples

```
>>> import pambox.utils
>>> b = [1, 1]
>>> x = [0, 1, 2, 3, 4, 5]
>>> y = pambox.utils.fftfilt(b, x)
```

The FFT length can also be specified: >>> y = pambox.utils.fftfilt(b, x, 16)

`pambox.utils.hilbert`(*x*, *N=None*, *axis=-1*)

Computes the analytic signal using the Hilbert transform.

The transformation is done along the last axis by default.

**Parameters** **x** : array_like

Signal data. Must be real.

**N** : int, optional

Number of Fourier components. Default: `x.shape[axis]`

**axis** : int, optional

Axis along which to do the transformation. Default: -1.

**Returns** **xa** : ndarray

Analytic signal of *x*, of each 1-D array along *axis*

### Notes

**NOTE**: This code is a copy-paste from the Scipy codebase. By redefining it here, it is possible to take advantage of the speed increase provided by using the MKL's FFT part of Enthough's distribution.

The analytic signal `x_a(t)` of signal `x(t)` is:

$$x_a = F^{-1}(F(x)2U) = x + iy$$

where *F* is the Fourier transform, *U* the unit step function, and *y* the Hilbert transform of *x*. *[R4]*

In other words, the negative half of the frequency spectrum is zeroed out, turning the real-valued signal into a complex signal. The Hilbert transformed signal can be obtained from `np.imag(hilbert(x))`, and the original signal from `np.real(hilbert(x))`.

### References

*[R4]*

`pambox.utils.`**`impz`**(*b*, *a=1*)

Plot step and impulse response of an FIR filter.

**b** [float] Forward terms of the FIR filter.

**a** [float] Feedback terms of the FIR filter. (Default value = 1)

From http://mpastell.com/2010/01/18/fir-with-scipy/

**Returns** None

`pambox.utils.`**`int2srt`**(*x*, *y*, *srt_at=50.0*)

Finds intersection using linear interpolation.

This function finds the x values at which a curve intersects with a constant value.

**Parameters** **x** : array_like

“x” values

**y** : array_like

“y” values

**srt_at** : float

Value of *y* at which the interception is calculated. (Default value = 50.0)

**Returns** float

`pambox.utils.`**`make_same_length`**(*a*, *b*, *extend_first=True*)

Make two vectors the same length.

**Parameters** **a,b** : array_like

Arrays to make of the same length.

**extend_first** : bool, optional

Zero-pad the first array if it is the shortest if *True*. Otherwise, cut array *b* to the length of *a*. (Default value = True)

**Returns** tuple of ndarrays

Two arrays with the same length along the last dimension.

`pambox.utils.`**`mfreqz`**(*b*, *a=1*, *fs=22050.0*)

Plot the frequency and phase response of an FIR filter.

From http://mpastell.com/2010/01/18/fir-with-scipy/

**Parameters** **b** : float

Forward terms of the FIR filter.

**a** : float

Feedback terms of the FIR filter. (Default value = 1)

**fs** : float

Sampling frequency of the filter. (Default value = 22050.0)

**Returns** None

`pambox.utils.`**`next_pow_2`**(*x*)

Calculates the next power of 2 of a number.

**Parameters** **x** : float

Number for which to calculate the next power of 2.

**Returns** int

`pambox.utils.`**`noctave_center_freq`**(*lowf*, *highf*, *width=3*)

Calculate exact center N-octave space center frequencies.

In practive, what is often desired is the "simplified" center frequencies, so this function is not of much use.

**Parameters** **lowf** : float

Lowest frequency.

**highf** : float

Highest frequency

**width** : float

Number of filters per octave. (Default value = 3)

**Returns** ndarray

List of center frequencies.

`pambox.utils.`**`psy_fn`**(*x*, *mu=0.0*, *sigma=1.0*)

Calculates a psychometric function with a given mean and variance.

**Parameters** **x** : array_like

"x" values of the psychometric functions.

**mu** : float, optional

Value at which the psychometric function reaches 50%, i.e. the mean of the distribution. (Default value = 0)

**sigma** : float, optional

Variance of the psychometric function. (Default value = 1)

**Returns** **pc** : ndarray

Array of "percent correct", between 0 and 100.

`pambox.utils.`**`read_wav_as_float`**`(`*path*`)`

Reads a wavefile as a float.

**Parameters** **path** : string

Path to the wave file.

**Returns** **wav** : ndarray

`pambox.utils.`**`rms`**`(`*x*, *ac=False*, *axis=-1*`)`

Calculates the RMS value of a signal.

**Parameters** **x** : array_like

Signal.

**ac** : bool

Consider only the AC component of the signal. (Default value = False)

**axis :**

Axis on which to calculate the RMS value. The default is to calculate the RMS on the last dimensions, i.e. axis = -1.

**Returns** ndarray

RMS value of the signal.

`pambox.utils.`**`setdbspl`**`(`*x*, *lvl*, *ac=False*, *offset=0.0*`)`

Sets the level of signal in dB SPL, along its last dimension.

**Parameters** **x** : array_like

Signal.

**lvl** : float

Level, in dB SPL, at which to set the signal.

**ac** : bool

Calculate the AC RMS power of the signal by default (*ac=True*), e.g. the mean is removed. If *False*, considers the non-RMS power. (Default value = False)

**offset** : float

Level, in dB SPL, corresponding to an RMS of 1. By default, an RMS of 1 corresponds to 0 dB SPL, i.e. the default is 0.

**Returns** ndarray

Signal of the same dimension as the original.

`pambox.utils.`**`write_wav`**`(`*fname*, *fs*, *x*, *normalize=False*`)`

Writes floating point numpy array to 16 bit wavfile.

Convenience wrapper around the scipy.io.wavfile.write function.

The '.wav' extension is added to the file if it is not part of the filename string.

Inputs of type *np.float* are converted to *int16* before writing to file.

**Parameters** **fname** : string

Filename with path.

**fs** : int

Sampling frequency.

**x** : array_like

Signal with the shape N_channels x Length

**normalize** : bool

Scale the signal such that its maximum value is one.

**Returns** None

CHAPTER 5

# Indices and tables

- genindex
- modindex
- search

# Bibliography

[wan2014] Wan, R., Durlach, N. I., and Colburn, H. S. (2014). "Application of a short-time version of the Equalization-Cancellation model to speech intelligibility experiments with speech maskers", The Journal of the Acoustical Society of America, 136(2), 768–776

[durlach1963] Durlach, N. I. (1963). "Equalization and Cancellation Theory of Binaural Masking-Level Differences", J. Acoust. Soc. Am., 35(), 1206–1218

[ewert2000] S. D. Ewert and T. Dau: Characterizing frequency selectivity for envelope fluctuations.. J. Acoust. Soc. Am. 108 (2000) 1181–1196.

[jorgensen2011] S. Jørgensen and T. Dau: Predicting speech intelligibility based on the signal-to-noise envelope power ratio after modulation-frequency selective processing. J. Acoust. Soc. Am. 130 (2011) 1475–1487.

[R2] S. Jørgensen and T. Dau: Predicting speech intelligibility based on the signal-to-noise envelope power ratio after modulation- frequency selective processing. J. Acoust. Soc. Am. 130 (2011) 1475–1487.

[jorgensen2013multi] S. Joergensen, S. D. Ewert, and T. Dau: A multi-resolution envelope-power based model for speech intelligibility. J Acoust Soc Am 134 (2013) 436–446.

[chabot-leclerc2016]

[ansi1997sii] American National Standards Institute: American National Standard methods for calculation of the Speech Intelligibility Index (1997).

[R1] A. Westermann and J. M. Buchholz: Release from masking through spatial separation in distance in hearing impaired listeners. Proceedings of Meetings on Acoustics 19 (2013) 050156.

[R3] Auditory Modeling Toolbox, Peter L. Soendergaard B. C. J. Moore. An Introduction to the Psychology of Hearing. Academic Press, 5th edition, 2003.

[R4] Wikipedia, "Analytic signal". http://en.wikipedia.org/wiki/Analytic_signal

# Python Module Index

p

pambox.audio, 9
pambox.central, 11
pambox.distort, 29
pambox.inner, 10
pambox.middle, 11
pambox.outer, 11
pambox.speech, 19
pambox.utils, 33

A

add_signals() (in module pambox.utils), 33
adjust_levels() (pambox.speech.Experiment method), 27
append_results() (pambox.speech.Experiment method), 27
apply() (pambox.distort.WestermannCrm method), 30
apply_jitter() (pambox.central.EC static method), 12
average_level() (pambox.speech.Material method), 24

B

BsEPSM (class in pambox.speech), 22

C

cancel() (pambox.central.EC method), 12
create_filtered_ssn() (pambox.speech.Material method), 24
create_jitter() (pambox.central.EC method), 13
create_ssn() (pambox.speech.Material method), 25

D

dbspl() (in module pambox.utils), 33

E

EC (class in pambox.central), 12
EPSMModulationFilterbank (class in pambox.central), 14
equalize() (pambox.central.EC method), 13
erb_bandwidth() (in module pambox.inner), 10
Experiment (class in pambox.speech), 26

F

fftfilt() (in module pambox.utils), 34
files_list() (pambox.speech.Material method), 25
filter() (pambox.central.EPSMModulationFilterbank method), 14
fit_obs() (pambox.central.IdealObs method), 15

G

get_params() (pambox.central.IdealObs method), 15

H

hilbert() (in module pambox.utils), 35
hilbert_envelope() (in module pambox.inner), 10

I

IdealObs (class in pambox.central), 14
impz() (in module pambox.utils), 36
int2srt() (in module pambox.utils), 36

J

jitter() (pambox.central.EC method), 13

L

load_file() (pambox.speech.Material method), 25
load_files() (pambox.speech.Material method), 25
lowpass_env_filtering() (in module pambox.inner), 10

M

make_same_length() (in module pambox.utils), 36
Material (class in pambox.speech), 24
mfreqz() (in module pambox.utils), 37
mix_noise() (in module pambox.distort), 30
MrSepsm (class in pambox.speech), 21

N

next_pow_2() (in module pambox.utils), 37
noctave_center_freq() (in module pambox.utils), 37
noise_from_signal() (in module pambox.distort), 31

O

overlap_and_add() (in module pambox.distort), 31

P

pambox.audio (module), 9
pambox.central (module), 11
pambox.distort (module), 29
pambox.inner (module), 10
pambox.middle (module), 11
pambox.outer (module), 11

pambox.speech (module), 19
pambox.utils (module), 33
phase_jitter() (in module pambox.distort), 31
pick_section() (pambox.speech.Material static method), 25
play() (in module pambox.audio), 9
plot_bands_above_thres() (pambox.speech.Sepsm method), 19
plot_exc_ptns() (pambox.speech.Sepsm method), 19
plot_filtered_envs() (pambox.speech.Sepsm method), 20
plot_mr_exc_ptns() (pambox.speech.MrSepsm method), 21
plot_snr_env_matrix() (pambox.speech.Sepsm method), 20
plot_tau_hist() (pambox.speech.BsEPSM method), 23
pred_to_pc() (pambox.speech.Experiment class method), 27
predict() (pambox.speech.BsEPSM method), 23
predict() (pambox.speech.MrSepsm method), 22
predict() (pambox.speech.Sepsm method), 20
predict_spec() (pambox.speech.SII method), 24
prediction() (pambox.speech.Experiment static method), 28
preprocessing() (pambox.speech.Experiment method), 28
psy_fn() (in module pambox.utils), 37

## R

read_wav_as_float() (in module pambox.utils), 37
reverb() (in module pambox.distort), 31
rms() (in module pambox.utils), 38
run() (pambox.speech.Experiment method), 28

## S

Sepsm (class in pambox.speech), 19
set_level() (pambox.speech.Material method), 26
setdbspl() (in module pambox.utils), 38
SII (class in pambox.speech), 23
spec_sub() (in module pambox.distort), 32
srts_from_df() (pambox.speech.Experiment class method), 29
ssn() (pambox.speech.Material method), 26

## T

transform() (pambox.central.IdealObs method), 16

## W

WestermannCrm (class in pambox.distort), 29
write_wav() (in module pambox.utils), 38