slab

*Release 1.0.1*

Marc Schoenwiesner, Ole Bialas

Jul 03, 2021
## CONTENTS

1 Why slab? .............................................. 3

2 Installation .......................................... 5

3 Citing slab ........................................... 7
   3.1 Introduction ...................................... 7
   3.2 Sound ............................................. 8
   3.3 Psychoacoustics ................................... 16
   3.4 Filters ............................................. 26
   3.5 HRTFs .............................................. 32
   3.6 Worked examples .................................. 37
   3.7 Reference documentation .......................... 38

Index ................................................... 79
Slab (‘es-lab’, or sound laboratory) is an open source project and Python package that makes working with sounds and running psychoacoustic experiments simple, efficient, and fun! For instance, it takes just eight lines of code to run a pure tone audiogram using an adaptive staircase: *Audiogram*
WHY SLAB?

The package aims to lower the entrance barrier for working with sounds in Python and provide easy access to typical operations in psychoacoustics, specifically for students and researchers in the life sciences. The typical BSc or MSc student entering our lab has limited programming and signal processing training and is unable to implement a psychoacoustic experiment from scratch within the time limit of a BSc or MSc thesis. Slab solves this issue by providing easy-to-use building blocks for such experiments. The implementation is well documented and sufficiently simple for curious students to understand. All functions provide sensible defaults and will in many cases ‘just work’ without arguments (vowel = slab.Sound.vowel() gives you a 1-second synthetic vowel ‘a’, vowel.spectrogram() plots the spectrogram). This turned out to be useful for teaching and demonstrations. Many students in our lab have now used the package to implement their final projects and exit the lab as proficient Python programmers.
CHAPTER TWO

INSTALLATION

Install the current stable release from the python package index with pip:

```
pip install slab
```

or get the latest development version directly from GitHub (if you have git) by running:

```
pip git+https://github.com/DrMarc/slab.git
```

The current version of slab is 1.0.1.

The releases use semantic versioning: major.minor.patch, where major increments for changes that break backwards compatibility, minor increments of added functionality, and patch increases for internal bug fixes. `slab.__version__` prints the installed version.

To run the tests:

```
pip install slab[testing]
```

Then go to the installation directory and run:

```
pytest
```

On Linux, you may need to install libsndfile (required by SoundFile) using your distribution’s package manager, for instance:

```
sudo apt-get install libsndfile1
```

On Windows, you may need to install windows-curses (required for getting button presses in the psychoacoustics classes):

```
pip install windows-curses
```

Working with head related transfer functions requires the h5netcdf module (trying to load a hrf file will raise an error and tell you to install:

```
pip install h5netcdf
```

All other dependencies should have been automatically installed, and you should see meaningful errors if that did not happen for some reason. The dependencies are: numpy, scipy.signal (for filtering and several other DSP functions), matplotlib (for all plotting), SoundFile (for reading and writing wav files), curses or windows-curses (for getting key presses), and SoundCard (for playing and recording sounds). We have seen a hard-to-replicate problem on some Macs with the SoundCard module: a pause of several seconds after a sound is played. If you experience this issue, just uninstall SoundCard:
pip uninstall SoundCard

Slab will then use another method to play sounds (winsound on Windows, afplay on Macs, and SoX on Linux), and will record sounds from the microphone using SoX. There are many other packages to play sounds, depending on our operating system. If you prefer a different one, you can easily modify or replace the `play()` method.
3.1 Introduction

3.1.1 Overview

In this documentation we do not aim at providing a comprehensive explanation of every single slab function (a complete description can be found in the Reference documentation section). Rather, we want to provide some guidance for you to start generating sounds and running experiments.

For starters, you should have a look at the Sound section. There, you will learn how to generate, manipulate and write/read Sounds in slab. Next, you should see the Psychoacoustics section which is about generating trial sequences and running experiments. With these tools you can already do plenty of things! For example...

The Filters section contains some more advanced, but powerful, methods for processing digital signals. The HRTFs section describes the handling of head related transfer functions and will only be relevant if you are interested in spatial audio.

3.1.2 Frequently Asked Questions

- Where can I learn enough Python to use this module?

You can find many free courses online. We usually point our students to Google’s Python class. For those of you who prefer video, Coursera has two suitable courses: Python for Everybody and An Introduction to Interactive Programming with Python. There are also courses specifically for sound and signal processing, for instance this one.

- Which Python environment do you use in the lab?
We recommend **miniconda**, which bundles Python and the conda package manager and installs quickly. You can then install only the packages that you need for your work, like IPython, numpy, scipy, and matplotlib, with a single command:

```bash
conda install ipython numpy scipy matplotlib
```

When programming we use the command line with IPython and the Atom text editor with a package for syntax highlighting. Some lab members use **PyCharm** or **Spyder** as integrated development environments. We don’t recommend IDEs for beginners, because in our experience, students tend to conflate the IDE with Python itself and develop programming habits that they need to unlearn when they want to get productive.

- **I get import errors when using certain functions!**

Slab requires additional modules for some functionality. These modules are not installed automatically because not everyone may need them (such as HRTF file reading) or the installation is OS-dependent (such as SoundFile and curses). Please see **Installation** for how and what to install should you need it. The import error messages will in most cases give you the necessary installation command for Mac/Linux systems.

- **I have set the level of a sound to 70 dB but it is way louder, why?**

This is because slab does not know the hardware you are using to play sound. For example, white noise is generated so that the maximum value in the time series is +1 and the minimum minus one (“full scale”). The RMS of this signal, expressed in deciBels happens to be about 82 dB, but you need to calibrate your system (see *Calibrating the output*) so that the calculated intensity is meaningful. Relative intensities are correct without calibration—so decreasing the intensity by 10 dB (`sound.level -= 10`) will work as expected.

- **What is the difference between white noise and pink noise?**

White noise is a signal that consists of random numbers. This signal has equal power at all frequencies. However, our auditory system does not perceive it that way, which is why white noise appears high-pitched. In the pink noise signal, the power decreases with frequency to correct for this effect. Pink noise is thus a more appropriate choice for a masking or background noise, because it has the same power in each octave. However, there are even better options. The `erb_noise()` method constructs a noise with equal energy not in octaves, but in fractions of approximated auditory filters widths (equivalent rectangular bandwidths, ERB). Or the `multitone_masker()`, which is a noise-like combination of many pure tones at ERB intervals. This noise does not have random amplitude variations and masks evenly across frequency and time.

- **I think I found a bug!**

Please see the **bug reports** section in the contribution guidelines.

- **How can I contribute to the project?**

Please see the **pull request** section in the contribution guidelines if you want to contribute code or useful examples for the documentation.

### 3.2 Sound

#### 3.2.1 Generating sounds

The **Sound** class provides methods for generating, manipulating, displaying, and analysing sound stimuli. You can generate typical experimental stimuli with this class, including tones, noises, and click trains, and also more specialized stimuli, like equally-masking noises, Schroeder-phase harmonics, iterated ripple noise and synthetic vowels. Slab methods assume sensible defaults where possible. You can call most methods without arguments to get an impression of what they do (i.e. `slab.Sound.tone()` returns a 1s-long 1kHz tone at 70 dB sampled at 8 kHz) and then customise from there. For instance, let’s make a 500 ms long 500 Hz pure tone signal with a band-limited (one octave below and above the tone) pink noise background with a 10 dB signal-to-noise ratio:
tone = slab.Sound.tone(frequency=500, duration=0.5)
tone.level = 80 # setting the intensity to 80 dB
noise = slab.Sound.pinknoise(duration=0.5)
noise.filter(frequency=(250, 1000), kind='bp') # bandpass .25 to 1 kHz
noise.level = 70 # 10 dB lower than the tone
stimulus = tone + noise # combine the two signals
stimulus = stimulus.ramp() # apply on- and offset ramps to avoid clicks
stimulus.play()

Sound objects have many useful methods for manipulating (like ramp(), filter(), and pulse()) or inspecting them (like waveform(), spectrum(), and spectral_feature()). A complete list is in the Reference documentation section, and the majority is also discussed here. If you use IPython, you can tap the tab key after typing slab.Sound., or the name of any Sound object followed by a full stop, to get an interactive list of the possibilities.

Sounds can also be created by recording them with slab.Sound.record(). For instance recording = slab.Sound.record(duration=1.0, samplerate=44100) will record a 1-second sound at 44100 Hz from the default audio input (usually the microphone). The record method uses SoundCard if installed, or SoX (via a temporary file) otherwise. Both are cross-platform and easy to install. If neither tool is installed, you won’t be able to record sounds.

### 3.2.2 Specifying durations

Sometimes it is useful to specify the duration of a stimulus in samples rather than seconds. All methods that generate sounds have a duration argument that accepts floating point numbers or integers. Floating point numbers are interpreted as durations in seconds (slab.Sound.tone(duration=1.0) results in a 1 second tone). Integers are interpreted as number of samples (slab.Sound.tone(duration=1000) gives you 1000 samples of a tone).

### 3.2.3 Setting the sample rate

We did not specify a sample rate for any of the stimuli in the examples above. When the samplerate argument of a sound-generating method is not specified, the default sample rate (8 kHz if not set otherwise) is used. It is possible to set a sample rate separately for each Sound object, but it is usually better to set a suitable default sample rate at the start of your script or Python session using slab.set_default_samplerate(). This rate is kept in the class variable _default_samplerate and is used whenever you call a sound generating method without specifying a rate. This rate depends on the frequency content of your stimuli and should be at least double the highest frequency of interest. For some speech sounds or narrow bad noises you might get away with 8 kHz; for spatial sounds you may need 48 kHz or more.

### 3.2.4 Specifying levels

Same as for the sample rate, sounds are generated at a default level (70 dB if not set otherwise). The default is kept in the class variable _default_level and you can set set it to a different value using slab.set_default_level(). Level are not specified directly when generating sounds, but rather afterwards by setting the level property:

```python
sig = slab.Sound.pinknoise()
sig.level # return the current level
sig.level = 85 # set a new level
```

Note that the returned level will not be the actual physical playback level, because that depends on the playback hardware (soundcard, amplifiers, headphones, speakers). Calibrate your system if you need to play stimuli at a known level (see Calibrating the output).
3.2.5 Calibrating the output

Analogous to setting the default level at which sounds are generated with slab.set_default_level(). Each sound's level can be set individually by changing its level property. Setting the level property of a stimulus changes the root-mean-square of the waveform and relative changes are correct (reducing the level attribute by 10 dB will reduce the sound output by the same amount), but the absolute intensity is only correct if you calibrate your output. The recommended procedure to set your system volume to maximum, connect the listening hardware (headphone or loudspeaker) and set up a sound level meter. Then call slab.calibrate(). The calibrate() function will play a 1 kHz tone for 5 seconds. Note the recorded intensity on the meter and enter it when requested. The function returns a calibration intensity, i.e. difference between the tone's level attribute and the recorded level. Pass this value to slab.set_calibration_intensity() to to correct the intensities returned by the level property all sounds. The calibration intensity is saved in the class variable _calibration_intensity. It is applied to all level calculations so that a sound's level attribute now roughly corresponds to the actual output intensity in dB SPL—'roughly' because your output hardware may not have a flat frequency transfer function (some frequencies play louder than others). See Filters for methods to equalize transfer functions.

Experiments sometimes require you to play different stimuli at comparable loudness. Loudness is the perception of sound intensity and it is difficult to calculate. You can use the Sound.aweight() method of a sound to filter it so that frequencies are weighted according to the typical human hearing thresholds. This will increase the correspondence between the rms intensity measure returned by the level attribute and the perceived loudness. However, in most cases, controlling relative intensities is sufficient.

To increase the accuracy of the calibration for your experimental stimuli, pass a sound with a similar spectrum to slab.calibrate(). For instance, if your stimuli are wide band pink noises, then you may want to use a pink noise for calibration. The level of the noise should be high, but not cause clipping.

If you do not have a sound level meter, then you can present sounds in dB HL (hearing level). For that, measure the hearing threshold of the listener at the frequency or frequencies that are presented in your experiment and play your stimuli at a set level above that threshold. You can measure the hearing threshold at one frequency (or for any broadband sound) with the few lines of code (see Audiogram).

3.2.6 Saving and loading sounds

You can save sounds to wav files by calling the object's Sound.write() method (signal.write('signal.wav')). By default, sounds are normalized to have a maximal amplitude of 1 to avoid clipping when writing the file. You should set signal.level to the intended level when loading a sound from file or disable normalization if you know what you are doing. You can load a wav file by initializing a Sound object with the filename: signal = slab.Sound('signal.wav').

3.2.7 Combining sounds

Several functions allow you to string stimuli together. For instance, in a forward masking experiment we need a masking noise followed by a target sound after a brief silent interval. An example implementation of a complete experiment is discussed in the Psychoacoustics section, but here, we will construct the stimulus:

```
masker = slab.Sound.tone(frequency=550, duration=0.5) # a 0.5s 550 Hz tone
masker.level = 80 # at 80 dB
masker.ramp() # default 10 ms raised cosine ramps
silence = slab.Sound.silence(duration=0.01) # 10 ms silence
signal = slab.Sound.tone(duration=0.05) # using the default 500 Hz
```

(continues on next page)

1 Forward masking occurs when a signal cannot be heard due to a preceding masking sound. Typically, three intervals are presented to the listener, two contain only the masker and one contains the masker followed by the signal. The listener has to identify the interval with the signal. The level of the masker is fixed and the signal level is varied adaptively to obtain the masked threshold.
We can make a classic non-interactive demonstration of forward masking by playing these stimuli with decreasing signal level in a loop, once without the masker, and once with the masker. Count for how many steps you can hear the signal tone:

```python
import time  # we need the sleep function
for level in range(80, 10, -5):  # down from 80 in steps of 5 dB
    signal.level = level
    signal.play()
    time.sleep(0.5)
# now with the masker
for level in range(80, 10, -5):  # down from 80 in steps of 5 dB
    signal.level = level
    stimulus = slab.Sound.sequence(masker, silence, signal)
    stimulus.play()
    time.sleep(0.5)
```

Many listeners can hear all of the steps without the masker, but only the first 6 or 7 steps with the masker. This depends on the intensity at which you play the demo (see Calibrating the output below). The `sequence()` method is an example of list unpacking—you can provide any number of sounds to be concatenated. If you have a list of sounds, call the method like so: `slab.Sound.sequence(*[list_of_sound_objects])` to unpack the list into function arguments.

Another method to put sounds together is `crossfade()`, which applies a crossfading between two sounds with a specified overlap in seconds. An interesting experimental use is in adaptation designs, in which one longer stimulus is played to adapt neuronal responses to its sound features, and then a new stimulus feature is introduced (but nothing else changes). Responses (measured for instance with EEG) at that point will be mostly due to that feature. A classical example is the pitch onset response, which is evoked when the temporal fine structure of a continuous noise is regularized to produce a pitch percept without altering the sound spectrum (see Krumbholz et al. (2003)). It is easy to generate the main stimulus of that study, a noise transitioning to an iterates ripple noise after two seconds, with 5 ms crossfade overlap, then filtered between 0.8 and 3.2 kHz:

```python
slab.set_default_samplerate(16000)  # we need a higher sample rate
slab.set_default_level(80)  # set the level for all sounds to 80 dB
adapter = slab.Sound.whitenoise(duration=2.0)
irn = slab.Sound.irn(frequency=125, n_iter=2, duration=1.0)  # pitched sound
stimulus = slab.Sound.crossfade(adapter, irn, overlap=0.005)  # crossfade
stimulus.filter(frequency=[800, 3200], kind='bp')  # filter
stimulus.ramp(duration=0.005)  # 5 ms on- and offset ramps
stimulus.spectrogram()  # note that there is no change at the transition
stimulus.play()  # but you can hear the onset of the regularity (pitch)
```
3.2.8 Plotting and analysis

You can inspect sounds by plotting the `waveform()`, `spectrum()`, or `spectrogram()`:

```python
from matplotlib import pyplot as plt
a = slab.Sound.vowel(vowel='a')
e = slab.Sound.vowel(vowel='e')
i = slab.Sound.vowel(vowel='i')
signal = slab.Sound.sequence(a,e,i)
import matplotlib.pyplot as plt
# preparing a 2-by-2 figure
_, [[ax1, ax2], [ax3, ax4]] = plt.subplots(nrows=2, ncols=2, constrained_layout=True)
signal.waveform(axis=ax1, show=False)
signal.waveform(end=0.05, axis=ax2, show=False) # first 50ms
signal.spectrogram(upper_frequency=5000, axis=ax3, show=False)
signal.spectrum(axis=ax4)
```

Instead of plotting, `spectrum()` and `spectrogram()` will return the time frequency bins and spectral power values for further analysis if you set the `show` argument to `False`. All plotting functions can draw into an existing `matplotlib.pyplot` axis supplied with the `axis` argument.

You can also extract common features from sounds, such as the `crest_factor()` (a measure of how ‘peaky’ the waveform is), or the average `onset_slope()` (a measure of how fast the on-ramps in the sound are—important for sound localization). Features of the spectral content are bundled in the `spectral_feature()` method. It can compute
spectral centroid, flux, flatness, and rolloff, either for an entire sound (suitable for stationary sounds), or for successive
time windows (frames, suitable for time-varying sounds). * The centroid is a measure of the center of mass of a
spectrum (i.e. the ‘center’ frequency). * The flux measures how quickly the power spectrum is changing by comparing
the power spectrum for one frame against the power spectrum from the previous frame; flatness measures how tone-like
a sound is, as opposed to being noise-like, and is calculated by dividing the geometric mean of the power spectrum
by the arithmetic mean (see Dubnov (2004)). * The rolloff measures the frequency at which the spectrum rolls off,
typically used to find a suitable low-cutoff frequency that retains most of the sound power. These particular features
are integrated in slab because we find them useful in our daily work. Many more features are available in packages
specialised on audio processing, for instance librosa. librosa interfaces easily with slab, you can just hand the sample
data and the sample rate of an slab object separately to most of its methods:

```python
import librosa
sig = slab.Sound('music.wav') # load wav file into slab.Sound object
librosa.beat.beat_track(y=sig.data, sr=sig.samplerate)
```

When working with environmental sounds or other recorded stimuli, one often needs to compute relevant features
for collections of recordings in different experimental conditions. The slab module contains a function slab.
apply_to_path(), which applies a function to all sound files in a given folder and returns a dictionary of file names
and computed features. In fact, you can also use that function to modify (for instance ramp and filter) all files in a
folder.

For other time-frequency processing, the frames() provides an easy way to step through the signal in short windowed
frames and compute some values from it. For instance, you could detect on- and offsets in the signal by computing the
crest factor in each frame:

```python
from matplotlib import pyplot as plt
signal.pulse()  # apply a 4 Hz pulse to the 3 vowels from above
signal.waveform()  # note the pulses
crest = []  # the short-term crest factor will show on- and offsets
frames = signal.frames(duration=64)
for f in frames:
    crest.append(f.crest_factor())
times = signal.frametimes(duration=64)  # frame center times
import matplotlib.pyplot as plt
plt.plot(times, crest)  # peaks in the crest factor mark intensity ramps
```

### 3.2.9 Binaural sounds

For experiments in spatial hearing, or any other situation that requires differential manipulation of the left and right
channel of a sound, you can use the Binaural class. It inherits all methods from Sound and provides additional meth-
ods for generating and manipulating binaural sounds, including advanced interaural time and intensity manipulation.

#### Generating binaural sounds

Binaural sounds support all sound generating functions with a n_channels attribute of the Sound class, but automati-
cally set n_channels to 2. Noises support an additional kind argument, which can be set to ‘diotic’ (identical noise
in both channels) or ‘dichotic’ (uncorrelated noise). Other methods just return 2-channel versions of the stimuli. You
can recast any Sound object as Binaural sound, which duplicates the first channel if n_channels is 1 or greater than
2:

```python
monaural = slab.Sound.tone()
monaural.n_channels
```
Loading a wav file with `slab.Binaural('file.wav')` returns a Binaural sound object with two channels (even if the wav file contains only one channel).

### Manipulating ITD and ILD

The easiest manipulation of a binaural parameter may be to change the interaural level difference (ILD). This can be achieved by setting the `level` attributes of both channels:

```
noise = slab.Binaural.pinknoise()
noise.left.level = 75
noise.right.level = 85
noise.level
```

The `ild()` makes this easier and keeps the overall level constant: `noise.ild(10)` amplifies the right channel by 5 dB and attenuates the left channel by the same amount to achieve a 10dB level difference. Positive dB values move the virtual sound source to the right and negative values move the source to the left. The pink noise in the example is a broadband signal, and the ILD is frequency dependent and should not be the same for all frequencies. A frequency-dependent level difference can be computed and applied with `interaural_level_spectrum()`. The level spectrum is computed from a head-related transfer function (HRTF) and can be customised for individual listeners. See HRTFs for how to handle these functions. The default level spectrum is computed from the HRTF of the KEMAR binaural recording mannequin (as measured by Gardner and Martin (1994) at the MIT Media Lab). The level spectrum takes a while to compute and it may be useful to save it. It is a Python dict containing the level differences in a numpy array along with a frequency vector, an azimuth vector, and the sample rate. You can save it for instance with pickle:

```
import pickle
ils = slab.Binaural.make_interaural_level_spectrum()
pickle.dump(ils, open('ils.pickle', 'wb'))  # save using pickle
ils = pickle.load(open('ils.pickle', 'rb'))  # load pickle
```

If the limitations of pickle worry you, you can use numpy.save with a small caveat when loading: numpy.save wraps the dict in an object and we need to remove that after loading with the somewhat strange index `[]`:

```
import numpy
numpy.save('ils.npy', ils)  # save using numpy
ils = numpy.load('ils.npy', allow_pickle=True)[()]  # load and get the original dict from the wrapping object
```

If you are unsure which ILD value is appropriate, `azimuth_to_ild()` can compute ILDs corresponding to an azimuth angle, for instance 45 degrees, and a frequency:

```
slab.Binaural.azimuth_to_ild(45)  # -9.12  # correct ILD in dB
noise.ild(-9.12)  # apply the ILD
```
A dynamic ILD, which evokes the perception of a moving sound source, can be applied with `ild_ramp()`. The ramp is linear from and to a given ILD.

Similar functions exist to manipulate interaural time differences (ITD): `itd()`, `azimuth_to_ild()` (using a given head radius), and `itd_ramp()`. To present a signal from a given azimuth using both cues, use the `at_azimuth()`, which calculates the correct ILD and ITD for you and applies it.

ITD and ILD manipulation leads to the percept of *lateralization*, that is, a source somewhere between the ears inside the head. Additional spectral shaping is necessary to generate an externalized percept (outside the head). This shaping can be achieved with the `externalize()`, which applies a low-resolution HRTF filter (KEMAR by default). Using both ramp functions and externalization, it is easy to generate a convincing sound source movement with pulsed pink noise:

```python
noise = slab.Binaural.pinknoise(samplerate=44100)
from_ild = slab.Binaural.azimuth_to_ild(-90)
from_itd = slab.Binaural.azimuth_to_itd(-90)
to_ild = slab.Binaural.azimuth_to_ild(90)
to_itd = slab.Binaural.azimuth_to_itd(90)
note_moving = noise.ild_ramp(from_ild, to_ild)
note_moving = note_moving.itd_ramp(from_itd, to_itd)
note_moving.externalize() # apply filter in place
note_moving.play() # best through headphones
```

### 3.2.10 Signals

Sounds inherit from the `Signal` class, which provides a generic signal object with properties duration, number of samples, sample times, number of channels. The actual samples are kept as numpy array in the `data` property and can be accessed, if necessary as for instance `signal.data`. Signals support slicing, arithmetic operations, and conversion between sample points and time points directly, without having to access the `data` property. The methods `resample()`, `envelope()`, and `delay()` are also implemented in Signal and passed to the child classes Sound, Binaural, and Filter. You do not normally need to use the Signal class directly.

```python
sig = slab.Sound.pinknoise(n_channels=3)
sig.duration
out: 1.0
sig.n_samples
out: 8000
sig.data.shape # accessing the sample array
out: (8000, 3) # which has shape (n_samples x n_channels)
sig2 = sig.resample(samplerate=4000) # resample to 4 kHz
env = sig2.envelope() # returns a new signal containing the lowpass Hilbert envelopes of both channels
sig.delay(duration=0.0006, channel=0) # delay the first channel by 0.6 ms
```
3.3 Psychoacoustics

The Psychoacoustics class simplifies psychoacoustic experiments by providing classes and methods for trial sequences and adaptive staircases, results and configuration files, response collection via keyboard and button boxes, and handling of collections of precomputed stimuli. This all-in-one approach makes for clean code and easy data management.

3.3.1 Trial sequences

Experiments are often defined by a sequence of trials of different conditions. This sequence is generated before the experiment according to certain rules. In the most basic case, a set of experimental conditions are repeated a number of times pseudorandom order. Such experiments can be handled by the Trialsequence class. To generate an instance of Trialsequence you define a list of conditions and specify how often each of them is repeated (n_reps). You can also specify the kind of list you want to generate: “non_repeating” means that the same condition will not appear twice in a row, “random_permutation” means that the order is completely randomised. For example, generate pure tones with different frequencies and play them in non-repeating, randomised order:

```python
freqs = [495, 498, 501, 504]  # frequencies of the tones
seq = slab.Trialsequence(conditions=freqs, n_reps=10)  # 10 repetitions per condition
# now we draw elements from the list, generate a tone and play it until we reach the end:
for freq in seq:
    stimulus = slab.Sound.tone(frequency=freq)
    stimulus.play()
```

Usually, we do not only want to play sounds to the participants in our experiment. Instead, we want them to perform some kind of task and give a response. In the example above we could, for instance, ask after every tone if that tone was higher or lower in frequency than the previous one. The response is captured with the key() context manager which can record single button presses (using either the curses module or the key_press_event() of the stairs plot, see _responses). In our example, we instruct the subject to press “y” (yes) if the played tone was higher then the previous and “n” (no) if it was lower (a 1-back task). After each trial we check if the response was correct and store that information as 1 (correct) or 0 (wrong) in the trial sequence:

```python
for freq in seq:
    stimulus = slab.Sound.tone(frequency=freq)
    stimulus.play()
    if seq.this_n > 0:  # don't get response for first trial
        previous = seq.get_future_trial(-1)
    with slab.key() as key:
        response = key.getch()
        # check if the response was correct, if so store a 1, else store 0
        if (freq > previous and response == ord('y')) or (freq<previous and response == ord('n')):
            seq.add_response(1)
        else:
            seq.add_response(0)
seq.save_json("sequence.json")  # save the trial sequence and response
```

There are two ways for a response to be correct in this experiment. Either the frequency of the stimulus was higher than the last one and the ‘y’ key was pressed, or it was lower and the ‘n’ key was pressed. (ord() is used to get the key codes of the ‘y’ and ‘n’ keys (112 and 110, respectively). All other options, including missed responses, are counted as wrong answers. Since we encoded correct responses as 1 and wrong responses as 0, we could just sum over the list of responses and divide by the length of the list to get the fraction of trials that was answered correctly.
Kinds of trial sequences

Trial sequences are useful for non-adaptive testing (the current stimulus does not depend on the listeners previous responses) and other situations where you need a controlled sequence of stimulus values. The TrialSequence class constructs several controlled sequences (random permutation, non-repeating, infinite, oddball), computes transition probabilities and condition frequencies, and can keep track of responses:

```python
# sequence of 5 conditions, repeated twice, without direct repetitions:
seq = slab.Trialsequence(conditions=5, n_reps=2)

# infinite sequence of color names:
seq = slab.Trialsequence(conditions=['red', 'green', 'blue'], kind='infinite')

# stimulus sequence for an oddball design:
seq = slab.Trialsequence(conditions=1, deviant_freq=0.12, n_reps=60)
```

The list of trials is contained in the trials of the TrialSequence object, but you don’t normally need to access this list directly. A TrialSequence object can be used like a Staircase object in a listening experiment and will return the current stimulus value when used in a loop. Below is the detection threshold task from the Staircase, rewritten using Fechner’s method of constant stimuli with a TrialSequence:

```python
stimulus = slab.Sound.tone(duration=0.5)
levels = list(range(0, 50, 10))  # the sound levels to test
trials = slab.Trialsequence(conditions=levels, n_reps=10)  # each repeated 10 times
for level in trials:
    stimulus.level = level
    stimulus.play()
    with slab.key() as key:
        response = key.getch()
    trials.add_response(response)
trials.response_summary()
```

Because there is no simple threshold, the TrialSequence class provides a response_summary(), which tabulates responses by condition index in a nested list.

The infinite kind of TrialSequence is perhaps less suitable for controlling the stimulus parameter of interest, but it is very useful for varying other stimulus attributes in a controlled fashion from trial to trial (think of ‘roving’ paradigms). Unlike when selecting a random value in each trial, the infinite TrialSequence guarantees locally equal value frequencies, avoids direct repetition, and keeps a record in case you want to include the sequence as nuisance covariate in the analysis later on. Here is a real-world example from an experiment with pseudo-words, in which several words without direct repetition were needed in each trial. word_list contained the words as strings, later used to load the correct stimulus file:

```python
word_seq = slab.Trialsequence(conditions=word_list, kind='infinite')
word = next(word_seq)  # draw a word from the list
```

This is one of the very few cases where it makes sense to get the next trial by calling Python’s next() function, because this is not the main trial sequence. The main trial sequence (the one determining the values of your main experimental parameter) should normally be used in a for loop as in the previous example.
Controlling transitions

While randomized sequences do the job most of the time, in some cases it is necessary to control the transitions between the individual conditions more tightly. For instance, you may want to ensure nearly equal transitions, or avoid certain combinations of subsequent conditions entirely. The `transitions()` method counts, for each condition, how often every other condition follows this one. You can divide the count by the number of repetitions in the sequence to get the transitional probabilities:

```python
trials = slab.Trialsequence(conditions=4, n_reps=10)
trials.transitions()
```

```
out:
array([[0., 2., 6., 2.],
       [3., 0., 0., 7.],
       [2., 6., 0., 1.],
       [4., 2., 4., 0.]]
)
```

```
trials.transitions() / 10  # divide by n_reps to get the probability
```

```
out:
array([[0. , 0.2, 0.6, 0.2],
       [0.3, 0. , 0. , 0.7],
       [0.2, 0.6, 0. , 0.1],
       [0.4, 0.2, 0.4, 0. ]])
```

The diagonal of this array contains only zeroes, because a condition cannot follow itself in the default non_repeating trial sequence. The other entries are uneven; for instance, condition 1 is followed by condition 3 seven times, but never by condition 2. If you want near-equal transitions, then you could generate sequences in a loop until a set condition is fulfilled, for instance, no transition > 4:

```python
import numpy
trans = 5
while numpy.any(trans>4):
    trials = slab.Trialsequence(conditions=4, n_reps=10)
    trans = trials.transitions()
print(trans)
```

```
out:
array([[0., 3., 3., 3.],
       [4., 0., 3., 3.],
       [3., 4., 0., 3.],
       [3., 3., 4., 0. ]])
```

If your condition is more complicated, you can perform several tests in the loop body and set a flag that determines when all have been satisfied and the loop should be end. But be careful, setting these constraints too tightly may result in an infinite loop.

Alternative Choices

Often, an experimental paradigm requires more complex responses than yes or no. A common option is the classical “forced choice” paradigm, in which the subject has to pick a response from a defined set of responses. Since this is a common paradigm, the `Trialsequence` and `Staircase` class have a method for it called `present_afc_trial()` (afc stands for alternative forced choice). With this function we can make our frequency discrimination task from the example above a bit more elaborate. We define the frequencies of our target tones and add two distractor tones with a frequency of 500 Hz. In each trial, all three tones (target + 2 x distractor) are played in random order. The participant answers the question: “which tone was different from the others?” and responds by pressing the key “1”, “2” or “3”. All of this can be done in only 6 lines of code:
distractor = slab.Sound.tone(duration=0.5, frequency=500)

freqs = list(range(495, 505))

trials = slab.Trialsequence(conditions=freqs, n_reps=2)

for freq in trials:
    target = slab.Sound.tone(frequency=freq, duration=0.5)
    trials.present_afc_trial(target, [distractor, distractor], isi=0.2)

3.3.2 Adaptive staircases

In many cases, you do not want to test every condition with the same frequency, but adapt the stimulus presentation to
the responses of the participant. For example, when measuring an audiogram, you want to spend most of the testing
time around the threshold to make the testing efficient. The Staircase class lets you do that. You pick an initial
value for the stimulus parameter (start_val) and a step size (step_sizes). With each trial, the starting value is
decreased by one step size until the subject is not able to respond correctly anymore. Then it is increased step wise
until the response is correct again, then decreased again and so on. This procedure is repeated until the given number
of reversals (n_reversals) is reached. The step size can be a list in which case the current step size moves one index
in the list by each reversal until the end of the list is reached. For example, we could use a step size of 4 until we
crossed the threshold for the first time, then use a step size of 1 for the rest of the experiment. This ensures that we get
to the threshold quickly and, once we are there, measure it precisely. (The simulate_response() method used here
is explained under simulating.)

stairs = slab.Staircase(start_val=10, n_reversals=18, step_sizes=[4,1])

for stimulus_value in stairs:
    response = stairs.simulate_response(threshold=3) # simulate subject's response
    stairs.add_response(response) # initiates calculation of next stimulus value
    stairs.plot()

Calling the plot function in the for loop (after Staircase.add_response()) will update the plot each trial and let
you monitor the performance of the participant, including the current stimulus value (grey dot), and correct/incorrect
responses (green and red dots). (On some Windows systems, the plot captures the focus and may prevent you from
entering responses in the terminal window. In that case, switch the slab.psychoacoustics.input_method to
‘figure’. This will get a button press through the stairs figure’s key_press_event().)

An audiogram is a typical example for a staircase procedure. We can define a list of frequencies and run a staircase for
each one. Afterwards we can print out the result using the thresh() method:

from matplotlib import pyplot as plt
freqs = [125, 250, 500, 1000, 2000, 4000]
threshs = []

for frequency in freqs:
    stimulus = slab.Sound.tone(frequency=frequency, duration=0.5)
    stairs = slab.Staircase(start_val=50, n_reversals=18)
    print(f"Starting staircase with \{frequency\} Hz: ")
    for level in stairs:
        stimulus.level = level
        stairs.present_tone_trial(stimulus)
        threshs.append(stairs.threshold())
        print(f"Threshold at \{frequency\} Hz: \{stairs.threshold()\} dB")

plt.plot(freqs, threshs) # would plot the audiogram

present_tone_trial() is a convenience method that presents the trial, acquires a response, and optionally prints
trial information. All of this can be done explicitly, as shown in the Trialsequence example.

3.3. Psychoacoustics
Staircase Parameters

Setting up a near optimal staircase requires some expertise and pilot data. Practical recommendations can be found in García-Pérez (1998). `start_val` sets the stimulus value presented in the first trial and the starting point of the staircase. This stimulus should in general be easy to detect/discriminate for all participants. You can limit the range of stimulus values between `min_val` and `max_val` (the default is infinity in both directions). `step_sizes` determines how far to go up or down when changing the stimulus value adaptively. If it is a list of values, then the first element is used until the first reversal, the second until the second reversal, etc. `step_type` determines what kind of steps are taken: ‘lin’ adds/subtracts the step size from the current stimulus value, ‘db’ and ‘log’ will step by a certain number of decibels or log units. Typically you would start with a large step size to quickly get close to the threshold, and then switch to a smaller step size. Steps going up are multiplied with `step_up_factor` to allow unequal step sizes and weighted up-down procedures (Kaernbach (1991)). Optimal step sizes are a bit smaller than the spread of the psychometric function for the parameter you are testing. You can set the number of correct responses required to reduce the stimulus value with `n_down` and the number of incorrect responses required to increase the value with `n_up`. The default is a 1up-2down procedure. You can also add a number of training trials, in which the stimulus value does not change, with `n_pretrials`.

Simulating responses

For testing and comparing different staircase settings it can be useful to simulate responses. The first staircase example uses `simulate_responses()` to draw responses from a logistic psychometric function with a given threshold and width (expressed as the stimulus range in which the function increases from 20% to 80% hitrate). For instance, if the current stimulus value is at the threshold, then the function returns a hit with 50% probability. This is useful to simulate and compare different staircase settings and determine to which hit rate they converge. For instance, let’s get a feeling for the effect of the length of the measurement (number of reversals required to end the staircase) and the accuracy of the threshold (standard deviation of thresholds across 100 simulated runs). We test from 10 to 40 reversals and run 100 staircases in the inner loop, each time saving the threshold, then computing the interquartile range and plotting it against the number of reversals. Longer measurements should reduce the variability:

```python
from matplotlib import pyplot as plt
stairs_iqr =[]
for reversals in range(10,41,5):
    thres = []
    for _ in range(100):
        stairs = slab.Staircase(start_val=10, n_reversals=reversals)
        for trial in stairs:
            resp = stairs.simulate_response(3)
            stairs.add_response(resp)
            thres.append(stairs.threshold())
        thres.sort()
        stairs_iqr.append(thres[74] - thres[24]) # 75th-25th percentile
plt.plot(range(10,41,5), stairs_iqr)
plt.gca().set(xlabel='reversals', ylabel='threshold IQR')
```

Many other useful simulations are possible. You could check whether a 1up-3down procedure procedure would arrive at a similar accuracy in fewer trials, what the best step size for a given psychometric function is, or how much a wider than expected psychometric function increases experimental time. Simulations are a good starting point, but the psychometric function is a very simplistic model for human behaviour. Check the results with pilot data.

Simulation is also useful for finding the hitrate (or point on the psychometric function) that a staircase converges on in cases that are difficult for calculate. For instance, it is not immediately obvious on what threshold a 1up-4down staircase with `step_up_factor` 1.5 and a 3-alternative forced choice presentation converges on:

3.3. Psychoacoustics
import numpy
threshs = []
width = 2
thresh = 3
for _ in range(100):
    stairs = slab.Staircase(start_val=10, n_reversals=30, n_down=4, step_up_factor=1.5)
    for trial in stairs:
        resp = stairs.simulate_response(threshold=thresh, transition_width=width, intervals=3)
        stairs.add_response(resp)
    threshs.append(stairs.threshold())
# now we have 100 thresholds, take mean and convert to equivalent hitrate:
hitrate = 1 / (1 + numpy.exp(4 * (0.5/width) * (thresh - numpy.mean(threshs))))
print(hitrate)
# 0.83

As you can see, even through the threshold in the response simulation is 3 (that is, the rate of correct responses is > 0.5 above this value; how fast it increases from there depends on the transition_width), the mean threshold returned from the procedure is over 4.5. The last line translates this value in relation to the width of the simulated psychometric function into a hitrate of about 0.83.

### 3.3.3 Acquiring key presses

When you use a staircase in a listening experiment, you need to record responses from the participant, usually in the form of button presses. The `key()` context manager can record single button presses from the computer keyboard (or an attached USB number pad), or via the key press event handler of a matplotlib figure, or from a custom USB buttonbox. The input is selected by setting `slab.psychoacoustics.input_method` to 'keyboard', 'buttonbox', or 'figure'. This allow you to test your code on your laptop and switch to button box input at the lab computer by changing a single line of code. Getting a button press from the keyboard will clear your terminal while waiting for the response, and restore it afterwards. The the lab, you may not want to use a keyboard, which can be distracting. A simple response box with the required number of buttons can be constructed easily with an Arduino-compatible micro-controller that can send key codes to the computer via USB. Check for a press of a button attached to a digital input and send a string corresponding to the key code of the desired key followed by the Enter key. If you use the `plot()` method of the `Staircase` class to show the progress of the test, you can set the `input_method` to 'figure' to get a keypress via the figure’s key press event handler.

The `key()` method uses the key code of a button, rather than the string character it produces when pressed. You can find the code of a key by calling Python’s `ord()` function. For instance, `ord('y')` returns 121, the code of the ‘y’ key.

The `Trialsequence` and `Staircase` classes have two convenience methods to present tones and acquire a response from the listener in one step: `present_tone_trial()` and `present_afc_trial()`. Both take a list of key codes that are considered valid responses (:param:`key_codes`). The list defaults to the number keys from 1 to 9. If you use any of these keys in `present_tone_trial()`, then you just need to specify which of them is counted as a correct response by setting the argument `correct_key_idx` to the list index that contains the correct key (instead of a single index you can specify a list of indices if you want to count several keys as correct). In `present_afc_trial()`, the order of the keys in :param:`key_codes` should correspond to the keys that should be pressed to indicate interval 1, 2, etc. In this case, the correct key is different in each trial, depending on the interval that contains the target stimulus.

Here is an example of how to use the `key()` in a staircase that finds the detection threshold for a 500 Hz tone, after every trial you have to indicate whether you could or could not hear the sound by pressing “y” for yes or any other button for no:
```python
stimulus = slab.Sound.tone(duration=0.5)
stairs = slab.Staircase(start_val=60, step_sizes=[10, 3])
for level in stairs:
    stimulus.level = level
    stimulus.play()
    with slab.key('Press y for yes or n for no.') as key:
        response = key.getch()
        if response == 121:  # 121 is the unicode for the "y" key
            stairs.add_response(True)  # initiates calculation of next stimulus value
        else:
            stairs.add_response(False)
stairs.plot()
stairs.threshold()
```

Note that slab is not optimal for measuring reaction times due to the timing uncertainties in the millisecond range introduced by modern multi-tasking operating systems. If you are serious about reaction times, you should use an external DSP device to ensure accurate timing. Ubiquitous in auditory research are the realtime processors from Tucker-Davies Technologies (our module freefield module works with these devices).

### 3.3.4 Precomputed sounds

If you present white noise in an experiment, you probably do not want to play the exact same noise in each trial (‘frozen’ noise), but different random instances of noise. The `Precomputed` class manages a list of pre-generated stimuli, but behave like a single sound. You can pass a list of sounds, a function to generate sounds together with an indication of how many you want, or a generator expression to initialize the `Precomputed` object. The object has a `play()` method that plays a random stimulus from the list (but never the stimulus played just before), and remembers all previously played stimuli in the sequence. The `Precomputed` object can be saved to a zip file and loaded back later on:

```python
# generate 10 instances of pink noise::
stims = slab.Precomputed(lambda: slab.Sound.pinknoise(), n=10)
stims.play()  # play a random instance
stims.play()  # play another one, guaranteed to be different from the previous one
stims.sequence  # the sequence of instances played so far
stims.write('stims.zip')  # save the sounds as zip file
stims = slab.Precomputed.read('stims.zip')  # reloads the file into a Precomputed object
```

### 3.3.5 Results files

In most experiments, the performance of the listener, experimental settings, the presented stimuli, and other information need to be saved to disk during the experiment. The `ResultsFile` class helps with several typical functions of these files, like generating timestamps, creating the necessary folders, and ensuring that the file is readable if the experiment is interrupted writing to the file after each trial. Information is written incrementally to the file in single lines of JSON (a JSON Lines file).

Set the folder that will hold results files from all participants for the experiment somewhere at the top of your script with the `results_folder`.

```python
subject_ID = 'MS01'
slab.ResultsFile.results_folder = 'MyResults'
file = slab.ResultsFile(subject='MS')
pnint(file.name)
file.write(subject_ID)
```
You can now use the `write()` method to write any information to the file, to be precise, you can write any object that can be converted to JSON, like strings, lists, or dictionaries. Numpy data types need to be converted to python types. A numpy array can be converted to a list before saving by calling its `numpy.ndarray.tolist()` method, and numpy ints or floats need to be converted by calling their `item()` method. You can try out what the JSON representation of an item is by calling:

```python
import json
import numpy
a = 'a string'
b = [1, 2, 3, 4]
c = {'frequency': 500, 'duration': 1.5}
d = numpy.array(b)
for item in [a, b, c]:
    json.dumps(item)
    json.dumps(d.tolist())
```

Trialsequence and Staircase objects can pass their entire current state to the write method, which makes it easy to save all settings and responses from these objects:

```python
trials = slab.Trialsequence(conditions=4, n_reps=10)
file.write(trials, tag='trials')
```

The `write()` method writes a dictionary with a single key-value pair, where the key is supplied as `tag` argument argument (default is a time stamp in the format '%Y-%m-%d-%H-%M-%S'), and the value is the json-serialized data you want to save. The information can be read back from the file, either while the experiment is running and you need to access a previously saved result (`read()`), or for later data analysis (`ResultsFile.read_file()`). Both methods can take a `tag` argument to extract all instances saved under that tag in a list.

### 3.3.6 Configuration files

Another recurring issue when implementing experiments is loading configuration settings from a text file. Experiments sometimes use configuration files when experimenters (who might not be Python programmers) need to set parameters without changing the code. The format is a plain text file with a variable assignment on each line, because it is meant to be written and changed by humans. The function `load_config()` reads the text file and return a `namedtuple()` with the variable names and values. If you have a text file with the following content:

```plaintext
samplerate = 32000
pause_duration = 30
speeds = [60, 120, 180]
```

you can make all variables available to your script as attributes of the named tuple object:

```python
conf = slab.load_config('example.txt')
conf.speeds
% [60, 120, 180]
```
3.4 Filters

The `Filter` class can be used to generate, manipulate and save filter banks and transfer functions. Filters are represented internally as `Signal` and come in two flavours: finite impulse responses (FIR) and frequency bin amplitudes (FFT). The `fir` (True or False).

3.4.1 Simple Filters

Simple low-, high-, bandpass, and bandstop filters can be used to suppress selected frequency bands in a sound. For example, if you don’t want the sound to contain power above 1 kHz, apply a 1 kHz lowpass filter:

```python
from matplotlib import pyplot as plt
sound = slab.Sound.whitenoise()
 filt = slab.Filter.band(frequency=1000, kind='lp')
sound_filt = filt.apply(sound)
_ , [ax1, ax2] = plt.subplots(2, sharex=True)
sound.spectrum(axis=ax1, color="blue")
sound_filt.spectrum(axis=ax2, color="red")
```

The `filter()` of the `Sound` class wraps around `Filter.cutoff_filter()` and `Filter.apply()` so that you can use these filters conveniently from within the `Sound` class.

Filter design is tricky and it is good practice to plot and inspect the transfer function of the filter:

```python
filt.tf()
```

Inspecting the waveform of the sound (using the `waveform()` method) or the rms level (using the `slab.Sound/level` attribute) shows that the amplitude of the filtered signal is smaller than that of the original, because the filter has removed power. You might be tempted to correct this difference by increasing the level of the filtered sound, but this is not recommended because the perception of intensity (loudness) depends non-linearly on the frequency content of the sound.

3.4.2 Filter banks

A `Filter` objects can hold multiple channels, just like a `Sound` object. In the following, we will refer to filters with multiple channels as filter banks. You can create filter banks the same way you create multi-channel sound (since the `Filter` and `Sound` class both inherit from the parent `Signal` class). When you apply a filter bank to a single sound, each filter will be applied to a separate copy of the sound and the `apply()` function will return a sound with a number of channels equal to the number of filters in the bank. This way you can create, for example, a series of sounds with different frequency bands:

```python
filters = []
low_cutoff_freqs = [500, 1000, 1500]
 high_cutoff_freqs = [1000, 1500, 2000]
 for low, high in zip(low_cutoff_freqs, high_cutoff_freqs):
 filters.append(slab.Filter.band(frequency=(low, high), kind='bp'))
 fbank = slab.Filter(filters) # put the list into a single filter object
sound_filt = fbank.apply(sound) # apply each filter to a copy of sound
 # plot the spectra, each color represents one channel of the filtered sound
_, ax = plt.subplots(1)
sound_filt.spectrum(axis=ax, show=False)
```

(continues on next page)
3.4. Filters
The channels, or subbands, of the filtered sound can be modified and re-combine with the `combine_subbands()` method. An example of this process is the vocoder implementation in the `Sound` class, which uses these features of the `Filter` class. The multi-channel filter is generated with `cos_filterbank()`, which produces cosine-shaped filters that divide the sound into small frequency bands which are spaced in a way that mimics the filters of the human auditory periphery (equivalent rectangular bandwidth, ERB). Here is an example of the transfer functions of this filter bank:

```python
fbank = slab.Filter.cos_filterbank()
fbank.tf()
```

A speech signal is filtered with this bank, and the envelopes of the subbands are computed using the `envelope()` method of the `Signal` class. The envelopes are filled with noise, and the subbands are collapsed back into one sound. This removes most spectral information but retains temporal information in a speech signal and sound a bit like whispered speech. Here are the essential bits of code from the `vocode()` method to illustrate the use of a filter bank. The first line records a speech sample from the microphone (say something!):

```python
signal = slab.Sound.record()  # record a 1 s speech sample from the microphone
fbank = slab.Filter.cos_filterbank(length=signal.n_samples)  # make the filter bank
subbands = fbank.apply(signal)  # get a sound channel for each filter channel
envs = subbands.envelope()  # now get the envelope of each frequency band...
```
3.4. Filters

Frequency Response

<table>
<thead>
<tr>
<th>Frequency [Hz]</th>
<th>Amplitude [dB]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-300</td>
</tr>
<tr>
<td>5000</td>
<td>-300</td>
</tr>
<tr>
<td>10000</td>
<td>-300</td>
</tr>
<tr>
<td>15000</td>
<td>-300</td>
</tr>
<tr>
<td>20000</td>
<td>-300</td>
</tr>
</tbody>
</table>
noise = slab.Sound.whitenoise()
subbands_noise = fbank.apply(noise)
subbands_noise *= envs  # ... and fill them with noise
subbands_noise.level = subbands.level # keep subband level of original
vocoded = slab.Filter.collapse_subbands(subbands_noise, filter_bank=fbank)
vocoded.play()

If you want to apply multiple filters to the same sound in sequence without creating subbands, you can simply use a for loop. For example, you could remove different parts of the spectrum using bandstop filters:

sound = slab.Sound.whitenoise()
# create a filter bank which consists of three separate bandstop filters
filter_bank = slab.Filter([slab.Filter.band(kind="bs", frequency=f) for f in [(200, 300), (500, 600), (800, 900)]]
for i in range(filter_bank.n_channels):
    sound = filter_bank.channel(i).apply(sound)

If the a one-channel filter is applied to a multi-channel sound, the filter will be applied to each channel individually. This can be used, for example, to easily pre-process a set of recordings (where every recordings is represented by a channel in the slab.Sound object). If a multi-channel filter is applied to a multi-channel signal with the same number of channels each filter channel is applied to the corresponding signal channel. This mechanism is used, for example, during the equalization of a set of loudspeakers.

### 3.4.3 Equalization

In Psychoacoustic experiments, we are often interested in the effect of a specific feature. One could, for example, take the bandpass filtered sounds from the example above and investigate how well listeners can discriminate them from a noisy background - a typical cocktail-party task. However, if the transfer function of the loudspeakers or headphones used in the experiment is not flat, the findings will be biased. Imagine that the headphones used were bad at transmitting frequencies below 1000 Hz. This would make a sound with center frequency of 550 Hz harder to detect than one with a center frequency of 1550 Hz. To prevent this from happening, we have to equalize the headphones’ transfer function. You can measure the transfer function of your system by playing a wide-band sound, like a chirp, and recording it with a probe microphone (which itself must have a flat transfer function). From this recording, you can calculate the transfer function, which is basically the difference in the power spectrum of the played sound and the recording. We can take the opposite of that difference to create an inverse filter. Apply the inverse filter to a sound before playing it through that system to compensate for the uneven transfer, because the inverse filter and the actual transfer function cancel each other. The equalizing_filterbank() method does most of this work for you. For a demonstration, we simulate a (pretty bad) loudspeaker transfer function by applying a random filter:

```
import random
def f or f in range(10):
    freqs = 400
    gain = random.random() + .4
    tf = slab.Filter.band(frequency=freqs, gain=gain)
    sound = slab.Sound.whitenoise()
    recording = tf.apply(sound)
    recording.spectrum()
```

With the original sound and the simulated recording we can compute an inverse filter und pre-filter the sound (or in this case, just filter the recording) to achieve a nearly flat playback through our simulated bad loudspeaker:
3.4. Filters
inverse = slab.Filter.equalizing_filterbank(reference=sound, sound=recording)
equalized = inverse.apply(recording)
equalized.spectrum()

If there are multiple channels in your recording (assembled from recordings of the same white noise through several loudspeakers, for instance) then the `equalizing_filterbank()` method returns a filter bank with one inverse filter for each signal channel, which you can `apply()` just as in the example above.

### 3.5 HRTFs

A head-related transfer function (HRTF) describes the impact of the listeners ears, head and torso on incoming sound for every position in space. Knowing the listeners HRTF, you can simulate a sound source at any position by filtering it with the transfer function corresponding to that position. The `HRTF` class provides methods for manipulating, plotting, and applying head-related transfer functions.
3.5.1 Reading HRTF data

Typically the HRTF class is instantiated by loading a file. The canonical format for HRTF-data is called sofa (Spatially Oriented Format for Acoustics). To read sofa files, you need to install the h5netcdf module: `pip install h5netcdf`. The module includes a set of standard HRTF recordings from the KEMAR (a mannequin for acoustic recordings). You can get the path to the folder containing the recordings with the `data_path()` function. The first time you call this function, the recordings will be downloaded from the sofa website. You can read them by calling the HRTF class with the name of the file as an argument. Print the resulting object to obtain information about the structure of the HRTF data.

```python
hrtf = slab.HRTF.kemar()
print(hrtf)
# <class 'hrtf.HRTF'> sources 710, elevations 14, samples 710, samplerate 44100.0
```

Libraries of many other recordings can be found on the [website of the sofa file format](https://sofa.sourceforge.net/).

**Note:** The class is at the moment geared towards plotting and analysis of HRTF files in the sofa format, because we needed that functionality for grant applications. The functionality will grow as we start to record and manipulate HRTFs more often.

**Note:** When we started writing this code, there was no python module for reading and writing sofa files. Now that `pysofaconventions` is available, we will at some point switch internally to using that module as backend for reading sofa files, instead of our own limited implementation.

3.5.2 Plotting sources

The HRTF is a set of many transfer functions, each belonging to a certain sound source position (for example, there are 710 sources in the KEMAR recordings). You can plot the source positions in 3D with the `plot_sources()` to get an impression of the density of the recordings. The red dot indicates the position of the listener and the red arrow indicates the lister’s gaze direction. Optionally, you can supply a list of source indices which will be highlighted in red. This can be useful when you are selecting source locations for an experiment and want to confirm that you chose correctly. In the example below, we select sources using the methods `elevation_sources()`, which selects sources along a horizontal slice at a given elevation and `cone_sources()`, which selects sources along a vertical slice through the source sphere in front of the listener a given angular distance away from the midline:

```python
# cone_sources and elevation_sources return lists of indices which are concatenated by adding:
hrtf = slab.HRTF.kemar()
sourceidx = hrtf.cone_sources(0) + hrtf.elevation_sources(0)
hrtf.plot_sources(sourceidx) # plot the sources in 3D, highlighting the selected sources
```

Try a few angles for the `elevation_sources()` and `cone_sources()` methods to understand how selecting the sources works!
Chapter 3. Citing slab
### 3.5.3 Plotting transfer functions

As mentioned before, a HRTF is collection of transfer functions. Each single transfer function is an instance of the `slab.Filter` with two channels - one for each ear. The transfer functions are located in the `data` list and the coordinates of the corresponding sources in the `sources` list. In the example below, we select a source, print it’s coordinates and plot the corresponding transfer function.

```python
from matplotlib import pyplot as plt
hrtf = slab.HRTF.kemar()
fig, ax = plt.subplots(1)
idx = 10
source = hrtf.sources[idx]  # the source's azimuth, elevation and distance
filt = hrtf.data[idx]  # the corresponding filter
fig.suptitle(f"source at azimuth {source[0].round(2)} and elevation {source[1]}"
 filt.channel(0).tf(axis=ax, show=False)
filt.channel(1).tf(axis=ax, show=False)
plt.legend()
plt.show()
```

The HRTF class also has a `plot_tf()` method to plot transfer functions as either waterfall (as is Wightman and Kistler, 1989), image plot (as in Hofman 1998). The function takes a list of source indices as an argument which will be included in the plot. The function below shows how to generate a waterfall and image plot for the sources along the central cone. Before plotting, we apply a diffuse field equalization to remove non-spatial components of the HRTF, which makes the features of the HRTF that change with direction easier to see:

```python
from slab import data_path
from matplotlib import pyplot as plt
hrtf = slab.HRTF.kemar()
fig, ax = plt.subplots(2)
dtf = hrtf.diffuse_field_equalization()
sourceidx = hrtf.cone_sources(0)
ax[0].set_title("waterfall plot")
ax[1].set_title("image plot")
hrtf.plot_tf(sourceidx, ear='left', axis=ax[0], show=False, kind="waterfall")
hrtf.plot_tf(sourceidx, ear='left', axis=ax[1], show=False, kind="image")
plt.tight_layout()
plt.show()
```

As you can see the HRTF changes systematically with the elevation of the sound source, especially for frequencies above 6 kHz. Individual HRTFs vary in the amount of spectral change across elevations, mostly due to differences in the shape of the ears. You can compute a measure of the HRTFs spectral dissimilarity the vertical axis, called vertical spatial information (VSI, Trapeau and Schönwiesner, 2016). The VSI relates to behavioral localization accuracy in the vertical dimension: listeners with acoustically more informative spectral cues tend to localize sounds more accurately in the vertical axis. Identical filters give a VSI of zero, highly dissimilar filters give a VSI closer to one. The hrtf has to be diffuse-field equalized for this measure to be sensible, and the `vsi()` method will apply the equalization. The KEMAR mannequin have a VSI of about 0.73:

```python
hrtf.vsi()
# .73328
```

The `vsi()` method accepts arbitrary lists of source indices for the dissimilarity computation. We can for instance check how the VSI changes when sources further off the midline are used. There are some reports in the literature that listeners can perceive the elevation of a sound source better if it is a few degrees to the side. We can check whether this is due to more dissimilar filters at different angles (we’ll reuse the `dtf` from above to avoid recalculation of the diffuse-field equalization in each iteration):

```python
hrtf.vsi()
# .73328
```
for cone in range(0,51,10):
    sources = dtf.cone_sources(cone)
    vsi = dtf.vsi(sources=sources, equalize=False)
    print(f'{cone}: {vsi:.2f}')
    # 0: 0.73
    # 10: 0.63
    # 20: 0.69
    # 30: 0.74
    # 40: 0.76
    # 50: 0.73

The effect seems to be weak for KEMAR, (VSI falls off for directions slightly off the midline and then increases again at around 30-40).

### 3.5.4 Virtually displaying 3D sound

The HRTF describes the directional filtering of incoming sounds by the listeners ears, head and torso. Since this is the basis for localizing sounds in three dimensions, we can apply the HRTF to a sound to evoke the impression of it coming from a certain direction in space when played through headphones. The HRTF.apply() method returns an instance of the slab.Binaural. It is important to use the apply() method of the HRTF class instead of the apply() method of the individual Filter class objects in the HRTF, because only the method:HRTF.apply method conserves ITDs. The Filter.apply() method does not do that, because when applying a generic filter, you normally do not want to introduce delays.

In the example below we apply the transfer functions corresponding to three sound sources at different elevations along the vertical midline to white noise.

```python
from slab import data_path, Sound
from matplotlib import pyplot as plt
hrtf = slab.HRTF.kemar()
sound = slab.Sound.pinknoise(samplerate=hrtf.samplerate)  # the sound to be spatialized
fig, ax = plt.subplots(3)
sourceidx = [0, 260, 536]  # sources at elevations -40, 0 and 40
spatial_sounds = []
for i, index in enumerate(sourceidx):
    spatial_sounds.append(hrtf.apply(index, sound))
    # only plot frequencies above 5kHz because low frequencies are unaffected by the HRTF
    spatial_sounds[i].spectrum(axis=ax[i], low_cutoff=5000, show=False)
plt.show()
```

You can use the play() method of the sounds to listen to them - see if you can identify the virtual sound source position. Your ability to do so depends on how similar your own HRTF is to that of the the KEMAR artificial head. Your auditory system can get used to new HRTFs, so if you listen to the KEMAR recordings long enough they will eventually produce virtual sound sources at the correct locations.

Binaural filters from the KEMAR HRTF will impose the correct spectral profile, but no ITD. After applying an HRTF filter corresponding to an off-center direction, you should also apply an ITD corresponding to the direction using the Binaural.azimuth_to_itd() and Binaural.itd() methods.

Finally, the HRTF filters are recorded only at certain locations (710, in case of KEMAR - plot the source locations to inspect them). You can interpolate a filter for any location covered by these sources with the HRTF.interpolate() method. It triangulates the source locations and finds three sources that form a triangle around the requested location and interpolate a filter with a (barycentric) weighted average in the spectral domain. The resulting filter may not have the same overall gain, so remember to set the level of your stimulus after having applied the interpolated HRTF.
3.6 Worked examples

The folder slab.experiments contains the full code from actual psychoacoustic experiments in our lab. We use this folder mainly to make the code available and enable easy replication. The examples are well documented and may give you and idea of the typical structure of such experiments. To run these experiments, import them from slab.experiments:

```python
from slab.experiments import motion_speed
motion_speed.main_experiment(subject='test')
```

Currently available are:

```python
slab.experiments.room_voice_interference.main_experiment(subject=None, do_jnd=True, do_interference=True)
```

Interference between room and voice processing. Pre-recorded voice recordings are presented in different simulated rooms (the large stimulus set is not included). Just-noticeable differences for changes in room volume and voice parameters (glottal pulse rate and vocal tract length) are first measured, then 3-alternative- forced-choice trial are presented with the reference in a larger room. Does a simultaneous voice change impede the detection of the room change? The experiment requires a set of recorded spoken word stimuli, each of which was offline manipulated to change speaker identity (using the STRAIGHT algorithm) and then run through a room acoustics simulation to add reverberation consistent with rooms of different sizes. The filenames of the recordings contain the word and the voice and room parameters, so that the correct file is loaded for presentation.

This experiment showcases participant data handling, AFC trials, prerecorded stimulus handling, among others.

```python
slab.experiments.motion_speed.main_experiment(subject=None)
```

A complex spatially extended moving sound is generated ('moving_gaussian'). This stimulus simulates the acoustics of a free-field loudspeaker arc. A gaussian profile moves from left to right or right to left across the virtual speaker array and the speed of the movement and modulation depth (across space) can be varied. Detection thresholds for motion direction are measured at different motion speeds. Then the effect of adaptation by a long moving adapter at one speed on the detectability of motion at different speeds is measured.

This experiment showcases complex stimulus generation and staircases, among others.

3.6.1 Quick standard experiments:

**Audiogram**

Run a pure tone audiogram at the standard frequencies 125, 250, 500, 1000, 2000, 4000 Hz using an adaptive staircase:

```python
from matplotlib import pyplot as plt
def audiogram(frequency, duration):
    stimulus = slab.Sound.tone(frequency=frequency, duration=0.5)
    stairs = slab.Staircase(start_val=50, n_reversals=18)
    print(f'Starting staircase with {frequency} Hz: 
    for level in stairs:
        stimulus.level = level
        stairs.present_tone_trial(stimulus)
        stairs.print_trial_info()
        thresholds.append(stairs.threshold())
        print(f'Threshold at {frequency} Hz: {stairs.threshold()} dB')
    plt.plot(frequency, thresholds) # plot the audiogram
```
Temporal modulation transfer function

Measure temporal modulation transfer functions via detection thresholds for amplitude modulations. The parameters of the test replicate Fig. 2 in Viemeister\textsuperscript{1979} and present sinusoidal 2 to 4000 Hz modulations in a 77-dB wideband noise carrier using an adaptive staircase.

```python
from matplotlib import pyplot as plt
mod_freqs = [2, 4, 8, 16, 32, 64, 125, 250, 500, 1000, 2000, 4000]
threshs = []
base_stimulus = slab.Sound.pinknoise(duration=1.)
base_stimulus.level = 77
for frequency in mod_freqs:
    stairs = slab.Staircase(start_val=0.8, n_reversals=16, step_type='db',
                             step_sizes=[4, 2], min_val=0, max_val=1, nup=1, ndown=2)
    print(f'Starting staircase with {frequency} Hz: ')
    for depth in stairs:
        stimulus = base_stimulus.am(frequency=frequency, depth=depth)
        stairs.present_afc_trial(stimulus, base_stimulus)
        threshs.append(stairs.threshold(n=14))
    print(f'Threshold at {frequency} Hz: {stairs.threshold(n=14)} modulation depth')
plt.plot(freqs, threshs)  # plot the transfer function
```

3.7 Reference documentation

Note: This reference documentation is auto-generated from the doc strings in the module. For a tutorial-like overview of the functionality of slab, please see the previous sections.

3.7.1 Sounds

Inherits from `slab.Signal`.

```python
class slab.Sound(data, samplerate=None)
```

Class for working with sounds, including loading/saving, manipulating and playing. Inherits from the base class `slab.Signal`. Instances of Sound can be created by either loading a file, passing an array of values and a samplerate or by using one of the sound-generating methods of the class (all of the @staticmethods).

Parameters

- **data** (str | pathlib.Path | numpy.ndarray | slab.Signal | list) – Given a string or Path pointing to the .wav file, the data and samplerate will be loaded from the file. Given and array, and instance of a Signal or a list, the data will be passed to the super class (see documentation of slab.Signal).
- **samplerate** (int | float) – must only be defined when creating a Sound from an array.

- **.data** the data-array of the Sound object which has the shape `n_samples x n_channels`.
- **.n_channels** the number of channels in data.

\textsuperscript{1979} Viemeister (1979) Temporal modulation transfer functions based upon modulation thresholds. JASA 66(5), 1364–1380
.n_samples
  the number of samples in data. Equals duration * samplerate.

duration
  the duration of the sound in seconds. Equals n_samples / samplerate.

Examples:

```python
import slab, numpy
# generate a Sound object from an array of random floats:
sig = slab.Sound(data=numpy.random.randn(10000), samplerate=41000)
# generate a Sound object using one of the modules methods, like 'tone':
sig = slab.Sound.tone()  # generate a tone
sig.level = 80  # set the level to 80 dB
sig = sig.ramp(duration=0.05)  # add a 50 millisecond ramp
sig.spectrum(log_power=True)  # plot the spectrum
sig.waveform()  # plot the time courses
```

property level
Can be used to get or set the rms level of a sound, which should be in dB. For single channel sounds a value in dB is used, for multiple channel sounds a value in dB can be used for setting the level (all channels will be set to the same level), or a list/tuple/array of levels. Use slab.calibrate() to make the computed level reflect output intensity.

static read(filename)
Load a wav file and create an instance of Sound.

Parameters
  filename (str) – the full path to a (.wav) file.

Returns the sound generated with the data and samplerate from the file.

Return type (slab.Sound)

static tone
Generate a pure tone.

Parameters
  • frequency (int | float | list) – frequency of the tone. Given a list of length n_channels, one element of the list is used as frequency for each channel.

  • duration (float | int) – duration of the sound in seconds (given a float) or in samples (given an int).

  • phase (int | float | list) – phase of the sinusoid, defaults to 0. Given a list of length n_channels, one element of the list is used as phase for each channel.

  • samplerate (int | None) – the samplerate of the sound. If None, use the default samplerate.

  • level (None | int | float | list) – the sounds level in decibel. For a multichannel sound, a list of values can be provided to set the level of each channel individually. If None, the level is set to the default

  • n_channels (int) – number of channels, defaults to one.

Returns the tone generated from the parameters.

Return type (slab.Sound)
static dynamic_tone(frequencies=None, times=None, phase=0, samplerate=None, level=None, n_channels=1)

Generate a sinusoid with time-varying frequency from a list of frequencies and times. If times is None, a sound with len(frequencies) samples is generated. Be careful when giving a list of times: integers are treated as samples and floats as seconds for each list element independently. So times=[0, 0.5, 1] is probably a mistake, because the last entry is treated a sample number 1, instead of the intended 1 second time point. This will raise and error because the time values are not monotonically ascending. Correct would be [0., .5, 1] or [0, 4000, 8000] for the default samplerate.

Parameters

- **frequencies** (list | numpy.ndarray) – frequencies of the tone. Intermediate values are linearily interpolated.
- **times** (list | numpy.ndarray | None) – list of time points corresponding to the given frequencies. Must have same length as frequencies if given. If None, frequencies are assumed to correspond to consecutive samples. Integer values specify times in samples, and floats specify times in seconds.
- **phase** (int | float) – initial phase of the sinusoid, defaults to 0
- **samplerate** (None | int) – the samplerate of the sound. If None, use the default samplerate.
- **level** (None | int | float) – the sound’s level in decibel. If None, use the default samplerate.
- **n_channels** (int) – number of channels, defaults to one.

Returns

the tone generated from the parameters.

Return type

(slab.Sound)

static harmoniccomplex(f0=500, duration=1.0, amplitude=0, phase=0, samplerate=None, level=None, n_channels=1)

Generate a harmonic complex tone composed of pure tones at integer multiples of the fundamental frequency.

Parameters

- **f0** (int) – the fundamental frequency. Harmonics will be generated at integer multiples of this value.
- **duration** (float | int) – duration of the sound in seconds (given a float) or in samples (given an int).
- **amplitude** (int | float | list) – Amplitude in dB, relative to the full scale (i.e. 0 corresponds to maximum intensity, -30 would be 30 dB softer). Given a single int or float, all harmonics are set to the same amplitude and harmonics up to 1/5th of of the samplerate are generated. Given a list of values, the number of harmonics generated is equal to the length of the list with each element of the list setting the amplitude for one harmonic.
- **phase** (int | float | string | list) – phase of the sinusoid, defaults to 0. Given a list (with the same length as the one given for the amplitude argument) every element will be used as the phase of one harmonic. Given a string, its value must be schroeder, in which case the harmonics are in Schröder phase, producing a complex tone with minimal peak-to-peak amplitudes (Schroeder 1970).
- **samplerate** (int | None) – the samplerate of the sound. If None, use the default samplerate.
• **level** *(None | int | float | list)* – the sounds level in decibel. For a multichannel sound, a list of values can be provided to set the level of each channel individually. If None, the level is set to the default.

• **n_channels** *(int)* – number of channels, defaults to one.

**Returns** the harmonic complex generated from the parameters.

**Return type** *(slab.Sound)*

Examples:

```python
sig = slab.Sound.harmoniccomplex(f0=200, amplitude=[0,-10,-20,-30])  # generate the harmonic complex tone
_ = sig.spectrum()  # plot it's spectrum
```

### static whitenoise *(duration=1.0, samplerate=None, level=None, n_channels=1)*

Generate white noise.

**Parameters**

• **duration** *(float | int)* – duration of the sound in seconds (given a float) or in samples (given an int).

• **samplerate** *(int | None)* – the samplerate of the sound. If None, use the default samplerate.

• **level** *(None | int | float | list)* – the sounds level in decibel. For a multichannel sound, a list of values can be provided to set the level of each channel individually.

• **n_channels** *(int)* – number of channels, defaults to one. If channels > 1, several channels of uncorrelated noise are generated.

**Returns** the white noise generated from the parameters.

**Return type** *(slab.Sound)*

Examples:

```python
noise = slab.Sound.whitenoise(1.0, n_channels=2).  # generate a 1 second white noise with two channels
```

### static powerlawnoise *(duration=1.0, alpha=1, samplerate=None, level=None, n_channels=1)*

Generate a power-law noise with a spectral density per unit of bandwidth scales as 1/(f**alpha).

**Parameters**

• **duration** *(float | int)* – duration of the sound in seconds (given a float) or in samples (given an int).

• **alpha** *(int)* – power law exponent.

• **samplerate** *(int | None)* – output samplerate

• **samplerate** – the samplerate of the sound. If None, use the default samplerate.

• **level** *(None | int | float | list)* – the sounds level in decibel. For a multichannel sound, a list of values can be provided to set the level of each channel individually. If None, the level is set to the default.

• **n_channels** *(int)* – number of channels, defaults to one. If channels > 1, several channels of uncorrelated noise are generated.

**Returns** the power law noise generated from the parameters.
### Examples:

```python
# Generate and plot power law noise with three different exponents
from matplotlib import pyplot as plt
fig, ax = plt.subplots()
alpha = [1, 2, 3]:
    noise = slab.Sound.powerlawnoise(0.2, alpha, samplerate=8000)
    noise.spectrum(axis=ax, show=False)
plt.show()
```

#### static pinknoise(duration=1.0, samplerate=None, level=None, n_channels=1)
Generate pink noise (power law noise with exponent \(\alpha=1\). This is simply a wrapper for calling the `powerlawnoise` method.

**Parameters**

- `slab.Sound.powerlawnoise` (see)

**Returns**

- power law noise generated from the parameters with exponent \(\alpha=1\).

**Return type** (slab.Sound)

#### static irn(frequency=100, gain=1, n_iter=4, duration=1.0, samplerate=None, level=None, n_channels=1)
Generate iterated ripple noise (IRN). IRN is a broadband noise with temporal regularities, which can give rise to a perceptible pitch. Since the perceptual pitch to noise ratio of these stimuli can be altered without substantially altering their spectral content, they have been useful in exploring the role of temporal processing in pitch perception [Yost 1996, JASA].

**Parameters**

- `frequency` (int | float) – the frequency of the signals perceived pitch in Hz.
- `gain` (int | float) – multiplicative factor of the repeated additions. Smaller values reduce the temporal regularities in the resulting IRN.
- `n_iter` (int) – number of iterations of additions. Higher values increase pitch saliency.
- `duration` (float | int) – duration of the sound in seconds (given a float) or in samples (given an int).
- `samplerate` (int | None) – the samplerate of the sound. If None, use the default samplerate.
- `level` (None | int | float | list) – the sounds level in decibel. For a multichannel sound, a list of values can be provided to set the level of each channel individually. If None, the level is set to the default
- `n_channels` (int) – number of channels, defaults to one. If channels > 1, several channels with copies of the noise are generated.

**Returns**

- ripple noise that has a perceived pitch at the given frequency.

**Return type** (slab.Sound)

#### static click(duration=0.0001, samplerate=None, level=None, n_channels=1)
Generate a click (a sequence of ones).

**Parameters**

- `duration` (float | int) – duration of the sound in seconds (given a float) or in samples (given an int).
• **samplerate** *(int / None)* – the samplerate of the sound. If None, use the default samplerate.

• **level** *(None / int / float / list)* – the sounds level in decibel. For a multichannel sound, a list of values can be provided to set the level of each channel individually. If None, the level is set to the default.

• **n_channels** *(int)* – number of channels, defaults to one.

**Returns** click generated from the given parameters.

**Return type** *(slab.Sound)*

### slab.clicktrain

`slab.clicktrain(duration=1.0, frequency=500, clickduration=0.0001, level=None, samplerate=None)`

Generate a series of n clicks (by calling the **click** method) with a perceived pitch at the given frequency.

**Parameters**

• **duration** *(float / int)* – duration of the sound in seconds (given a float) or in samples (given an int).

• **frequency** *(float / int)* – the frequency of the signals perceived pitch in Hz.

• **clickduration** *(float / int): duration of a single click in seconds (given a float) or in samples (given an int). The number of clicks in the train is given by duration / clickduration.*

• **samplerate** *(int / None)* – the samplerate of the sound. If None, use the default samplerate.

• **level** *(None / int / float / list)* – the sounds level in decibel. For a multichannel sound, a list of values can be provided to set the level of each channel individually. If None, the level is set to the default.

**Returns** click train generated from the given parameters.

**Return type** *(slab.Sound)*

### slab.chirp

`slab.chirp(duration=1.0, from_frequency=100, to_frequency=None, samplerate=None, level=None, kind='quadratic')`

Returns a pure tone with in- or decreasing frequency using the function scipy.sound.chirp.

**Parameters**

• **duration** *(float / int)* – duration of the sound in seconds (given a float) or in samples (given an int).

• **from_frequency** *(float / int)* – the frequency of tone in Hz at the start of the sound.

• **to_frequency** *(float / int / None)* – the frequency of tone in Hz at the end of the sound. If None, the nyquist frequency (samplerate / 2) will be used.

• **samplerate** *(int / None)* – the samplerate of the sound. If None, use the default samplerate.

• **level** *(None / int / float / list)* – the sounds level in decibel. For a multichannel sound, a list of values can be provided to set the level of each channel individually. If None, the level is set to the default.

• **kind** *(str)* – determines the type of ramp (see scipy.sound.chirp() for options).

**Returns** chirp generated from the given parameters.

**Return type** *(slab.Sound)*

### slab.silence

`slab.silence(duration=1.0, samplerate=None, n_channels=1)`

Generate silence (all samples equal zero).

3.7. Reference documentation
Parameters

- **duration** *(float / int)* – duration of the sound in seconds (float) or samples (int).
- **samplerate** *(int / None)* – the samplerate of the sound. If None, use the default samplerate.
- **n_channels** *(int)* – number of channels, defaults to one.

Returns silence generated from the given parameters.

Return type *(slab.Sound)*

```python
static vowel(vowel='a', gender=None, glottal_pulse_time=12, formant_multiplier=1, duration=1.0, samplerate=None, level=None, n_channels=1)
```

Generate a sound resembling the human vocalization of a vowel.

Parameters

- **vowel** *(str / None)* – kind of vowel to generate can be: ‘a’, ‘e’, ‘i’, ‘o’, ‘u’, ‘ae’, ‘oe’, or ‘ue’. For these vowels, the function has pre-set format frequencies. If None, a vowel will be generated from random formant frequencies in the range of the existing vowel formants.
- **gender** *(str / None)* – Setting the gender (‘male’, ‘female’) is a shortcut for setting the arguments glottal_pulse_time and formant_multiplier.
- **glottal_pulse_time** *(int / float)* – the distance between glottal pulses in milliseconds (determines vocal tract length).
- **formant_multiplier** *(int / float)* – multiplier for the pre-set formant frequencies (scales the voice pitch).
- **duration** *(float / int)* – duration of the sound in seconds (given a float) or in samples (given an int).
- **samplerate** *(int / None)* – the samplerate of the sound. If None, use the default samplerate.
- **level** *(None / int / float / list)* – the sounds level in decibel. For a multichannel sound, a list of values can be provided to set the level of each channel individually. If None, the level is set to the default
- **n_channels** *(int)* – number of channels, defaults to one.

Returns vowel generated from the given parameters.

Return type *(slab.Sound)*

```python
static multitone_masker(duration=1.0, low_cutoff=125, high_cutoff=4000, bandwidth=0.3333333333333333, samplerate=None, level=None)
```

Generate noise made of ERB-spaced random-phase pure tones. This noise does not have random amplitude variations and is useful for testing CI patients [Oxenham 2014, Trends Hear].

Parameters

- **duration** *(float / int)* – duration of the sound in seconds (given a float) or in samples (given an int).
- **low_cutoff** *(int / float)* – the lower frequency limit of the noise in Hz
- **high_cutoff** *(int / float)* – the upper frequency limit of the noise in Hz
- **bandwidth** *(float)* – the signals bandwidth in octaves.
- **samplerate** *(int / None)* – the samplerate of the sound. If None, use the default samplerate.
• **level** *(None / int / float / list)* – the sound’s level in decibel. For a multichannel sound, a list of values can be provided to set the level of each channel individually. If None, the level is set to the default

**Returns** multi tone masker noise, generated from the given parameters.

**Return type** *(slab.Sound)*

**Examples:**

```python
sig = Sound.multitone_masker()
sig = sig.ramp()
sig.spectrum()
```

---

**static equally_masking_noise** *(duration=1.0, low_cutoff=125, high_cutoff=4000, samplerate=None, level=None)*

Generate an equally-masking noise (ERB noise) within a given frequency band.

**Parameters**

- **duration** *(float / int)* – duration of the sound in seconds (given a float) or in samples (given an int).
- **low_cutoff** *(int / float)* – the lower frequency limit of the noise in Hz
- **high_cutoff** *(int / float)* – the upper frequency limit of the noise in Hz
- **samplerate** *(int / None)* – the samplerate of the sound. If None, use the default samplerate.
- **level** *(None / int / float / list)* – the sound’s level in decibel. For a multichannel sound, a list of values can be provided to set the level of each channel individually. If None, the level is set to the default

**Returns** equally masking noise noise, generated from the given parameters.

**Return type** *(slab.Sound)*

**Examples:**

```python
sig = Sound.erb_noise()
sig.spectrum()
```

---

**static sequence** *(*sounds)*

Join sounds into a new sound object.

**Parameters** *sounds* *(slab.Sound)* – two or more sounds to combine.

**Returns** the input sounds combined in a single object.

**Return type** *(slab.Sound)*

**write** *(filename, normalise=True, fmt='WAV')*

Save the sound as a WAV.

**Parameters**

- **filename** *(str / pathlib.Path)* – path, the file is written to.
- **normalise** *(bool)* – if True, the maximal amplitude of the sound is normalised to 1.
- **fmt** *(str)* – data format to write. See soundfile.available_formats().

**ramp** *(when=’both’, duration=0.01, envelope=None)*

Adds an on and/or off ramp to the sound.
Parameters

- **when** *(str)* – can take values ‘onset’, ‘offset’ or ‘both’
- **duration** *(float | int)* – duration of the sound in seconds (given a float) or in samples (given an int).
- **envelope** *(callable)* – function to compute the samples of the ramp, defaults to a sine

Returns copy of the sound with the added ramp(s)

Return type *(slab.Sound)*

**repeat**(n)

Repeat the sound n times.

Parameters **n** *(int)* – the number of repetitions.

Returns copy of the sound repeated n times.

Return type *(slab.Sound)*

**static crossfade**(sounds, overlap=0.01)

Crossfade several sounds.

Parameters

- **sounds** *(instances of slab.Sound)* – sounds to crossfade
- **overlap** *(float | int)* – duration of the overlap between the cross-faded sounds in seconds (given a float) or in samples (given an int).

Returns

A single sound that contains all input sounds cross-faded. The duration will be the sum of the input sound’s durations minus the overlaps.

Return type *(slab.Sound)*

Examples:

```python
noise = Sound.whitenoise(duration=1.0)
vowel = Sound.vowel()
noise2vowel = Sound.crossfade(vowel, noise, vowel, overlap=0.4)
noise2vowel.play()
```

**pulse**(frequency=4, duty=0.75, gate_time=0.005)

Apply a pulsed envelope to the sound.

Parameters

- **frequency** *(float)* – the frequency of pulses in Hz.
- **duty** *(float)* – duty cycle, i.e. ratio between the pulse duration and pulse period, values must be between 1 (always high) and 0 (always low). When using values close to 0, gate_time may need to be decreased to avoid on and off ramps being longer than the pulse.
- **gate_time** *(float)* – rise/fall time of each pulse in seconds

Returns pulsed copy of the instance.

Return type *slab.Sound*

**am**(frequency=10, depth=1, phase=0)

Apply an amplitude modulation to the sound by multiplication with a sine function.
Parameters

- **frequency** (*int*) – frequency of the modulating sine function in Hz
- **depth** (*int, float*) – modulation depth/index of the modulating sine function
- **phase** (*int, float*) – initial phase of the modulating sine function

**Returns** amplitude modulated copy of the instance.

**Return type** `slab.Sound`  

```python
def filter(frequency=100, kind='hp')
    Convenient wrapper for the Filter class for a standard low-, high-, bandpass, and bandstop filter.

    Parameters

    - **frequency** (*int, tuple*) – cutoff frequency in Hz. Integer for low- and highpass filters, tuple with lower and upper cutoff for bandpass and -stop.
    - **kind** (*str*) – type of filter, can be “lp” (lowpass), “hp” (highpass) “bp” (bandpass) or “bs” (bandstop)

    **Returns** filtered copy of the instance.

    **Return type** `slab.Sound`
```

```python
def aweight()
    Returns A-weighted version of the sound. A-weighting is applied to instrument-recorded sounds to account for the relative loudness of different frequencies perceived by the human ear. See: https://en.wikipedia.org/wiki/A-weighting.
```

```python
def static record(duration=1.0, samplerate=None)
    Record from inbuilt microphone. Uses SoundCard module if installed [recommended], otherwise uses SoX.

    Parameters

    - **duration** (*float | int*) – duration of the sound in seconds (given a float) or in samples (given an int). Note that duration has to be in seconds when using SoX
    - **samplerate** (*int | None*) – the samplerate of the sound. If None, use the default samplerate. Note that most sound cards can only record at 44100 Hz samplerate.

    **Returns** The recorded sound.

    **Return type** `(slab.Sound)`
```

```python
def play()
    Plays the sound through the default device. If the soundcard module is installed it is used to play the sound. Otherwise the sound is saved as .wav to a temporary directory and is played via the `play_file` method.
```

```python
static play_file(filename)
    Play a .wav file using the OS-specific mechanism for Windows, Linux or Mac.

    Parameters **filename** (*str | pathlib.Path*) – full path to the .wav file to be played.
```

```python
def waveform(start=0, end=None, show=True, axis=None)
    Plot the waveform of the sound.

    Parameters

    - **start** (*int | float*) – start of the plot in seconds (float) or samples (int), defaults to 0
    - **end** (*int | float | None*) – the end of the plot in seconds (float) or samples (int), defaults to None.
```

3.7. Reference documentation 47
• `show (bool)` – whether to show the plot right after drawing.

• `axis (matplotlib.axes.Axes | None)` – axis to plot to. If None create a new plot.

`spectrogram(window_dur=0.005, dyn_range=120, upper_frequency=None, other=None, show=True, axis=None, **kwargs)`

Plot a spectrogram of the sound or return the computed values.

**Parameters**

• `window_dur` – duration in samples (int) or seconds (float) of time window for short-term FFT, default 0.005 s.

• `dyn_range` – dynamic range in dB to plot, defaults to 120.

• `upper_frequency` – The upper frequency limit of the plot. If None use the maximum.

• `other (slab.Sound)` – if a sound object is given, subtract the waveform and plot the difference spectrogram.

• `show (bool)` – whether to show the plot right after drawing. Note that if show is False and no axis is passed, no plot will be created.

• `axis (matplotlib.axes.Axes | None)` – axis to plot to. If None create a new plot.

• `**kwargs` – keyword arguments for computing the spectrogram. See documentation for scipy.signal.spectrogram.

**Returns**

If `show == True` or an axis was passed, a plot is drawn and nothing is returned. Else, a tuple is returned which contains frequencies, time bins and a 2D array of powers.

**Return type** (None | tuple)

`cochleagram(bandwidth=0.2, show=True, axis=None)`

Computes a cochleagram of the sound by filtering with a bank of cosine-shaped filters with given bandwidth and applying a cube-root compression to the resulting envelopes.

**Parameters**

• `bandwidth (float)` – filter bandwidth in octaves.

• `show (bool)` – whether to show the plot right after drawing. Note that if show is False and no axis is passed, no plot will be created.

• `axis (matplotlib.axes.Axes | None)` – axis to plot to. If None create a new plot.

**Returns**

If `show == True` or an axis was passed, a plot is drawn and nothing is returned. Else, an array with the envelope is returned.

**Return type** (None | numpy.ndarray)

`spectrum(low_cutoff=16, high_cutoff=None, log_power=True, axis=None, show=True)`

Compute the spectrum of the sound.

**Parameters**

• `low_cutoff` – If these are left unspecified, it shows the full spectrum, otherwise it shows only between low and high in Hz.

• `log_power (bool)` – whether to compute the log of the power.
• **show**(bool) – whether to show the plot right after drawing. Note that if show is False and no **axis** is passed, no plot will be created.

• **axis**(matplotlib.axes.Axes / None) – axis to plot to. If None create a new plot.

**Returns**

If show=False, returns *Z, freqs*, where *Z* is a 1D array of powers and *freqs* are the corresponding frequencies.

**spectral_feature**(feature='centroid', mean='rms', frame_duration=None, rolloff=0.85)

Computes one of several features of the spectrogram of a sound for each channel.

**Parameters**

• **feature**(str) – the kind of feature to compute, options are: “centroid”, the center of mass of the short-term spectrum, “fwhm”, the width of a Gaussian of the same variance as the spectrum around the centroid, “flux”, a measure of how quickly the power spectrum of a sound is changing, “flatness”, measures how tone-like a sound is, as opposed to being noise-like, “rolloff”, the frequency at which the spectrum rolls off.

• **mean**(str / None) – method of computing the mean of the feature value over all samples. Can be “rms”, “average” or None. If None, a new sound with the feature value at each sample is generated.

• **frame_duration**(float) – duration of frames in samples (int) or seconds (float) in which to compute features, defaults to 0.05 s

• **rolloff**(float) – only used if feature is “rolloff”, fraction of spectral power below the rolloff frequency

**Returns** Mean feature for each channel in a list or a new Signal of feature values.

**Return type** (list | slab.Signal)

**vocode**(bandwidth=0.3333333333333333)

Returns a noise vocoded version of the sound by computing the envelope in different frequency subbands, filling these envelopes with noise, and collapsing the subbands into one sound. This removes most spectral information but retains temporal information in a speech sound.

**Parameters** bandwidth(float) – width of the subbands in octaves.

**Returns** a vocoded copy of the sound.

**Return type** (slab.Sound)

**crest_factor**()

The crest factor is the ratio of the peak amplitude and the RMS value of a waveform and indicates how extreme the peaks in a waveform are. Returns the crest factor in dB. Numerically identical to the peak-to-average power ratio.

**Returns** the crest factor or NaN if there are no peaks in the sound.

**Return type** (float | numpy.nan)

**onset_slope**()

Compute the centroid of a histogram of onset slopes as a measure of how many quick intensity increases the sound has. These onset-like features make the sound easier to localize via envelope ITD.

**Returns** the histograms centroid or 0 if there are no onsets in the sound.

**Return type** (float)
frames(duration=1024)
A generator that steps through the sound in overlapping, windowed frames. Get the frame center times by calling Sound’s frametimes method.

Parameters
• duration (int | float) – half of the length of the returned frames in samples (int) or seconds (float),
• be larger than 7 samples. (must) –

Returns the generator object that yields frames which are of the same type as the object.

Return type (generator)

Examples:
```python
sound = slab.Sound.vowel()
windows = sound.frames()
for window in windows:  # get the flatness of each frame
    print(window.spectral_feature("flatness"))
```

frametimes(duration=1024)
Returns the time points at the frame centers constructed by the frames method.

Parameters
• duration (int | float) – half of the length of the returned frames in samples (int) or seconds (float),
• be larger than 7 samples. (must) –

Returns the center of each frame in seconds.

Return type (numpy.ndarray)

sound.apply_to_path(method=None, kwargs=None, out_path=None)
Apply a function to all wav files in a given directory.

Parameters
• path (str | pathlib.Path) – path to the folder from which wav files are collected for processing.
• method (callable) – function to be applied to each file.
• kwargs (dict) – dictionary of keyword arguments and values passed to the function.
• out_path (str | pathlib.Path) – if is supplied, sounds are saved with their original file name in this directory.

Examples:
```python
slab.apply_to_path('.', slab.Sound.spectral_feature, {'feature': 'fwhm'})
slab.apply_to_path('.', slab.Sound.ramp, out_path='.modified')
slab.apply_to_path('.', slab.Sound.ramp, kwargs={'duration': 0.3}, out_path='.test')
```
Signal

`slab.Sound` inherits from Signal, which provides basic methods to handle signals:

```python
class slab.Signal(data, samplerate=None)
```

Base class for Signal data (from which the Sound and Filter class inherit). Provides arithmetic operations, slicing, and conversion between samples and times.

**Parameters**

- `data` *(numpy.ndarray | slab.Signal | list)* – samples of the sound. If it is an array, the first dimension should represent the number of samples and the second one the number of channels. If it’s an object, it must have a `.data` attribute containing an array. If it’s a list, the elements can be arrays or objects. The output will be a multi-channel sound with each channel corresponding to an element of the list.

- `samplerate` *(int | None)* – the samplerate of the sound. If None, use the default samplerate.

**Examples:**

```python
import slab, numpy
sig = slab.Signal(numpy.ones([10,2]),samplerate=10)  # create a sound
sig[:5] = 0  # set the first 5 samples to 0
sig[:,:,1]  # select the data from the second channel
sig2 = sig * 2  # multiply each sample by 2
sig_inv = -sig  # invert the phase
```

**property n_samples**

The number of samples in the Signal.

**property duration**

The length of the Signal in seconds.

**property times**

An array of times (in seconds) corresponding to each sample.

**property n_channels**

The number of channels in the Signal.

**static in_samples(ctime, samplerate)**

Converts time values in seconds to samples. This is used to enable input in either samples (integers) or seconds (floating point numbers) in the class.

**Parameters**

- `ctime` *(int | float | list | numpy.ndarray)* – the time(s) to convert to samples.

- `samplerate` *(int)* – the samplerate of the sound.

**Returns** the time(s) in samples.
**Return type** (int | list | numpy.ndarray)

```python
static set_default_samplerate(samplerate)
```

Sets the global default samplerate for Signal objects, by default 8000 Hz.

**channel(n)**

Get a single data channel.

**Parameters**

- n (int) – channel index

**Returns** a new instance of the class that contains the selected channel as data.

**Return type** (slab.Signal)

```python
channels()
```

Returns generator that yields channel data as objects of the calling class.

```python
resize(duration)
```

Change the duration by padding with zeros or cutting the data.

**Parameters**

- duration (float | int) – new duration of the sound in seconds (given a float) or in samples (given an int).

**Returns** a new instance of the same class with the specified duration.

**Return type** (slab.Signal)

```python
trim(start=0, stop=None)
```

Trim the signal by returning the section between `start` and `stop`. :

- **Parameters**
  - start: start of the section in seconds (given a float) or in samples (given an int).
  - **Returns** a new instance of the same class with the specified duration.

**Return type** (slab.Signal)

```python
resample(samplerate)
```

Resample the sound.

**Parameters**

- samplerate (int) – the samplerate of the resampled sound.

**Returns** a new instance of the same class with the specified samplerate.

**Return type** (slab.Signal)

```python
envelope(apply_envelope=None, times=None, kind='gain')
```

Either apply an envelope to a sound or, if no `apply_envelope` was specified, compute the Hilbert envelope of the sound.

**Parameters**

- apply_envelope (None | numpy.ndarray) – data to multiply with the sound. The envelope is linearly interpolated to be the same length as the sound. If None, compute the sound’s Hilbert envelope

- times (None | numpy.ndarray | list) – If None a vector linearly spaced from 0 to the duration of the sound is used. If time points (in seconds, clamped to the the sound duration) for the amplitude values in envelope are supplied, then the interpolation is piecewise linear between pairs of time and envelope valued (must have same length).

- kind (str) – determines the unit of the envelope value

**Returns**

Either a copy of the instance with the specified envelope applied or the sound’s Hilbert envelope.
**Return type** *(slab.Signal)*

**delay** *(duration=1, channel=0, filter_length=2048)*

Add a delay to one channel.

**Parameters**

- **duration** *(int | float | array-like)* – duration of the delay in seconds (given a float) or samples (given an int). Given an array with the same length as the sound, each sample is delayed by the corresponding number of seconds. This option is used by in *slab.Binaural.itd_ramp*.
- **channel** *(int)* – The index of the channel to add the delay to
- **filter_length** *(int)* – Must be an even number. Determines the accuracy of the reconstruction when using fractional sample delays. Defaults to 2048, or the sound length for shorter signals.

**Returns** a copy of the instance with the specified delay.

**Return type** *(slab.Signal)*

**plot_samples**(show=True, axis=None)*

Stem plot of the samples of the signal.

**Parameters**

- **show** *(bool)* – whether to show the plot right after drawing.
- **axis** *(matplotlib.axes.Axes | None)* – axis to plot to. If None create a new plot.

### Binaural sounds

Binaural sounds inherit from Sound and provide methods for manipulating interaural parameters of two-channel sounds.

**class slab.Binaural(data, samplerate=None)**

Class for working with binaural sounds, including ITD and ILD manipulation. Binaural inherits all sound generation functions from the Sound class, but returns binaural signals. Recasting an object of class sound or sound with 1 or 3+ channels calls Sound.copychannel to return a binaural sound with two channels identical to the first channel of the original sound.

**Parameters**

- **data** *(slab.Signal | numpy.ndarray | list | str)* – see documentation of slab.Sound for details. The data must have either one or two channels. If it has one, that channel is duplicated
- **samplerate** *(int)* – samplerate in Hz, must only be specified when creating an instance from an array.

**.left**

the first data channel, containing the sound for the left ear.

**.right**

the second data channel, containing the sound for the right ear

**.data**

the data-array of the Sound object which has the shape n_samples x n_channels.

**.n_channels**

the number of channels in data. Must be 2 for a binaural sound.
.n_samples
the number of samples in \textit{data}. Equals \textit{duration} \times \textit{samplerate}.

duration
the duration of the sound in seconds. Equals \textit{n_samples} \div \textit{samplerate}.

property left
The left channel for a stereo sound.

property right
The right channel for a stereo sound.

\textit{itd}(\textit{duration}=\text{None}, \textit{max_lag}=0.001)
Either estimate the interaural time difference of the sound or generate a new sound with the specified interaural time difference. The resolution for computing the ITD is \textit{1/samplerate} seconds. A negative ITD value means that the right channel is delayed, meaning the sound source is to the left.

\textbf{Parameters}

- \textbf{\textit{duration} (\text{None} | \text{int} | \text{float})} – Given \text{None}, the instance’s ITD is computed. Given another value, a new sound with the desired interaural time difference in samples (given an integer) or seconds (given a float) is generated.

- \textbf{\textit{max_lag} (\text{float})} – Maximum possible value for ITD estimation. Defaults to 1 millisecond which is barely outside the physiologically plausible range for humans. Is ignored if \textit{duration} is specified.

\textbf{Returns}

The interaural time difference in samples or a copy of the instance with the specified interaural time difference.

\textbf{Return type} (\text{int} | \textsl{slab}.Binaural)

Examples:

```python
sound = \textsl{slab}.Binaural.whitenoise()
lateral = sound.itd(duration=0.0005)  # generate a sound with 0.5 ms ITD
lateral.itd()  # estimate the ITD of the sound
```

\textit{ild}(\textit{dB}=\text{None})
Either estimate the interaural level difference of the sound or generate a new sound with the specified interaural level difference. Negative ILD value means that the left channel is louder than the right channel, meaning that the sound source is to the left. The level difference is achieved by adding half the ILD to one channel and subtracting half from the other channel, so that the mean intensity remains constant.

\textbf{Parameters} \textbf{\textit{dB} (\text{None} | \text{int} | \text{float})} – If \text{None}, estimate the sound’s ITD. Given a value, a new sound is generated with the desired interaural level difference in decibels.

\textbf{Returns} The sound’s interaural level difference, or a new sound with the specified ILD.

\textbf{Return type} (\text{float} | \textsl{slab}.Binaural)

Examples:

```python
sig = \textsl{slab}.Binaural.whitenoise()
lateral_right = sig.ild(3)  # attenuate left channel by 1.5 dB and amplify rightchannel by the same amount
lateral_left = sig.ild(-3)  # vice-versa
```
**itd_ramp** *(from_itd=-0.0006, to_itd=0.0006)*

Generate a sound with a linearly increasing or decreasing interaural time difference. This is achieved by sinc interpolation of one channel with a dynamic delay. The resulting virtual sound source moves left or right.

**Parameters**

- **from_itd** *(float)* – interaural time difference in seconds at the start of the sound. Negative numbers correspond to sources to the left of the listener.
- **to_itd** *(float)* – interaural time difference in seconds at the end of the sound.

**Returns** a copy of the instance with the desired ITD ramp.

**Return type** *(slab.Binaural)*

**Examples**

```python
sig = slab.Binaural.whitenoise()
moving = sig.itd_ramp(from_itd=-0.001, to_itd=0.01)
moving.play()
```

**ild_ramp** *(from_ild=-50, to_ild=50)*

Generate a sound with a linearly increasing or decreasing interaural level difference. The resulting virtual sound source moves to the left or right.

**Parameters**

- **from_ild** *(int | float)* – interaural level difference in decibels at the start of the sound. Negative numbers correspond to sources to the left of the listener.
- **to_ild** *(int | float)* – interaural level difference in decibels at the end of the sound.

**Returns** a copy of the instance with the desired ILD ramp. Any previously existing level difference is removed.

**Return type** *(slab.Binaural)*

**Examples**

```python
sig = slab.Binaural.whitenoise()
moving = sig.ild_ramp(from_ild=-10, to_ild=10)
moving.play()
```

**static azimuth_to_itd** *(azimuth, frequency=2000, head_radius=8.75)*

Compute the ITD for a sound source at a given azimuth. For frequencies >= 2 kHz the Woodworth (1962) formula is used. For frequencies <= 500 Hz the low-frequency approximation mentioned in Aronson and Hartmann (2014) is used. For frequencies in between, we interpolate linearly between the two formulas.

**Parameters**

- **azimuth** *(int | float)* – The azimuth angle of the sound source, negative numbers refer to sources to the left.
- **frequency** *(int | float)* – Frequency in Hz for which the ITD is estimated. Use the default for for sounds with a broadband spectrum.
- **head_radius** *(int | float)* – Radius of the head in centimeters. The bigger the head, the larger the ITD.

**Returns** The interaural time difference for a sound source at a given azimuth.
Return type (float)

Examples:

```python
# compute the ITD for a sound source 90 degrees to the left for a large head
itd = slab.Binaural.azimuth_to_itd(-90, head_radius=10)
```

**static azimuth_to_ild(azimuth, frequency=2000, ils=None)**

Get the interaural level difference corresponding to a sound source at a given azimuth and frequency.

**Parameters**

- **azimuth** (*int | float*) – The azimuth angle of the sound source, negative numbers refer to sources to the left.
- **frequency** (*int | float*) – Frequency in Hz for which the ITD is estimated. Use the default for for sounds with a broadband spectrum.
- **ils** (*dict | None*) – interaural level spectrum from which the ILD is taken. If None, is called. For repeated use (make_interaural_level_spectrum()) – is better to generate and keep the ils in(it) –

**Returns** The interaural level difference for a sound source at a given azimuth in decibels.

Return type (float)

Examples:

```python
ils = slab.Binaural.make_interaural_level_spectrum() # using default KEMAR HRTF
ild = slab.Binaural.azimuth_to_ild(-90, ils=ils) # ILD equivalent to 90 deg.
```

**at_azimuth(azimuth=0, ils=None)**

Convenience function for adding ITD and ILD corresponding to the given azimuth to the sound source. Values are obtained from azimuth_to_itd and azimuth_to_ild. Frequency parameters for these functions are generated from the centroid frequency of the sound.

**Parameters**

- **azimuth** (*int | float*) – The azimuth angle of the sound source, negative numbers refer to sources to the left.
- **ils** (*dict | None*) – interaural level spectrum from which the ILD is taken. If None, is called. For repeated use (make_interaural_level_spectrum()) – is better to generate and keep the ils in(it) –

**Returns** a sound with the appropriate ITD and ILD applied

Return type (*slab.Binaural*)

**externalize(hrtf=None)**

Convolve the sound with a smoothed HRTF to evoke the impression of an external sound source without adding directional information, see Kulkarni & Colburn (1998) for why that works.

**Parameters** **hrtf** (*None | slab.HRTF*) – The HRTF to use. If None use the one from the MIT KEMAR mannequin. The sound source at zero azimuth and elevation is used for convolution so it has to be present in the HRTF.
**Returns**

externalized copy of the instance.

**Return type** *(slab.Binaural)*

### static make_interaural_level_spectrum(hrtf=None)*

Compute the frequency band specific interaural intensity differences for all sound source azimuth’s in a head-related transfer function. For every azimuth in the *hrtf*, the respective transfer function is applied to a sound. This sound is then divided into frequency sub-bands. The interaural level spectrum is the level difference between right and left for each of these sub-bands for each azimuth. When individual HRTFs are not available, the level spectrum of the KEMAR mannequin may be used (default). Note that the computation may take a few minutes. Save the level spectrum to avoid re-computation, for instance with pickle or numpy.save (see documentation on readthedocs for examples).

**Parameters**

- **hrtf** *(None | slab.HRTF)* – The head-related transfer function used to compute the level spectrum. If None, use the recordings from the KEMAR mannequin.

**Returns**

A dictionary with keys `samplerate`, `frequencies [n]`, `azimuths [m]`, and `level_diffs [n x m]`, where `frequencies` lists the centres of sub-bands for which the level difference was computed, and `azimuths` lists the sound source azimuth’s in the *hrtf*. `level_diffs` is a matrix of the interaural level difference for each sub-band and azimuth.

**Return type** *(dict)*

Examples:

```python
ils = slab.Binaural.make_interaural_level_spectrum() # get the ils from the KEMAR recordings
ils['samplerate'] # the sampling rate
ils['frequencies'] # the sub-band frequencies
ils['azimuths'] # the sound source azimuth's for which the level difference was calculated
ils['level_diffs'][5, : ] # the level difference for each azimuth in the 5th sub-band
```

### interaural_level_spectrum(azimuth, ils=None)*

Apply an interaural level spectrum, corresponding to a sound sources azimuth, to a binaural sound. The interaural level spectrum consists of frequency specific interaural level differences which are computed from a head related transfer function (see the `make_interaural_level_spectrum()` method). The binaural sound is divided into frequency sub-bands and the levels of each sub-band are set according to the respective level in the interaural level spectrum. Then, the sub-bands are summed up again into one binaural sound.

**Parameters**

- **azimuth** *(int | float)* – azimuth for which the interaural level spectrum is calculated.
- **ils** *(dict)* – interaural level spectrum to apply. If None, `make_interaural_level_spectrum()` is called.

- **repeated use** *(For)* –

  * is better to generate and keep the ils in a variable to avoid re-computing it. *(it)* –

**Returns**

A binaural sound with the interaural level spectrum corresponding to the given azimuth.

**Return type** *(slab.Binaural)*

Examples:
slab, Release 1.0.1

```python
noise = slab.Binaural.pinknoise(kind='diotic')
ils = slab.Binaural.make_interaural_level_spectrum() # using default KEMAR HRTF
noise.interaural_level_spectrum(azimuth=-45, ils=ils).play()
```

**static whitenoise** *(kind='diotic', **kwargs)*
Generate binaural white noise. *kind*='diotic' produces the same noise samples in both channels, *kind*='dichotic' produces uncorrelated noise. The rest is identical to `slab.Sound.whitenoise`.

**static pinknoise** *(kind='diotic', **kwargs)*
Generate binaural pink noise. *kind*='diotic' produces the same noise samples in both channels, *kind*='dichotic' produces uncorrelated noise. The rest is identical to `slab.Sound.pinknoise`.

**static powerlawnoise** *(kind='diotic', **kwargs)*
Generate binaural power law noise. *kind*='diotic' produces the same noise samples in both channels, *kind*='dichotic' produces uncorrelated noise. The rest is identical to `slab.Sound.powerlawnoise`.

**static irn** *(kind='diotic', **kwargs)*
Generate iterated ripple noise (IRN). *kind*='diotic' produces the same noise samples in both channels, *kind*='dichotic' produces uncorrelated noise. The rest is identical to `slab.Sound.irn`.

**static tone** *(**kwargs)*
Identical to slab.Sound.tone, but with two channels.

**static dynamic_tone** *(**kwargs)*
Identical to slab.Sound.dynamic_tone, but with two channels.

**static harmoniccomplex** *(**kwargs)*
Identical to slab.Sound.harmoniccomplex, but with two channels.

**static click** *(**kwargs)*
Identical to slab.Sound.click, but with two channels.

**static clicktrain** *(**kwargs)*
Identical to slab.Sound.clicktrain, but with two channels.

**static chirp** *(**kwargs)*
Identical to slab.Sound.chirp, but with two channels.

**static silence** *(**kwargs)*
Identical to slab.Sound.silence, but with two channels.

**static vowel** *(**kwargs)*
Identical to slab.Sound.vowel, but with two channels.

**static multitone_masker** *(**kwargs)*
Identical to slab.Sound.multitone_masker, but with two channels.

**static equally_masking_noise** *(**kwargs)*
Identical to slab.Sound.erb_noise, but with two channels.
3.7.2 Psychoacoustic procedures

```python
class slab.Trialsequence(conditions=2, n_reps=1, trials=None, kind=None, deviant_freq=None, label="")
```
Randomized, non-adaptive trial sequences.

Parameters

- **conditions** *(list | int | str)* -- defines the different stimuli appearing the sequence. If given a list, every element is one condition. The elements can be anything - strings, dictionaries, objects etc. Note that, if the elements are not JSON serializable, the sequence can only be saved as a pickle file. If conditions is an integer i, the list of conditions is given by range(i). A string is treated as the filename of a previously saved trial sequence object, which is then loaded.

- **n_reps** *(int)* -- number of repetitions for each condition. Number of trials is given by len(conditions)*n_reps).

- **trials** *(None | list | numpy.ndarray)* -- The sequence of trials in the order in which they are appearing in sequence. Defaults to None, because trials are usually generated by the class based on the other parameters. However, it is possible to pass a list or one-dimensional array. In that case the parameters for generating the sequence are ignored.

- **kind** *(str)* -- The kind of randomization used to generate the trial sequence. Possible options are: `non_repeating` (randomization without direct repetition of a condition, default if n_conditions > 2), `random_permutation` (complete randomization, default if n_conditions <= 2) or `infinite` (sequence that reset when reaching the end to generate an infinite number of trials. randomization method is random_permutation if n_conditions` <= 2 and non_repeating otherwise).

- **deviant_freq** *(float)* -- frequency with which deviants (encoded as 0) appear in the sequence. The minimum number of trials between two deviants is 3 if deviant frequency is below 10%, 2 if it is below 20% and 1 if it is below 30%. A deviant frequency greater than 30% is not supported

- **label** *(str)* -- a text label for the sequence.

- **.trials**
  the order in which the conditions are repeated in the sequence. The elements are integers referring to indices in conditions, starting from 1. 0 represents a deviant (only present if deviant_freq > 0)

- **.n_trials**
  the total number of trials in the sequence

- **.conditions**
  list of the different unique elements in the sequence

- **.n_conditions**
  number of conditions, is equal to len(conditions) or len(conditions)+1 if there are deviants

- **.n_remaining**
  the number of trials remaining i.e. that have not been called when iterating through the sequence

- **.this_n**
  current trials index in the entire sequence, equals the number of trials completed so far

- **.this_trial**
  a dictionary giving the parameters of the current trial

- **.finished**
  boolean signaling if all trials have been called
.kind
randomization kind of sequence (random_permutation, non_repeating, infinite)

.data
list with the same length as the one in the trials attribute. On sequence generation, data is a list of empty lists. Then, one can use the add_response method to append to the list belonging to the current trial

add_response(response)
Append response to the list in the data attribute belonging to the current trial (see Trialsequence doc).

response
data to append to the list. Can be anything but save_json method won’t be available if the content of response is not JSON serializable (if it’s an object for example).

Type any

print_trial_info()
Convenience method for printing current trial information.

get_future_trial(n=1)
Returns the condition of a trial n iterations into the future or past, without advancing the trials.

Parameters n (int) – number of iterations into the future or past (negative numbers).

Returns
element of the list stored in the conditions attribute belonging to the trial n iterations into the past/future. Returns None if attempting to go beyond the first/last trial

Return type (any)

transitions()
Count the number of transitions between conditions.

Returns table of shape n_conditions x n_conditions where the rows represent the condition transitioning from and the columns represent the condition transitioning to. For example [0, 2] shows the number of transitions from condition 1 to condition 3. If the kind of the sequence is “non_repeating”, the diagonal is 0 because no condition transitions into itself.

Return type (numpy.ndarray)

condition_probabilities()
Return the frequency with which each condition appears in the sequence.

Returns

list of floats floats, where every element represents the frequency of one condition. The first element is the frequency of the first condition and so on.

Return type (list)

response_summary()
Generate a summary of the responses for each condition. The function counts how often a specific response was given to a condition for all conditions and each possible response (including None).

Returns indices of the outer list represent the conditions in the sequence. Each inner list contains the number of responses per response key, with the response keys sorted in ascending order, the last element always represents None. If the sequence is not finished yet, None is returned.

Return type (list of lists | None)

Examples:
import slab
import random

sequence = slab.Trialsequence(conditions=3, n_reps=10) # a sequence with three conditions

# iterate through the list and generate a random response. The response can be
# either yes (1), no (0) or
# there can be no response at all (None)
for trial in sequence:
    response = random.choice([0, 1, None])
    sequence.add_response(response)

sequence.response_summary()

# Out: [[1, 1, 7], [2, 5, 3], [4, 4, 2]]
# The first sublist shows that the subject responded to the first condition
# once with no (0),
# once with yes (1) and did not give a response seven times, the second and
# third list show
# prevalence of the same response keys for conditions two and three.

plot(axis=None, show=True)

Plot the trial sequence as scatter plot.

Parameters

- **axis** (matplotlib.pyplot.Axes) – plot axis to draw on, if none a new plot is generated
- **show** (bool) – show the plot immediately, defaults to True

load_json(file_name)

Read JSON file and deserialize the object into self.__dict__.

**file_name**

name of the file to read.

**Type** str | pathlib.Path

load_pickle(file_name)

Read pickle file and deserialize the object into self.__dict__.

**file_name**

name of the file to read.

**Type** str | pathlib.Path

present_afc_trial(target, distractors, key_codes=range(49, 58), isi=0.25, print_info=True)

Present the reference and distractor sounds in random order and acquire a response keypress. The subject
has to identify at which position the reference was played. The result (True if response was correct or False
if response was wrong) is stored in the sequence via the add_response method.

Parameters

- **target** (instance of slab.Sound) – sound that ought to be identified in the trial
- **distractors** (instance or list of slab.Sound) – distractor sound(s)
- **key_codes** (list of int) – ascii codes for the response keys (get code for button ‘1’: ord(‘1’) → 49 pressing the second button in the list is equivalent to the response “the reference was the second sound played in this trial”). Defaults to the key codes for buttons ‘1’ to ‘9’
- **isi** (int or float) – inter stimulus interval which is the pause between the end of one sound and the start
• the next one. (of) –
• print_info (bool) – If true, call the print_trial_info method afterwards

**present_tone_trial**(stimulus, correct_key_idx=0, key_codes=range(49, 58), print_info=True)
Present the reference and distractor sounds in random order and acquire a response keypress. The result (True if response was correct or False if response was wrong) is stored in the sequence via the add_response method.

**Parameters**

- **stimulus** (slab.Sound) – sound played in the trial.
- **correct_key_idx**(int | list of int) – index of the key in key_codes that represents a correct response. Response is correct if response == key_codes[correct_key_idx]. Can be a list of ints if several keys are counted as correct response.
- **key_codes**(list of int) – ascii codes for the response keys (get code for button ‘1’: ord(‘1’) -> 49).
- **print_info** (bool) – If true, call the print_trial_info method afterwards.

**save_json**(file_name=None, clobber=False)
Save the object as JSON file. The object’s __dict__ is serialized and saved in standard JSON format, so that it can be easily reconstituted (see load_json method). Raises FileExistsError if the file exists, unless clobber is True. When file_name in None (default), the method returns the JSON string, in case you want to inspect it. Note that Numpy arrays are not serializable and are converted to Python int. This works because the Trialsequence and Staircase classes use arrays of indices. If your instances of these classes contain arrays of float, use save_pickle instead.

**Parameters**

- **file_name**(str | pathlib.Path) – name of the file to create. If None or ‘stdout’, return a JSON object.
- **clobber** (bool) – overwrite existing file with the same name, defaults to False.

**Returns** True if writing was successful.

**Return type** (bool)

**save_pickle**(file_name, clobber=False)
Save the object as pickle file.

**Parameters**

- **file_name**(str | pathlib.Path) – name of the file to create.
- **clobber** (bool) – overwrite existing file with the same name, defaults to False.

**Returns** True if writing was successful.

**Return type** (bool)

**simulate_response**(threshold=None, transition_width=2, intervals=1, hitrates=None)
Return a simulated response to the current condition index value by calculating the hitrate from a psychometric (logistic) function. This is only sensible if trials is numeric and an interval scale representing a continuous stimulus value.

**Parameters**

- **threshold**(None | int | float) – Midpoint of the psychometric function for adaptive testing. When the intensity of the current trial is equal to the threshold the hitrate is 50 percent.
• **transition_width** *(int / float)* – range of stimulus intensities over which the hitrate increases from 0.25 to 0.75.

• **intervals** *(int)* – use 1 (default) to indicate a yes/no trial, 2 or more to indicate an alternative forced choice trial. The number of choices determines the probability for a correct response by chance.

• **hitrates** *(None / list / numpy.ndarray)* – list or numpy array of hitrates for the different conditions, to allow custom rates instead of simulation. If given, **threshold** and **transition_width** are not used. If a single value is given, this value is used.

```python
class slab.Staircase(start_val, n_reversals=None, step_sizes=1, step_up_factor=1, n_pretrials=0, n_up=1, n_down=2, step_type='lin', min_val=inf, max_val=inf, label='')
```

Class to handle adaptive testing which means smoothly the selecting next trial, report current values and so on. The sequence will terminate after a certain number of reversals have been exceeded.

**Parameters**

• **start_val** *(int / float)* – initial stimulus value for the staircase

• **n_reversals** *(int)* – number of reversals needed to terminate the staircase

• **step_sizes** *(int / list)* – Size of steps in the staircase. Given an integer, the step size is constant. Given a list, the step size will progress to the next entry at each reversal. If the list is exceeded before the sequence was finished, it will continue with the last entry of the list as constant step size.

• **step_up_factor** – allows different sizes for up and down steps to implement a Kaernbach1991 weighted up-down method. step_sizes sets down steps, which are multiplied by step_up_factor to obtain up step sizes. The default is 1, i.e. same size for up and down steps.

• **n_pretrials** *(int)* – number of trial at the initial stimulus value presented as before start of the staircase

• **n_up** *(int)* – number of incorrect (or 0) responses before the staircase level increases. Is 1, regardless of specified value until the first reversal. Lewitt (1971) gives the up-down values for different threshold points on the psychometric function: 1-1 (0.5), 1-2 (0.707), 1-3 (0.794), 1-4 (0.841), 1-5 (0.891).

• **n_down** *(int)* – number of correct (or 1) responses before the staircase level decreases (see **n_up**).

• **step_type** *(str)* – defines the change of stimulus intensity at each step of the staircase. Possible inputs are ‘lin’ (adds or subtract a certain amount), ‘db’, and ‘log’ (prevents the intensity from reaching zero).

• **min_val** *(int or float)* – smallest stimulus value permitted, or -Inf for staircase without lower limit

• **max_val** *(int or float)* – largest stimulus value permitted, or Inf for staircase without upper limit

• **label** *(str)* – text label for the sequence, defaults to an empty string

```python
.this_trial_n
    number of completed trials

.intensities
    presented stimulus values

.current_direction
    ‘up’ or ‘down’
```
.data
    list of responses
.reversal_points
    indices of reversal trials
.reversal_intensities
    stimulus values at the reversals (used to compute threshold)
.finished
    True/False: have we finished yet?

Examples:

```python
def stairs = Staircase(start_val=50, n_reversals=10, step_type='lin',
                       step_sizes=[4,2], min_val=10, max_val=60, n_up=1, n_down=1)
print(stairs)
for trial in stairs:
    response = stairs.simulate_response(30)
    stairs.add_response(response)
print(f'reversals: {stairs.reversal_intensities}
print(f'mean of final 6 reversals: {stairs.threshold()}
```

.add_response(result, intensity=None)
    Add a True or 1 to indicate a correct/detected trial or False or 0 to indicate an incorrect/missed trial. This is essential to advance the staircase to a new intensity level. Supplying an intensity value indicates that you did not use the recommended intensity in your last trial and the staircase will replace its recorded value with the one supplied.

calculate_next_intensity()
    Based on current intensity, counter of correct responses, and current direction.

.threshold(n=0)
    Returns the average of the last n reversals.

Parameters
    n (int) -- number of reversals to average over, if 0 use n_reversals - 1.

Returns
    the arithmetic (if step_type 'lin') or geometric mean of the reversal_intensities.

.print_trial_info()
    Convenience method for printing current trial information.

.save_csv(filename)
    Write a csv text file with the stimulus values in the 1st line and the corresponding responses in the 2nd.

Parameters
    filename (str) -- the name under which the csv file is saved.

Returns
    True if saving was successful, False if there are no trials to save.

Return type
    (bool)

.plot(axis=None, show=True)
    Plot the staircase. If called after each trial, one plot is created and updated.

Parameters
    axis (matplotlib.pyplot.Axes) -- plot axis to draw on, if none a new plot is generated
    show (bool) -- whether to show the plot right after drawing.

.static close_plot()
    Closes a staircase plot (if not drawn into a specified axis) - used for plotting after each trial.
**load_json**(*file_name*)
Read JSON file and deserialize the object into `self.__dict__`.

- `file_name`
  - name of the file to read.
  - **Type**: str | pathlib.Path

**load_pickle**(*file_name*)
Read pickle file and deserialize the object into `self.__dict__`.

- `file_name`
  - name of the file to read.
  - **Type**: str | pathlib.Path

**presentafc_trial**(*target*, *distractors*, *key_codes=range(49, 58)*, *isi=0.25*, *print_info=True*)
Present the reference and distractor sounds in random order and acquire a response keypress. The subject has to identify at which position the reference was played. The result (True if response was correct or False if response was wrong) is stored in the sequence via the `add_response` method.

**Parameters**
- `target` (*instance of slab.Sound*) – sound that ought to be identified in the trial
- `distractors` (*instance or list of slab.Sound*) – distractor sound(s)
- `key_codes` (*list of int*) – ascii codes for the response keys (get code for button ‘1’: ord(‘1’) → 49) pressing the second button in the list is equivalent to the response “the reference was the second sound played in this trial”. Defaults to the key codes for buttons ‘1’ to ‘9’
- `isi` (*int or float*) – inter stimulus interval which is the pause between the end of one sound and the start
- `the next one, (of) –`
- `print_info` (*bool*) – If true, call the `print_trial_info` method afterwards

**presenttone_trial**(*stimulus*, *correct_key_idx=0*, *key_codes=range(49, 58)*, *print_info=True*)
Present the reference and distractor sounds in random order and acquire a response keypress. The result (True if response was correct or False if response was wrong) is stored in the sequence via the `add_response` method.

**Parameters**
- `stimulus` (*slab.Sound*) – sound played in the trial.
- `correct_key_idx` (*int | list of int*) – index of the key in `key_codes` that represents a correct response. Response is correct if `response == key_codes[correct_key_idx]`. Can be a list of ints if several keys are counted as correct response.
- `key_codes` (*list of int*) – ascii codes for the response keys (get code for button ‘1’: ord(‘1’) → 49).
- `print_info` (*bool*) – If true, call the `print_trial_info` method afterwards.

**save_json**(*file_name=None*, *clobber=False*)
Save the object as JSON file. The object’s `__dict__` is serialized and saved in standard JSON format, so that it can be easily reconstituted (see `load_json` method). Raises FileExistsError if the file exists, unless `clobber` is True. When `file_name` in None (default), the method returns the JSON string, in case you want to inspect it. Note that Numpy arrays are not serializable and are converted to Python int. This works because the Trialssequence and Staircase classes use arrays of indices. If your instances of these classes contain arrays of float, use `save_pickle` instead.
Parameters

- **file_name** (*str | pathlib.Path*) – name of the file to create. If None or ‘stdout’, return a JSON object.
- **clobber** (*bool*) – overwrite existing file with the same name, defaults to False.

Returns

True if writing was successful.

Return type *(bool)*

**save_pickle**(*file_name*, **clobber**=False)

Save the object as pickle file.

Parameters

- **file_name** (*str | pathlib.Path*) – name of the file to create.
- **clobber** (*bool*) – overwrite existing file with the same name, defaults to False.

Returns

True if writing was successful.

Return type *(bool)*

**simulate_response**(*threshold*=None, **transition_width**=2, **intervals**=1, **hitrates**=None)

Return a simulated response to the current condition index value by calculating the hitrate from a psychometric (logistic) function. This is only sensible if trials is numeric and an interval scale representing a continuous stimulus value.

Parameters

- **threshold** (*None | int | float*) – Midpoint of the psychometric function for adaptive testing. When the intensity of the current trial is equal to the *threshold* the hitrate is 50 percent.
- **transition_width** (*int | float*) – range of stimulus intensities over which the hitrate increases from 0.25 to 0.75.
- **intervals** (*int*) – use 1 (default) to indicate a yes/no trial, 2 or more to indicate an alternative forced choice trial. The number of choices determines the probability for a correct response by chance.
- **hitrates** (*None | list | numpy.ndarray*) – list or numpy array of hitrates for the different conditions, to allow custom rates instead of simulation. If given, *threshold* and *transition_width* are not used. If a single value is given, this value is used.

**class slab.Precomputed**(*sounds*, **n**=10)

This class is a list of pre-computed sound stimuli which simplifies their generation and presentation. It is typically used when stimulus generation takes too long to happen in each trial. In this case, a list of stimuli is precomputed and a random stimulus from the list is presented in each trial, ideally without direct repetition. The Precomputed list has a play method which automatically selects an element other than the previous one for playing, and can be used like an *slab.Sound* object.

Parameters

- **sounds** (*list | callable | iterator*) – sequence of Sound objects (each must have a play method).
- **n** – only used if sounds is a callable, calls it n times to make the stimuli.

**.sequence**

a list of all the elements that have been played already.

Examples:
stims = slab.Precomputed(sound_list)  # using a pre-made list
# using a lambda function to make 10 examples of pink noise
stims = slab.Precomputed(lambda: slab.Sound.pinknoise(), n=10)
stims = slab.Precomputed((slab.Sound.vowel(vowel=v) for v in ['a', 'e', 'i']))  # using a generator
stims.play()  # playing a sound from the list

```
play()

Play a random, but never the previous, stimulus from the list.

random_choice(n=1)

Pick (without replacement) random elements from the list.

Parameters n (int) – number of elements to pick.

Returns list of n random elements.

Return type (list)

write(filename)

Save the Precomputed object as a zip file containing all sounds as wav files.

Parameters filename (str | pathlib.Path) – full path to under which the file is saved.

static read(filename)

Read a zip file containing wav files.

Parameters filename (str | pathlib.Path) – full path to the file to be read.

Returns the file content.

Return type (slab.Precomputed)

class slab.ResultsFile(subject='test', folder=None)

A class for simplifying the typical use cases of results files, including generating the name, creating the folders, and writing to the file after each trial. Writes a JSON Lines file, in which each line is a valid self-contained JSON string (see http://jsonlines.org).

Parameters

• subject (str) – determines the name of the sub-folder and files.

• folder (None | str) – folder in which all results are saved, if None use the global variable results_folder.

.path

full path to the results file.

.subject

the subject’s name.

Example:

```python
ResultsFile.results_folder = 'MyResults'
file = ResultsFile(subject='MS')
print(file.name)
```

property name

The name of the results file.

write(data, tag=None)

 Safely write data to the file which is opened just before writing and closed immediately after to avoid data loss. Call this method at the end of each trial to save the response and trial state.
Parameters

- **data** *(any)* – data to save must be JSON serializable [string, list, dict, ...]. If data is an object, the __dict__ is extracted and saved.

- **tag** *(str)* – The tag is prepended as a key. If None is provided, the current time is used.

**static read_file(filename, tag=None)**
Read a results file and return the content.

**Parameters**

- **filename** *(str | pathlib.Path)* –
- **tag** *(None | str)* –

**Returns**

The content of the file. If tag is None, the whole file is returned, else only the dictionaries with that tag as a key are returned. The content will be a list of dictionaries or a dictionary if there is only a single element.

**Return type** *(list | dict)*

**read(tag=None)**
Wrapper for the read_file method.

**static previous_file(subject=None)**
Returns the name of the most recently used results file for a given subject. Intended for extracting information from a previous file when running partial experiments.

**Parameters** **subject** *(str)* – the subject name name under which the file is stored.

**Returns** full path to the most recent results file.

**Return type** *(pathlib.Path)*

**clear()**
Clears the file by erasing all content.

**psychoacoustics.key()**
Wrapper for curses module to simplify getting a single keypress from the terminal (default), a buttonbox, or a figure. Set slab.psychoacoustics.input_method = ‘buttonbox’ to use a custom USB buttonbox or to ‘figure’ to open a figure called ‘stairs’ (if not already opened by the slab.Staricase.plot method). Optionally takes a string argument which is printed in the terminal for conveying instructions to the participant.

Example:

```python
with slab.key('Waiting for buttons 1 (yes) or 2 (no).') as key:
    response = key.getch()
```

**psychoacoustics.load_config()**
Reads a text file with variable assignments. This is a simple convenience method that allows easy writing and loading of configuration text files. Experiments sometimes use configuration files when experimenters (who might not by Python programmers) need to set parameters without changing the code. The format is a plain text file with a variable assignment on each line, because it is meant to be written and changed by humans. These variables and their values are then accessible as a namedtuple.

**Parameters** **filename** *(str | pathlib.Path)* – path to the file to be read.

**Returns** a tuple containing the variables and values defined in the text file.

**Return type** *(collections.namedtuple)*

Example:
# assuming there is a file named 'example.txt' with the following content:
samplerate = 32000
pause_duration = 30
speeds = [60, 120, 180]
# call load_config to parse the file into a named tuple:
conf = load_config('example.txt')
conf.speeds
# Out: [60, 120, 180]

Filters

class slab.Filter(data, samplerate=None, fir=True)
Class for generating and manipulating filter banks and transfer functions. Filters can be either finite impulse response (FIR) or Fourier filters.

Parameters

- **data** (numpy.ndarray | slab.Signal | list) – samples of the filter. If it is an array, the first dimension should represent the number of samples and the second one the number of channels. If it's an object, it must have a .data attribute containing an array. If it's a list, the elements can be arrays or objects. The output will be a multi-channel sound with each channel corresponding to an element of the list.
- **samplerate** (int | None) – the samplerate of the sound. If None, use the default samplerate.
- **fir** (bool) – whether this is a finite impulse filter (True) or a Fourier filter (False).

.n_filters
number of filters in the object (overloads n_channels attribute in the parent Signal class)

.n_taps
the number of taps in a finite impulse response filter. Analogous to n_samples in a Signal.

.n_frequencies
the number of frequency bins in a Fourier filter. Analogous to n_samples in a Signal.

.frequencies
the frequency axis of a Fourier filter.

property n_filters
The number of filters in the bank.

property n_taps
The number of filter taps.

property n_frequencies
The number of frequency bins.

property frequencies
The frequency axis of the filter.

static band(kind='hp', frequency=100, gain=None, samplerate=None, length=1000, fir=True)
Generate simple passband or stopband filters, or filters with a transfer function defined by pairs of frequency and gain values.

Parameters

- **kind** – The type of filter to generate. Can be ‘lp’ (lowpass), ‘hp’ (highpass), ‘bp’ (bandpass) or ‘bs’ (bandstop/notch). If gain is specified the kind argument is ignored.
• **frequency** (int | float | tuple | list) – For a low- or highpass filter, a single integer or float value must be given which is the filters edge frequency in Hz. A bandpass or -stop filter takes a tuple of two values which are the filters lower and upper edge frequencies. Given a list of values and a list of equal length as gain the resulting filter will have the specified gain at each frequency.

• **gain** (None | list) – Must be None when generating an lowpass, highpass, bandpass or bandstop filter. For generating a custom filter, define a list of the same length as frequency with values between 1.0 (no suppression at that frequency) and 0.0 (maximal suppression at that frequency).

• **samplerate** (int | None) – the samplerate of the sound. If None, use the default samplerate.

• **length** (int) – The number of samples in the filter

• **fir** – If true generate a finite impulse response filter, else generate a Fourier filter.

**Returns** a filter with the specified properties

**Return type** (slab.Filter)

Examples:

```python
filt = slab.Filter.band(frequency=3000, kind='lp')  # lowpass filter
filt = slab.Filter.band(frequency=(100, 2000), kind='bs')  # bandstop filter
filt = slab.Filter.band(frequency=[100, 1000, 3000, 6000], gain=[0., 1., 0., 1.])  # custom filter
```

**apply**(sig)

Apply the filter to a sound. If sound and filter have the same number of channels, each filter channel will be applied to the corresponding channel in the sound. If the filter has multiple channels and the sound only 1, each filter is applied to the same sound. In that case the filtered sound will contain the same number of channels as the filter with every channel being a copy of the original sound with one filter channel applied. If the filter has only one channel and the sound has multiple channels, the same filter is applied to each sound channel.

**Parameters** sig (slab.Signal | slab.Sound) – The sound to be filtered.

**Returns** a filtered copy of sig.

**Return type** (slab.Signal | slab.Sound)

Examples:

```python
filt = slab.Filter.band(frequency=(100, 1500), kind='bp')  # bandpass filter
sound = slab.Sound.whitenoise()  # generate sound
filtered_sound = filt.apply(sound)  # apply the filter to the sound
```

**tf**(channels='all', n_bins=None, show=True, axis=None)

Compute a filter’s transfer function (magnitude over frequency) and optionally plot it.

**Parameters**

• **channels** (str | list | int) – the filter channels to compute the transfer function for. Defaults to the string “all” which includes all channels. To compute the transfer function for multiple channels, pass a list of channel integers. For the transfer function for a single channel pass it’s index as integer.

• **n_bins** (None) – number of bins in the transfer function (determines frequency resolution). If None, use the maximum number of bins.
• `show (bool)` – whether to show the plot right after drawing.
• `axis (matplotlib.axes.Axes | None)` – axis to plot to. If None create a new plot.

**Returns** the frequency bins in the range from 0 Hz to the Nyquist frequency. (numpy.ndarray: the magnitude of each frequency in \( w \). None: If `show` is True OR and `axis` was specified, a plot is drawn and nothing is returned.

**Return type** (numpy.ndarray)

**Examples:**
```python
filt = slab.Filter.band(frequency=(100, 1500), kind='bp')  # bandpass filter
filt.tf(show=True)  # compute and plot the transfer functions
w, h = filt.tf(show=False)  # compute and return the transfer functions
```

**static cos_filterbank**(length=5000, bandwidth=0.3333333333333333, low_cutoff=0, high_cutoff=None, pass_bands=False, samplerate=None)

Generate a set of Fourier filters. Each filter’s transfer function is given by the positive phase of a cosine wave. The amplitude of the cosine is that filters central frequency. Following the organization of the cochlea, the width of the filter increases in proportion to it’s center frequency. This increase is defined by Moore & Glasberg’s formula for the equivalent rectangular bandwidth (ERB) of auditory filters. This functions is used for example to divide a sound into bands for equalization.

**length**

The number of bins in each filter, determines the frequency resolution.

  **Type** int

**bandwidth**

Width of the sub-filters in octaves. The smaller the bandwidth, the more filters will be generated.

  **Type** float

**low_cutoff**

The lower limit of frequency range in Hz.

  **Type** int | float

**high_cutoff**

The upper limit of frequency range in Hz. If None, use the Nyquist frequency.

  **Type** int | float

**pass_bands**

Whether to include a half cosine at the filter bank’s lower and upper edge frequency. If True, allows reconstruction of original bandwidth when collapsing subbands.

  **Type** bool

**samplerate**

the samplerate of the sound that the filter shall be applied to. If None, use the default samplerate.s

  **Type** int | None

**Examples:**
```python
sig = slab.Sound.pinknoise(samplerate=44100)
fbank = slab.Filter.cos_filterbank(length=sig.n_samples, bandwidth=1/10, low_cutoff=100, samplerate=sig.samplerate)

fbank.tf()
# apply the filter bank to the data. The filtered sound will contain as many channels as there are filters in the bank. Every channel is a copy of the original sound with one filter applied.
```

(continues on next page)
In this context, the channels are the signals sub-bands:

```python
sig_filt = fbank.apply(sig)
```

**static collapse_subbands** *(subbands, filter_bank=None)*

Sum a sound that has been filtered with a filterbank and which channels represent the sub-bands of the original sound. For each sound channel, the fourier transform is calculated and the result is multiplied with the corresponding filter in the filter bank. For the resulting spectrum, an inverse fourier transform is performed. The resulting sound is summed over all channels.

**Parameters**

- **subbands** *(slab.Signal)* – The sound which is divided into subbands by filtering. The number of channels in the sound must be equal to the number of filters in the filter bank.

- **filter_bank** *(None | slab.Filter)* – The filter bank applied to the sound’s subbands. The number of filters must be equal to the number of channels in the sound. If None a filter bank with the default parameters is generated. Note that the filters must have a number of frequency bins equal to the number of samples in the sound.

**Returns** A sound generated from summing the spectra of the subbands.

**Return type** *(slab.Signal)*

**Examples:**

```python
sig = slab.Sound.whitenoise()  # generate a sound
fbank = slab.Filter.cos_filterbank(length=sig.n_samples)  # generate a filter bank
subbands = fbank.apply(sig)  # divide the sound into subbands by applying the filter
# by collapsing the subbands, a new sound is generated that is (almost) equal to the original sound:
collapsed = fbank.collapse_subbands(subbands, fbank)
```

**filter_bank_center_freqs()**

Get the maximum of each filter in a filter bank. For filter banks generated with the `cos_filterbank` method this corresponds to the filters center frequency.

**Returns**

array with length equal to the number of filters in the bank, containing each filter’s center frequency.

**Return type** *(numpy.ndarray)*

**static equalizing_filterbank** *(reference, sound, length=1000, bandwidth=0.125, low_cutoff=200, high_cutoff=None, alpha=1.0)*

Generate an equalizing filter from the spectral difference between a `sound` and a `reference`. Both are divided into sub-bands using the `cos_filterbank` and the level difference per sub-band is calculated. The sub-band frequencies and level differences are then used to generate an equalizing filter that makes the spectrum of the `sound` more equal to the one of the `reference`. The main use case is equalizing the differences between transfer functions of individual loudspeakers.

**Parameters**

- **reference** *(slab.Sound)* – The reference for equalization, i.e. what the sound should look like after applying the equalization. Must have exactly one channel.

- **sound** *(slab.Sound)* – The sound to equalize. Can have multiple channels.
• **length** *(int)* – Number of frequency bins in the filter.

• **bandwidth** *(float)* – Width of the filters, used to divide the signal into subbands, in octaves. A small bandwidth results in a fine tuned transfer function which is useful for equalizing small notches.

• **low_cutoff** *(int | float)* – The lower limit of frequency range in Hz.

• **high_cutoff** *(int | float)* – The upper limit of frequency range in Hz. If None, use the Nyquist frequency.

• **alpha** *(float)* – Filter regularization parameter. Values below 1.0 reduce the filter’s effect, values above amplify it. WARNING: large filter gains may result in temporal distortions of the sound.

**Returns**

An equalizing filter bank with a number of filters equal to the number of channels in the equalized sound.

**Return type** *(slab.Filter)*

**Example**

```python
# generate a sound and apply some arbitrary filter to it
sound = slab.Sound.pinknoise(samplerate=44100)
 filt = slab.Filter.band(frequency=[100., 800., 2000., 4300., 8000., 14500., 18000.], gain=[0., 1., 0., 1., 0., 1., 0.], samplerate=sound.
samplerate)
 filtered = filt.apply(sound)

# make an equalizing filter and apply it to the filtered signal. The result looks more like the original
fbank = slab.Filter.equalizing_filterbank(sound, filtered, low_cutoff=200, high_cutoff=16000)
 equalized = fbank.apply(sound)
```

**save**(filename)

Save the filter in Numpy’s `.npy` format to a file. :param filename: Full path to which the data is saved. :type filename: str | pathlib.Path

**static load**(filename)

Load a filter from a `.npy` file.

**Parameters** filename *(str | pathlib.Path)* – Full path to the file to load.

**Returns** The `Filter` loaded from the file.

**Return type** *(slab.Filter)*
HRTFs

class slab.HRTF(data, samplerate=None, sources=None, listener=None, verbose=False)
Class for reading and manipulating head-related transfer functions with attributes and functions to manage them.

Parameters

- **data** *(str | Filter | numpy.ndarray)* – Typically, this is the path to a file in the .sofa format. The file is then loaded and the data of each source for which the transfer function was recorded is stored as a Filter object in the `data` attribute. Instead of a file name, the data can be passed directly as Filter or numpy array. Given a Filter, every filter channel in the instance is taken as a source (this does not result in a typical HRTF object and is only intended for equalization filter banks). Given an 3D array, the first dimension represents the sources, the second the number of taps per filter and the last the number of filter channels per filter (should be always 2, for left and right ear).

- **samplerate** *(None | float)* – rate at which the data was acquired, only relevant when not loading from .sofa file

- **sources** *(None | array)* – positions of the recorded sources, only relevant when not loading from .sofa file

- **listener** *(None | list | dict)* – position of the listener, only relevant when not loading from .sofa file

- **verbose** *(bool)* – print out items when loading .sofa files, defaults to False

.n_sources
The number of sources in the HRTF.

Type int

.sources
spherical coordinates (azimuth, elevation, distance) of all sources.

Type array

.n_elevations
The number of elevations in the HRTF.

Type int

.data
The HRTF data. The elements of the list are instances of slab.Filter.

Type list

.listener
a dictionary containing the position of the listener (“pos”), the point which the listener is fixating (“view”), the point 90° above the listener (“up”) and vectors from the listener to those points.

Type dict

.samplerate
sampling rate at which the HRTF data was acquired.

Type float

Example:

```python
import slab
hrtf = slab.HRTF.kemar() # use inbuilt KEMAR data
sourceidx = hrtf.cone_sources(20)
```
property n_sources
    The number of sources in the HRTF.

property n_elevations
    The number of elevations in the HRTF.

apply(source, sound, allow_resampling=True)
    Apply a filter from the HRTF set to a sound. The sound will be recast as slab.Binaural. If the samplerates
    of the sound and the HRTF are unequal and allow_resampling is True, then the sound will be resampled to
    the filter rate, filtered, and then resampled to the original rate. The filtering is done with scipy.signal.ffcconvolve.

Parameters
    • source (int) – the source index of the binaural filter in self.data.
    • sound (slab.Signal | slab.Sound | slab.Binaural) – the sound to be rendered spatially.

Returns a spatialized copy of sound.

Return type (slab.Binaural)

elevations()
    Get all different elevations at which sources were recorded. Note: This currently only works as intended for
    HRTFs recorded in horizontal rings.

Returns a sorted list of source elevations.

Return type (list)

plot_tf(sourceidx, ear='left', xlim=(1000, 18000), n_bins=None, kind='waterfall', linesep=20,
        xscale='linear', show=True, axis=None)
    Plot transfer functions of FIR filters at a list of source indices.

Parameters
    • ear (str) – the ear from which data is plotted. Can be ‘left’, ‘right’, or ‘both’.
    • sourceidx (list of int) – sources to plot. Typically be generated using the
        hrtf.cone_sources Method.
    • xlim (tuple of int) – frequency range of the plot
    • n_bins (int) – passed to slab.Filter.tf() and determines frequency resolution
    • kind (str) – type of plot to draw. Can be waterfall (as in Wightman and Kistler, 1989) or image (as in Hofman 1998).
    • linesep (int) – vertical distance between transfer functions in the waterfall plot
    • xscale (str) – sets x-axis scaling (‘linear’, ‘log’)
    • show (bool) – If True, show the plot immediately
    • axis (matplotlib.axes._subplots.AxesSubplot) – Axis to draw the plot on

diffuse_field_avg()
    Compute the diffuse field average transfer function, i.e. the constant non-spatial portion of a set of HRTFs. The
    filters for all sources are averaged, which yields an unbiased average only if the sources are uniformly distributed
    around the head.
Returns the diffuse field average as FFR filter object.

Return type (*Filter*)

diffuse_field_equalization(*dfa=None*)

Equalize the HRTF by dividing each filter by the diffuse field average. The resulting filters have a mean close to 0 and are Fourier filters.

Parameters *dfa (None)* – Filter object containing the diffuse field average transfer function of the HRTF. If none is provided, the *diffuse_field_avg* method is called to obtain it.

Returns diffuse field equalized version of the HRTF.

Return type (*HRTF*)

cone_sources(*cone=0*)

Get all sources of the HRTF that lie on a “cone of confusion”. The cone is a vertical off-axis sphere slice. All sources that lie on the cone have the same interaural level and time difference. Note: This currently only works as intended for HRTFs recorded in horizontal rings.

Parameters *cone (int | float)* – azimuth of the cone center in degree.

Returns elements of the list are the indices of sound sources on the frontal half of the cone.

Return type (*list*)

Examples:

```python
import HRTF
hrtf = slab.HRTF.kemar()
sourceidx = hrtf.cone_sources(20)  # get the source indices
print(hrtf.sources[sourceidx])  # print the coordinates of the source indices
hrtf.plot_sources(sourceidx)  # show the sources in a 3D plot
```

elevation_sources(*elevation=0*)

Get the indices of sources along a horizontal sphere slice at the given elevation.

Parameters *elevation (int | float)* – The elevation of the sources in degree. The default returns sources along the frontal horizon.

Returns

indices of the sound sources. If the hrtf does not contain the specified elevation an empty list is returned.

Return type (*list*)

tfs_from_sources(*sources, n_bins=96*)

Get the transfer function from sources in the hrtf.

Parameters

- *sources (list)* – Indices of the sources (as generated for instance with the *HRTF.cone_sources* method), for which the transfer function is extracted.
- *n_bins (int)* – The number of frequency bins for each transfer function

Returns

2-dimensional array where the first dimension represents the frequency bins and the second dimension represents the sources.

Return type (*numpy.ndarray*)
interpolate(azimuth=0, elevation=0, method='nearest', plot_tri=False)
Interpolate a filter at a given azimuth and elevation from the neighboring HRTFs. A weighted average of the 3 closest HRTFs in the set is computed in the spectral domain with barycentric weights. The resulting filter values vary smoothly with changes in azimuth and elevation. The fidelity of the interpolated filter decreases with increasing distance of the closest sources and should only be regarded as appropriate approximation when the contributing filters are less than 20 away.

Parameters

• azimuth (float) – the azimuth component of the direction of the interpolated filter

• elevation (float) – the elevation component of the direction of the interpolated filter

• method (str) – interpolation method, ‘nearest’ returns the filter of the nearest direction. Any other string returns a barycentric interpolation.

• plot_tri (bool) – plot the triangulation of source positions used of interpolation. Useful for checking for areas where the interpolation may not be accurate (look for irregular or elongated triangles).

Returns an HRTF object with a single source

Return type (slab.HRTF)

cartesian_source_locations(coordinates=None)
Convert spherical coordinates of source locations in an HRTF object into cartesian coordinates useful for plotting and distance calculations. If you supply a list or array of coordinates, then those are converted.

Parameters coordinates (None | numpy.ndarray) – source locations in spherical coordinates. If None use the object’s coordinate array (self.sources).

Returns the source locations in cartesian coordinates as (n x 3) array

Return type (numpy.ndarray)

vsi(sources=None, equalize=True)
Compute the “vertical spectral information” which is a measure of the dissimilarity of spectral profiles at different elevations. The vsi relates to behavioral localization accuracy in the vertical dimension (Trapeau and Schönwiesner, 2016). It is computed as one minus the average of the correlation coefficients between all combinations of directional transfer functions of the specified sources. A set of identical transfer functions results in a vsi of 0 whereas highly different transfer functions will result in a high VSI (empirical maximum is ~1.07, KEMAR has a VSI of 0.82).

Parameters

• sources (None | list) – indices of sources for which to compute the VSI. If None use the vertical midline.

• equalize (bool) – If True, apply the diffuse_field_equalization method (set to False if the hrtf object is already diffuse-field equalized).

Returns the vertical spectral information between the specified sources.

Return type (float)

plot_sources(idx=None, show=True, label=False, axis=None)
Plot source locations in 3D.

Parameters

• idx (list of int) – indices to highlight in the plot

• show (bool) – whether to show plot (set to False if plotting into an axis and you want to add other elements)
- **label** (*bool*) – if True, show the index of each source in self.sources as text label, if idx is also given, then only these sources are labeled
- **axis** (*mpl_toolkits.mplot3d.axes3d.Axes3D*) – axis to draw the plot on

```python
static kemar()
```
Provides HRTF data from the KEMAR recording (normal pinna) conducted by Gardner and Martin at MIT in 1994 (MIT Media Lab Perceptual Computing - Technical Report #280) and converted to the SOFA Format. Slab includes a compressed copy of the data. This function reads it and returns the corresponding HRTF object. The objects is cached in the class variable `_kemar` and repeated calls return the cached object instead of reading the file from disk again.

- **Returns** the KEMAR HRTF data.
- **Return type** (*slab.HRTF*)

Index of functions and classes: genindex

Search the documentation: search
INDEX

A
add_response() (slab.Staircase method), 64
add_response() (slab.Trialsequence method), 60
am() (slab.Sound method), 46
apply() (slab.Filter method), 70
apply() (slab.HRTF method), 75
apply_to_path() (slab.sound method), 50
at_azimuth() (slab.Binaural method), 56
aweight() (slab.Sound method), 47
azimuth_to_ild() (slab.Binaural static method), 56
azimuth_to_itd() (slab.Binaural static method), 55

B
band() (slab.Filter static method), 69
bandwidth (slab.Filter attribute), 71
Binaural (class in slab), 53

C
calculate_next_intensity() (slab.Staircase method), 64
cartesian_source_locations() (slab.HRTF method), 77
channel() (slab.Signal method), 52
channels() (slab.Signal method), 52
chirp() (slab.Binaural static method), 58
chirp() (slab.Sound static method), 43
clear() (slab.ResultsFile method), 68
click() (slab.Binaural static method), 58
click() (slab.Sound static method), 42
clicktrain() (slab.Binaural static method), 58
clicktrain() (slab.Signal method), 52
close_plot() (slab.Staircase static method), 43
cochleagram() (slab.Sound method), 48
collapse_subbands() (slab.Filter static method), 72
condition_probabilities() (slab.Trialsequence method), 60
conditions (slab.Trialsequence attribute), 59
cone_sources() (slab.HRTF method), 76
cos_filterbank() (slab.Filter static method), 71
crest_factor() (slab.Sound method), 49
crossfade() (slab.Sound static method), 46
current_direction (slab.Staircase attribute), 63

data (slab.Binaural attribute), 53
data (slab.Sound attribute), 38
data (slab.Staircase attribute), 63
data (slab.Trialsequence attribute), 60
delay() (slab.Signal method), 53
diffuse_field_avg() (slab.HRTF method), 75
diffuse_field_equalization() (slab.HRTF method), 76
duration (slab.Binaural attribute), 54
duration (slab.Signal property), 51
duration (slab.Signal attribute), 51
duration (slab.Sound attribute), 39
dynamic_tone() (slab.Binaural static method), 58
dynamic_tone() (slab.Sound static method), 39
elevation_sources() (slab.HRTF method), 76
elevations() (slab.HRTF method), 75
equally_masking_noise() (slab.Binaural static method), 58
equally_masking_noise() (slab.Sound static method), 45
equalizing_filterbank() (slab.Filter static method), 72
externalize() (slab.Binaural method), 56
file_name (slab.Staircase attribute), 65
file_name (slab.Trialsequence attribute), 61
Filter (class in slab), 69
filter() (slab.Sound method), 47
filter_bank_center_freqs() (slab.Filter method), 72
finished (slab.Staircase attribute), 64
finished (slab.Trialsequence attribute), 59
frames() (slab.Sound method), 49
frametimes() (slab.Sound method), 50
frequencies (slab.Filter property), 69
frequencies (slab.Filter attribute), 69
read() (slab.Precomputed static method), 67
read() (slab.ResultsFile method), 68
read() (slab.Sound static method), 39
read_file() (slab.ResultsFile static method), 68
record() (slab.Sound static method), 47
repeat() (slab.Sound method), 46
resample() (slab.Signal method), 52
resize() (slab.Signal method), 52
response (slab.Trialsequence attribute), 60
response_summary() (slab.Trialsequence method), 60
ResultsFile (class in slab), 67
reversal_intensities (slab.Staircase. attribute), 64
reversal_points (slab.Staircase. attribute), 64
right (slab.Binaural property), 54
right (slab.Binaural. attribute), 53
S
samplerate (slab.Filter attribute), 71
samplerate (slab.HRTF. attribute), 74
save() (slab.Filter method), 73
save_csv() (slab.Staircase method), 64
save_json() (slab.Staircase method), 65
save_json() (slab.Trialsequence method), 62
save_pickle() (slab.Staircase method), 66
save_pickle() (slab.Trialsequence method), 62
sequence (slab.Precomputed. attribute), 66
sequence() (slab.Sound static method), 45
set_default_samplerate() (slab.Signal static method), 52
Signal (class in slab), 51
silence() (slab.Binaural static method), 58
silence() (slab.Sound static method), 43
simulate_response() (slab.Staircase method), 66
simulate_response() (slab.Trialsequence method), 62
Sound (class in slab), 38
sources (slab.HRTF. attribute), 74
spectral_feature() (slab.Sound method), 49
spectrogram() (slab.Sound method), 49
spectrum() (slab.Sound method), 48
Staircase (class in slab), 63
subject (slab.ResultsFile. attribute), 67
T
tf() (slab.Filter method), 70
tfs_from_sources() (slab.HRTF method), 76
this_n (slab.Trialsequence. attribute), 59
this_trial (slab.Trialsequence. attribute), 59
this_trial_n (slab.Staircase. attribute), 63
threshold() (slab.Staircase method), 64
times (slab.Signal property), 51
times (slab.Signal. attribute), 51
tone() (slab.Binaural static method), 58
tone() (slab.Sound static method), 39
transitions() (slab.Trialsequence method), 60
trials (slab.Trialsequence. attribute), 59
Trialsequence (class in slab), 59
trim() (slab.Signal method), 52
V
vocode() (slab.Sound method), 49
vowel() (slab.Binaural static method), 58
vowel() (slab.Sound static method), 44
vsi() (slab.HRTF method), 77
W
waveform() (slab.Sound method), 47
whitenoise() (slab.Binaural static method), 58
whitenoise() (slab.Sound static method), 41
write() (slab.Precomputed method), 67
write() (slab.ResultsFile method), 67
write() (slab.Sound method), 45

Index 81